UDC 614.2 + 574/578 + 004.38
DOI: 10.15587/2706-5448.2025.337168

**Oleksandr Lytvynov,
Dmytro Hruzin**

DECISION-MAKING ON COMMAND QUERY RESPONSIBILITY SEGREGATION WITH EVENT SOURCING ARCHITECTURAL VARIATIONS

The object of the research is the process of selecting and evaluating architectural solutions, both at the design stage and during the migration of a software application's architecture, within the context of evolutionary architecture. The paper is focused on variations of the Command Query Responsibility Segregation (CQRS) with Event Sourcing (ES) architecture, which, in fact, is a family of architectural variations that differ in complexity, performance, development time, and the required expertise from developers. These differences have a significant impact on the development cost and maintainability of the software application. Moreover, changes in business requirements or technical context often necessitate migration among architectural variations, which may drastically increase costs if not planned properly.

In the absence of objective evaluation criteria, decisions are often based on expert judgment, which may be unavailable or insufficient. This work proposes a decision-making support approach for CQRS with ES architectural variation selection and migration planning. The approach is based on classification of processes and breaking them down into smaller activities. This enables objective comparisons of architectural variations based on complexity and performance metrics.

The application of the approach is shown on two basic variations. Metrics were obtained, and a bitmap chart was built to visualize architectural applicability, depending on the project priorities. The applicability score of mCQRS ranges from 39% to 53%, while that of Classical CQRS – 47–61%.

The proposed approach is applicable in projects where architecture evolution is expected. It is especially useful in organizations operating at Capability Maturity Models Integration (CMMI) Level 4 (Quantitatively Managed Organization) which is focused on predictability of quantitative performance improvement objectives.

Keywords: *software architectures, software metrics, formal methods, support decision-making, CQRS, Event Sourcing.*

Received: 27.05.2025
Received in revised form: 17.07.2025
Accepted: 03.08.2025
Published: 29.08.2025

How to cite
Lytvynov, O., Hruzin, D. (2025). Decision-making on Command Query Responsibility Segregation with Event Sourcing architectural variations. Technology Audit and Production Reserves, 4 (2 (84)), 37–59. https://doi.org/10.15587/2706-5448.2025.337168

1. Introduction

The complexity of software applications 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 like [1–3] provide a lot of patterns and solutions focused on how to manage the complexity of modern business-oriented software applications, making them more flexible, scalable and maintainable.

Command Query Responsibility Segregation (CQRS) with Event Sourcing (ES) architecture [4, 5], which was proposed in [6–8], can be considered as one of the most powerful modern solutions for building high-quality software applications. It is a variation of a more general Event-driven architecture paradigm [9] with a concentration on increasing application's performance by separating the handling of commands (i. e., write operations) and queries (i. e., read operations), which means that the results of Write operations don't return to the caller directly, but rather delivered via notification-oriented asynchronous mechanism. The advantages of the CQRS with ES architecture compared to the conventional approach [10] not only include improved performance. It also includes better flexibility and scalability due to asynchronous event processing and reduced risk of conflicts when making changes. 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.

However, the pure variation of CQRS with ES architecture is not without drawbacks. The attempts to resolve the issues led to variations of the approach. The applicability of the variations depends not only on the project requirements, but also on the time and cost limits of the project. The developers are faced with the problem of choosing the most suitable architectural variation for the concrete project. The situation is complicated by the fact that the requirements and project budget can change during the development and maintenance phases of the project life cycle. It may cause not only technological, but also architectural evolution or even migration of the project.

The problem of architectural evolution of the software seems to be one of the important software development problems. In accordance with [11], the architecture should be ready to react to changing requirements, as well as feedback from end-users and developers. This reaction should take the form of guided, incremental changes across multiple dimensions (technical, data, security etc.). These changes should allow the architecture to evolve toward more effective solutions. However, the architecture should not evolve in ways that would harm any important architectural concern. Briefly, the main point of the Evolutionary

Architecture approach is that evolution should be controllable using different mechanisms (e. g., fitness functions).

