International Journal of Science and Research (IJSR)
ISSN: 2319-7064
Impact Factor 2024: 7.101

Using CQRS and Event Sourcing in the Architecture of Complex Software Solutions

Mantu Singh

Software Architect, Reged, Acton, USA

Abstract: ***The article explores the theoretical and practical aspects of migrating complex software solutions built on traditional Domain - Driven Design to an architecture based on Command Query Responsibility Segregation (CQRS) combined with Event Sourcing. The analysis focuses on the key advantages and existing limitations of these approaches, including their impact on code complexity metrics, system performance, and infrastructure requirements (Event Store, message brokers). As an example, the article references the results of an experiment conducted by other researchers, comparing an initial task - tracking system with its CQRS and Event Sourcing - based version, demonstrating improved scalability alongside increased infrastructure complexity. The study provides recommendations on migration strategies (evolutionary or "cold"), methods for ensuring idempotency, and best practices for configuring monitoring and testing tools. The findings presented in the article will be of interest to researchers, software architects, and practitioners involved in the development of distributed systems seeking to implement CQRS and Event Sourcing paradigms to enhance scalability, fault tolerance, and consistency in complex software solutions within dynamic business environments.***

Keywords: CQRS, Event Sourcing, Domain - Driven Design, cyclomatic complexity, architecture migration, high - load systems, idempotency, monitoring.

1. Introduction

Information systems are increasingly facing challenges related to the growing complexity of domain logic and the rising demands for scalability and fault tolerance. Traditional monolithic architecture or the classic Domain - Driven Design (DDD) approach can lead to dependencies, increased maintenance costs, and difficulties in rapidly updating functionality. As a result, in recent years, attention has shifted toward the Command Query Responsibility Segregation (CQRS) and Event Sourcing approaches, which enhance architectural flexibility by clearly separating read and write operations while preserving all state changes as a sequence of events.

Lytvynov O. and Frolov M. [1] propose a model for migrating domain - driven design principles to a CQRS context with Event Sourcing, aiming to enhance architectural flexibility and extensibility. The scientific novelty of their study lies in formulating a new paradigm for adapting existing domain models to event streams. In parallel, Shkryabin G. D. [6] focuses on applying CQRS and Event Sourcing in high - load systems, substantiating the hypothesis that performance can be improved through parallel processing of commands and events, addressing the existing research gap in ensuring the scalability of distributed computing processes. Additionally, Youssfi M. et al. [8] introduce a middleware model for multi - agent systems, where the scientific novelty lies in integrating a microservices approach with event - based architectures, enabling dynamic system management and adaptation to changing loads. Their methodology is based on a combined analysis of architectural patterns and empirical hypothesis testing.

Alongside integration research, some authors focus on optimizing performance and designing event - driven systems. Ok E. and Eniola J. [2] demonstrate the use of event - driven architecture for real - time data streaming in microservice - based systems, where the goal is to enhance the responsiveness of information processing. Similarly, Stopford B. [4] systematizes the principles of designing event - driven systems, hypothesizing that applying these principles can contribute to the development of adaptive and resilient architectures.

A separate area of research in the literature is dedicated to analyzing event logs and identifying causal relationships in CQRS and Event Sourcing - based architectures. Lytvynov O. A. and Hruzin D. L. [3] focus their study on significant events that directly impact system correctness and stability, allowing for the identification of bottlenecks in command and event flows. Breitmayer M. et al. [5] propose advanced methods for extracting event logs from legacy software systems using process mining, representing a significant step forward in reconstructing event streams and transforming outdated architectures, thereby filling the gap in systematic analysis and modeling of evolutionary processes.

Finally, in the context of formalizing architectural approaches and defining modalities, Jejić O., Škembarević M., and Babarogić S. [7] propose a systematic methodology for classifying architectural solutions based on the Event Sourcing pattern. Their objective is to develop a unified terminology framework and formal criteria for evaluating the efficiency of architectural models, which, according to the authors, facilitates the optimization of development and implementation processes for complex systems.