In the case of CQRS with ES architectural variations it means that managers and developers need to ensure that the choice of a certain variation is objectively correct, or architectural modifications are objectively necessary.

The CQRS with ES architecture enables the development of flexible applications with high levels of maintainability and performance. But its original variation [7] has the problems, which, in turn, can be divided into the three following groups: technical development issues, development and maintenance complexity, architectural modifications at later stages of development.

Technical development issues associated with pure CQRS architecture are described below. Known solutions are provided along with the minimum contextual information required to understand each problem and its corresponding solution.

Write requests can be divided into two broad classes: creation-oriented and update-oriented commands. For update-oriented commands at the aggregate fetching phase, there could be a situation when the Repository attempts to retrieve an aggregate from the Cache and finds it missing (Fig. 1). In that case, the Repository creates a new object of the specified type. It then receives all events related to this aggregate instance from the Event Store that have occurred since its initial creation up to the required version. The Repository applies all retrieved events to the newly created object and saves the object to the Cache. The process of retrieving events from ES and applying them to the created object is called Replaying Events.

The advantage of the Replaying Events mechanism is the ability to realize the Undo operation. This operation allows rolling back the modifications of an aggregate to a certain version. It is done by creating compensating events based on the existing ones. These compensating events are then added to the Event Store [12]. In this case, the persisted information is only the collection of events stored within the ES. It is said that ES plays a role as a Source of Truth for a certain aggregate, i. e., the actual, consistent version of an aggregate can be constructed, relying on the events from the ES. Common practice is that each aggregate has its own ES, but there could be different variations as well.

Due to various reasons, such as server restart, removal of the aggregate from the cache, or some errors during the aggregate's state updates, it becomes necessary to perform event replay. 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 application performance. Thus, in the case of a large number of events and a large number of aggregates replaying events process starts to cause performance issues.

As a solution to this problem, it is suggested to use an additional data store that holds a snapshot of the aggregate's state at a specific moment, e. g., after saving every hundred or thousand events [13]. Thus, the assembly of the aggregate is based on the acquired snapshot and only the events that occurred after the snapshot was taken are replayed.

For example, the application discussed in [14] takes a snapshot for every 100 events (i. e., the threshold. The threshold can be different from application to application and even within one application there could be different thresholds and event threshold tuning mechanism for different aggregates. This is important because frequent snapshot creation can slow down the application's performance.

The second problem of the Event Sourcing approach [15] is handling the event types' versioning. The Event Store keeps all the events, but the events written some years ago may be different in structure from the events currently used; the interpretation of the events can be changed over time. Thus, the handling of the different versions of events is a non-trivial problem. In [15] proposed a number of solutions were proposed. The most sustainable is seemingly doing a transformation of event storage on every release. As a result, all old version events get transformed into the events of a new version. But, firstly, this method needs appropriate infrastructure to automate the process, which is also not a trivial task, and, secondly, it needs much time to install this sort of software update. Another solution is to update event handlers to process a variety of event versions, which also increases the complexity of the change's implementation in response to evolving requirements. As a result, the development increases over time and becomes more complex.

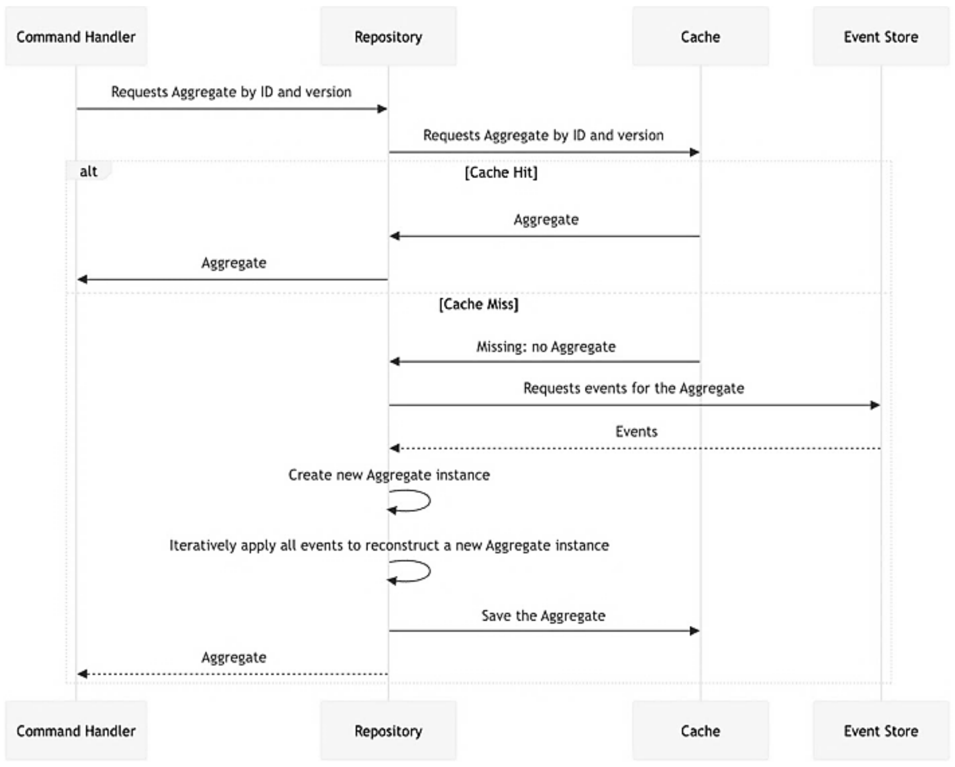

Command Handler
Repository
Cache
Event Store
Requests Aggregate by ID and version
Requests Aggregate by ID and version
alt
[Cache Hit]
Aggregate
Aggregate
[Cache Miss]
Missing: no Aggregate
Requests events for the Aggregate
Events
Create new Aggregate instance
Iteratively apply all events to reconstruct a new Aggregate instance
Save the Aggregate
Aggregate
Command Handler
Repository
Cache
Event Store

Fig. 1. Sequence diagram of the Fetching Aggregate workflow

The third problem is the immutability of events [16], which concerns the event-sourcing system. In accordance with the pure CQRS principles, since the event store is the source of truth, all the events are immutable, i. e., the events in the ES cannot be modified and cannot be removed from the ES as well. To leverage the principle, the ES can be realized using blockchain [17] technology rather than a traditional database approach. But in practice, the situations when the event modification is required may arise. For example, a user requests the removal or anonymization of all user's data from the system based on the GDPR [18]. In such cases, it is required to release a new version of the Event Store. This includes extracting all data from the current Event Store, converting it into events (in case they are encrypted), modifying the user's data using a transformation function, and storing all events in the new Event Store [15]. This significantly complicates a task that is trivial in traditional architectures, such as deleting several records from a database.

The fourth issue is connected to the event delivery subsystem based on Event Bus, which does not guarantee that the events will be delivered in the order they were published. For example, let's assume that two account transactions, namely deposit and withdrawal events, were published in order <withdrawal, deposit> (deposit after withdrawal). Account events have some information about the state of the account (Balance, TotalBonus, TotalDeposit, TotalFee, etc.). For instance, the Projection that subscribed to the Account events is responsible for reporting the balance changing dynamics relying on the Balance property of the event message. In this case, the getting ahead of the deposit event, which was published second, may lead the Projection to a wrong calculation based on the Balance value from the payload of the event (Table 1).