The research gap arises from the fact that despite the existing studies on DDD, CQRS, and Event Sourcing, there is no comprehensive analysis demonstrating the relationship between code complexity metrics, scalability, and actual performance improvements when transitioning from a traditional DDD architecture to CQRS with Event Sourcing. A systematic study is needed that considers the staged migration process and risk mitigation methods (e. g., idempotency issues and event versioning).

The objective of this study is to develop and experimentally validate a methodological approach for migrating complex

software solutions based on DDD to a CQRS and Event Sourcing architecture, evaluating the impact of this transition on complexity metrics, performance, and system manageability.

The scientific novelty lies in analyzing existing research on CQRS and Event Sourcing in the architecture of complex software solutions, identifying strengths and weaknesses, and providing recommendations for their implementation.

The author's hypothesis is that transitioning to CQRS and Event Sourcing for systems initially built on classic DDD enhances performance by reducing write conflicts and optimizing read operations. While the overall number of modules and lines of code increases, the overall system complexity either decreases or remains the same due to a clearer separation of concerns.
A comparative analysis of previous studies was conducted as part of this research.

2. Theoretical Foundations of CQRS and event sourcing in the context of complex systems

Domain - Driven Design (DDD) was introduced in 2004 and has become one of the most influential approaches to designing complex systems. The core idea of DDD is to focus on the domain and develop a model that accurately reflects business rules and logic.
One of the key concepts in DDD is "bounded contexts, " which define boundaries of responsibility within a system. Each context contains its own domain model and ensures that modules within it evolve independently of other parts of the system. Objects in DDD are grouped into aggregates—aggregate roots—with defined lifecycles and encapsulation rules [1].
Experience shows that DDD is valuable in developing enterprise applications with complex business logic [3]. However, as the number of entities and interactions within a domain increases, transaction management becomes more challenging, and scaling under high loads presents additional difficulties. As a solution, the literature increasingly discusses combining DDD with Command Query Responsibility Segregation (CQRS) and Event Sourcing [1].

Command Query Responsibility Segregation (CQRS) was introduced in the context of event - driven systems as a way to offload monolithic logic by dividing it into two clearly separated parts:

- Commands: Modify the system state (create, update, delete).
- Queries: Return data in a read - only format.

The fundamental idea is to eliminate resource contention when attempting to read and modify the same objects simultaneously. CQRS allows for specialized optimization of the write model and read model, enhancing overall system performance [8].

CQRS, when combined with microservices, improves scalability: services handling commands can be deployed separately from services processing read requests, allowing independent resource allocation for different workloads. However, careful planning of API contracts and event formats is necessary to maintain data consistency between the write and read sides.

Event Sourcing replaces traditional CRUD - based snapshots of domain entities with a sequential record of immutable events. The current state of an entity can be reconstructed by replaying events from its creation onward [4].

To provide a structured comparison of the three approaches—DDD, CQRS, and Event Sourcing—Table 1 presents their key characteristics.

Table 1: Comparative characteristics of DDD, CQRS, and Event Sourcing [1, 3, 4, 7].

Criterion	DDD	CQRS	Event Sourcing
Main Focus	Domain model considering business logic (bounded contexts, aggregates)	Separation of reading (queries) and writing (commands)	Storing a sequence of events (event log) instead of "state snapshots"
Advantages	- Unified object model - Convenience with moderate complexity	- Independent scalability - Full history of changes - Flexibility in integration - Easy "rollback" (replay)	- Optimized for reading and writing
Challenges	- Complexity growth during scaling - Clear aggregation model needed - Difficulty synchronizing read/write models	- More complex architecture - Large volumes of stored data - Migration and replay complications	- Requires thoughtful idempotency mechanisms
Use Cases	Corporate systems with moderate complexity	High - load microservices with large numbers of operations	Systems requiring full history (fintech, auditing, analytics)
Relationship to DDD	This is a basic conceptual framework	Supplement to DDD focusing on responsibility segregation	Often used in combination with DDD/CQRS to enhance transparency and manageability

Thus, CQRS and Event Sourcing represent an evolution of DDD principles, addressing scalability challenges and the increasing complexity of business logic. Their application is effective in systems that need to process high data volumes while ensuring a high level of auditability and traceability. However, successful implementation of these patterns requires consideration of additional factors, including the increased number of modules and events, infrastructure demands for storage and replication, and the need to ensure idempotency and event version consistency.