Table 1

Events publishing/delivery timeline

Time	ES	Event	Projection
*t*1	publish	*e*1 – Balance = 100000	0
*t*2	publish	*e*2 – Balance = 80000	0
*t*3	delivery	*e*2 – Balance = 80000	80000
*t*4	delivery	*e*1 – Balance = 100000	100000

To avoid such situations, the projection stores the version of the aggregate state according to which it was updated [15, 19]. In some cases (for example, when the version received in the event is higher than the expected version of the projection), a version mismatch error may occur. There can be different ways to handle such an error depending on the application's characteristics. However, typically several retry attempts are made with a certain delay in case the events arrive in the wrong order. If there is a situation when an event with a lower version is completely missed (e. g., because of some technical issues or an unexpected error), the error log is saved, and the operation is canceled. Such situations are usually resolved manually by developers, and the projection is typically rebuilt.

The complexity challenge is discussed in [20]. While the CQRS pattern itself is relatively simple, its combination with other approaches like Domain-Driven Design (DDD) or ES, as examined in the study, drastically increases the architectural complexity of the software application. As it was shown in [21], the transition of the solution from DDD to CQRS architecture results in an increase in the number of classes, modules, and layers. It means that adding or updating features in the case of CQRS with ES takes more time and requires a higher level of skills from the development team. This leads to longer timelines and increased costs, which significantly reduces its attractiveness to investors.

The challenges of development and maintenance complexity in pure CQRS architecture are typically addressed at a more abstract level. This is done through team training, improved project documentation, increased code coverage, and enhancements in code review and delivery processes. Often, pure CQRS architecture is not applied to the entire software system but only to specific parts of it. For instance, in Microservices Architecture [22], pure CQRS may be applied to some services where this approach is most beneficial, while other services are built using different architectural solutions. However, the complexity of developing and maintaining these services remains significantly higher compared to services developed using DDD.

The listed CQRS with ES challenges have different solutions, but each applied solution increases the application's development time and, consequently, its cost. Thus, applying the most expensive variation of CQRS, which resolves all the issues to any project independently of its type and without a thorough analysis of its requirements, seems ineffective. Based on project statistics from DBB Software [23], not all applications require solving all these issues. It is worth noting that some of the projects do not even need some architectural features (e. g., replaying events) throughout the lifecycle, but some require features that were not implemented at the further development stages. It makes developers face the problems of architectural flexibility, predicting and evaluating the costs of migration from cheap variations to more expensive ones, and vice versa.

It also remains unclear how to evaluate which solutions should be applied to a certain project and why, and what the cost of applying a particular solution would be in terms of time and effort. Thus, the objective decision-making based on approach can facilitate the process of CQRS-based applications development and maintenance.

Some resolutions of technical development issues (event replay performance problems during aggregate assembly, the complexity of handling event versioning, etc.) and methods for reducing development and maintenance complexity have led to the emergence of numerous variations of the CQRS with ES architecture.

Currently, these variations lack formal descriptions and names, however, they actually form a set of architectural variations, which could be regarded as different stages of CQRS with ES architecture evolution. It corresponds with the Evolutionary Architecture concept, which assumes the ability of software architecture to change in order to produce more effective solutions, but disallows that evolution to harm some architectural concern.

The distance between the variations of CQRS with ES may be defined in terms of different practices used to resolve the issues (e. g., use snapshots derivation or not, release event store every time, or handle all event type versions, etc.). These practices, in turn, can be evaluated using such criteria as complexity, performance, development time, cost etc. This evaluation allows choosing an appropriate variation in response to the project requirements at the development stage [24]. It also helps assess the necessity and to estimate the cost of the potential migration from one architectural variation to another at the maintenance phase (utilization and support stages [24]) of the project's lifecycle. Such migration may be caused by different reasons (e. g., business agility and requirements change, incorrect initial choice).

To date, unfortunately, there are no well-known methods which could be directly applied to facilitate decision-making on the objective selection of CQRS with ES architectural variation during the development and maintenance phases of the application's lifecycle. However, there are general approaches and methods [25, 26] that support decision-making in software architecture.