3. Practical Aspects of Migrating Complex Systems to CQRS and Event Sourcing

This section explores the main stages and methodologies for transitioning from a traditional architecture (e. g., DDD without an event - driven approach) to CQRS and Event Sourcing. It addresses migration strategy selection, infrastructure preparation, as well as risks and metrics that help evaluate the effectiveness of the changes. Below, Figure

1 illustrates the process of implementing CQRS and Event Sourcing.

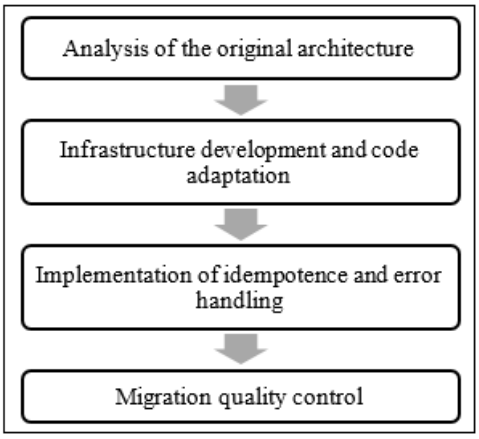

Figure 1: The process of implementing CQRS and Event Sourcing [1].

As shown in Figure 1, the first stage in implementing CQRS and Event Sourcing involves analyzing the existing architecture, taking into account:

- The level of module coupling (tight coupling), which manifests through high cyclomatic complexity and difficulty in making changes [1, 6].
- Load characteristics (number of concurrent write operations, read request volume, latency requirements), which indicate scalability bottlenecks.
- Business logic complexity: the more dynamic and intricate the model, the greater the value of an event - driven approach with full change history storage [3, 7].

Different migration strategies are described in the literature: incremental/evolutionary migration and "cold turkey" migration, where the system is temporarily shut down and restarted with a new repository and business logic [5]. The incremental approach mitigates risks for critical systems but requires simultaneous support of both the old and new storage models. Table 2 compares the characteristics of evolutionary and "cold" migrations.

Table 2: Comparative features of evolutionary and "cold" migration [1, 3, 5].

Criterion	Evolutionary (Incremental)	"Cold Turkey"
Migration Principle	Gradual transfer of functionality, parallel operation of old and new models	Complete system shutdown with subsequent launch on new architecture
Risks	- Synchronization difficulties - Increased infrastructure requirements (dual data storage) - Potential version conflicts	- Prolonged system downtime - Extensive preparatory work needed
Advantages	- No long downtime - Rollback to previous stage in case of problems - Quick migration finalization - Cleaner adaptation to Event Sourcing	- Faster migration completion - More "clean" transition to Event Sourcing
Use Cases	Systems where prolonged downtime is not acceptable (e - commerce, SaaS platforms)	Internal corporate systems with planned downtime capability (some ERP systems)

As seen in Table 2, the strategy choice depends on the ability to support parallel data storage and event processing (evolutionary migration) or the team's readiness for temporary system shutdown ("cold" migration). In industrial practice, a hybrid approach is often used: some systems are migrated incrementally, while others undergo a "cold turkey" transition [4, 5].

The next stage of implementation involves infrastructure development and code adaptation. CQRS and Event Sourcing require specialized infrastructure for event storage and processing:

- Event Store: A storage solution must be selected (e. g., relational databases, specialized NoSQL solutions, or ready - made options like EventStoreDB). The main requirements include fault tolerance, support for large event logs, and replication capabilities.
- Message Broker: Kafka, RabbitMQ, or AWS Kinesis facilitate event stream separation and subscriber scalability. In CQRS, message brokers simplify read model projections through asynchronous message delivery [2].
- During domain service modification, the following processes take place:
- Separation of domain logic: Operations modifying system state are implemented as command handlers, while read operations are assigned to query handlers. Classic DDD services are split into narrower functional classes.
- Command creation: Each command represents an intention to modify system state (e. g., CreateOrderCommand). Command objects contain the necessary business logic data.
- Event processing: Upon successful command execution, corresponding events are generated (e. g., OrderCreatedEvent). These events are stored in the Event Store and published to the message queue for subscribers.
- Read models (projections) are built to ensure quick data access and are typically stored in separate tables or databases. Event consumers process events and update "flattened" structures optimized for querying. In some cases, read models aggregate multiple tables based on business rules [4].
- The next step involves implementing idempotency and error handling. In distributed systems, duplicate event deliveries can occur due to network failures or retry mechanisms. Each event handler must ensure idempotency: repeated processing should not disrupt data consistency or lead to duplicate operations. Fault tolerance mechanisms include:
- Dead - letter Queue (DLQ): If an event cannot be processed correctly, it is placed in a special queue for manual review or delayed processing [4].
- Manual or semi - automated replay: In case of critical failures, events can be "replayed" from a specific checkpoint. This requires strict monitoring of event sequence and, if schema changes occur, transformation logic [1, 5].
- During the migration quality assessment, cyclomatic complexity analysis is conducted to determine whether module dependencies have decreased and whether system architecture has become more transparent. To enhance performance, it is recommended to:

- Analyze command (Create, Update, Delete) and query (Read) execution time—measured using standard scenarios in a test environment.
- Evaluate event - to - view latency: If the read model is built asynchronously, a time lag may be required for data consistency.
- Monitor resource consumption (CPU, I/O, memory) in the Event Store and message broker.

Thus, transitioning to CQRS and Event Sourcing requires a comprehensive approach, starting with a detailed analysis of the existing architecture, selecting a migration strategy, and concluding with ensuring idempotency and a reliable monitoring system. The next section will explore the effectiveness of migration by analyzing experimental data obtained from conducted research [1], comparing complexity and performance metrics, and discussing identified advantages and limitations.

4. Evaluation of effectiveness and practical results

Litvinov O. and Frolov M. [1] conducted an experiment comparing a task tracking system (TaskTrackingSystem) implemented using "classic" Domain - Driven Design (DDD) with a system that adopted CQRS and Event Sourcing. The quality of migration was assessed based on the following parameters:

- Number of classes and lines of code (SLOC).
- Cyclomatic complexity.
- Number of modules in business logic.

As a result of implementing CQRS and Event Sourcing, the total number of classes increased significantly (from approximately 47 to 213); however, the overall cyclomatic complexity of the system decreased (from 534 to 522). The reason for this is that while additional infrastructure classes (commands, event handlers, etc.) are introduced, each individual module becomes simpler and is responsible for only a limited portion of the logic. Table 3 presents summarized results for key operations.

Table 3: Comparison of average execution times (in milliseconds) for basic operations [1]

Method	DDD (ms)	CQRS+ES (ms)	Performance Gain
getUsers	281	43	Read speed increased more than sixfold
addUser	28	48	Slight increase in write time
updateUser	119	46	Almost 2.5 times faster
deleteUser	71	52	Around 25% improvement
getUser	37	37	No change in single record read time

From Table 3, the following conclusions can be drawn:

- Faster read operations (getUsers, getUser). CQRS and Event Sourcing significantly benefit bulk read operations (list queries). Specifically, getUsers performed on average more than six times faster.
- Write performance (create/update/delete) varies. Some operations demonstrate acceleration (updateUser), while others experience slight slowdowns (addUser). Overall, the CQRS concept optimizes read queries by shifting logic to projections, whereas the command processing layer may introduce infrastructure overhead (event generation, storage, and notification of other services).

While the overall cyclomatic complexity may decrease, the number of classes and lines of code increases, which implies:

- Additional requirements for version control and testing. Larger codebases require extensive module validation, especially when dealing with event - based workflows.
- Higher training costs for developers. Understanding Event Store principles, microservices communication, and CQRS patterns is necessary.

When transitioning from a traditional storage model to Event Sourcing, careful migration of historical data is required. A common approach involves an interim phase where new changes are recorded in the Event Store while queries continue reading from the "old" database until query modules are fully refactored.

It is recommended to maintain a complete event log, which simplifies the implementation of real - time analytics, for example, using Kafka Streams or Apache Flink. This enables the development of predictive models and timely responses to system state changes. Monitoring (observability) through centralized logging (e. g., ELK Stack) and event tracing systems (Jaeger, OpenTelemetry) facilitates faster failure detection and performance tracking. For large - scale projects, mechanisms such as Event Store and broker replication, distributed event processing, and geographic data center partitioning may be required. At the same time, ensuring idempotency of event handlers and designing an intelligent retry system are crucial for maintaining system reliability.

Thus, the final effectiveness and benefits of implementing CQRS and Event Sourcing largely depend on proper migration planning, a well - configured infrastructure (Event Store, message brokers), and effective management of testing and monitoring processes.

5. Conclusion

This study conducted an extensive analysis of the transition from a traditional DDD architecture to CQRS and Event Sourcing, covering theoretical foundations, practical implementation aspects, and an evaluation of results based on an experimental project. The findings confirm that the combined use of CQRS and Event Sourcing provides:

- Improved read performance due to dedicated projections and asynchronous query processing.
- Enhanced scalability and flexibility in managing domain logic through the separation of commands and queries, as well as by storing the complete history of changes as events.
- Increased transparency and auditability, as the system retains all events, simplifies state recovery (replay), and facilitates integration with other services.

At the same time, the study highlights risks and challenges associated with migration:

- Increased infrastructure and code complexity due to the addition of components such as command handlers, event stores, and event handlers.
- The need for a reliable data migration strategy when transitioning from a traditional database to an event - based storage model.
- Higher requirements for monitoring and testing, including ensuring idempotency and correct handling of duplicate events.

Thus, the decision to adopt CQRS and Event Sourcing should be based on system scale, the criticality of fast read operations, and the necessity of maintaining a complete change history. To mitigate risks, a phased migration approach is recommended, along with the use of centralized observability tools and the early preparation of development teams for event - driven patterns. The obtained results can serve as a foundation for further research into optimizing Event Store performance and expanding analytical capabilities for real - time event stream processing.

References

[1] Litvinov O., Orlov M. On the transition from domain - specific design to CQRS with Event - Sourcing software architecture //Information technology: computer science, software engineering and cybersecurity. - 2024. – Vol.1. – pp.50 - 60.
[2] Ok E., Eniola J. Optimizing Performance: Implementing Event - Driven Architecture for Real - Time Data Streaming in Microservices. – 2024. – pp.1 - 15.
[3] Lytvynov O. A., Hruzin D. L. Critical causal events in systems based on CQRS with event sourcing architecture //Radio Electronics, Computer Science, Control. – 2024. – Vol.3. – pp.119 - 119.
[4] Stopford B. Designing event - driven systems. – O'Reilly Media, Incorporated. - 2018. – pp.5 - 40.
[5] Breitmayer M. et al. Deriving event logs from legacy software systems //International Conference on Process Mining. – Cham: Springer Nature Switzerland, 2022. – pp.409 - 421.
[6] Shkriabin G. D. Application of CQRS and Event Sourcing patterns in Event - Driven architecture: experience in developing highly loaded systems //Bulletin of Science. – 2025. – Vol.1 (82). – pp.280 - 295.
[7] Jejić O., Škembarević M., Babarogić S. Defining Software Architecture Modalities Based on Event Sourcing Architecture Pattern //European Conference on Advances in Databases and Information Systems. – Cham: Springer International Publishing. - 2022. – pp.450 - 458.
[8] Youssfi M. et al. Multi - Micro - Agent System middleware model based on event sourcing and CQRS patterns //Smart Trajectories. – CRC Press. - 2022. – pp.25 - 46.