All the approaches can be classified in two major ways. Firstly, they could be divided into two broad categories in accordance with the life-cycle phases: Design-time and Run-time. Design-time methods are applied during the software application planning phase, while Run-time methods help assess the architecture of existing applications and provide insights for potential modifications or improvements. Secondly, the approaches can be classified by application characteristics as follows: utility-based, scenario-based, parametric-based, search-based, economics-based, and learning-based.

Utility-based methods adopt utility theory to shortlist candidate architectures. The challenges of this approach include the complexity of defining the utility function and quantifying utility values.

Scenario-based methods evaluate alternatives based on the identification of quality attributes and their quantitative assessment by stakeholders. Practical examples include SAAM [27], ATAM [28], CBAM [29], ATMIS [30], etc. These methods rely on exploratory, evolutionary, and stress scenarios to test the potential challenges the architecture may encounter, such as sudden load spikes or partial network failures etc. A major drawback of this category is the heavy reliance on the quality of scenario descriptions and expert assessments from stakeholders.

Parametric-based methods use a parameterized mathematical model of the system and compare approaches based on their parameters. Examples include Multiple-Criteria Decision Analysis (MCDA) [31] methods such as Analytic Hierarchy Process (AHP) [32] and AHP-based methods like ArchDesigner [33] and LiVASAE [34]. These methods typically do not provide specific parameters for evaluation or ways to obtain parameter values. Instead, they rely on the presence of quality attributes and their quantitative assessment, which is often performed through expert evaluations by stakeholders. The main drawbacks are the uncertainty in obtaining input data and the resulting high dependence on stakeholder assessments.

Methods of search-based category employ mathematical search algorithms to find the most suitable option. The challenges of this category include uncertainty in defining search keywords and the "good enough" stopping condition for candidate testing. Additionally, these methods are rarely applicable during the design phase.

Economics-based methods incorporate approaches from various other categories, emphasizing the importance of cost assessment.

Learning-based methods adapt machine learning techniques for software architecture evaluation. They generally belong to the Run-time category. The main drawback of this category is the difficulty in verifying the correctness of machine learning models.

Based on the problem definition, to address it the Design-time methods should be used. The most suitable categories are scenario-based and parametric-based methods, as they evaluate multiple candidate approaches based on quality attributes. Let's consider their use cases.

Scenario-based methods (e. g., ATAM, CBAM, ATMIS, etc.) work well when choosing between fundamentally different architectural solutions, such as monolithic vs. microservices architecture or relational databases vs. event sourcing. In such cases, stakeholders can relatively easily identify quality attributes and provide expert assessments. While these methods remain applicable for selecting architectural variations, their use becomes more challenging due to the small differences in variation parameters, making it difficult to quantify quality attributes.

Parametric-based methods (e. g., AHP, ArchDesigner, LiVASAE, etc.) work with predefined parameters or, like scenario-based methods, suggest identifying and assessing parameters through stakeholder evaluations. However, their objective application during the software application design phase often suffers from insufficient system information.

The common drawbacks of all the reviewed methods in the context of solving the problem are:

1) the difficulty of defining quality attributes when comparing similar approaches;

2) the high reliance on stakeholders for determining the relative importance (i. e., ranking) of architectural decisions and their impact on quality attributes.

Therefore, the direct application of the reviewed methods for decision-making on selecting a CQRS with ES architecture variation does not provide a sufficient level of accuracy.

Thus, *the aim of this research* is to develop an approach for the evaluation and comparison of CQRS with ES architectural variations based on quantitative metrics and an analysis of specific project requirements, priorities, and constraints.

This will reduce uncertainty and allow more informed architectural decision-making when selecting a CQRS with ES variation, both at the design stage and during the application's evolution and maintenance, subsequently lowering the cost of software system development.

To achieve this aim, the following objectives are accomplished:

1. To define a mechanism for formalizing the processes that constitute a given CQRS with ES architectural variation.
2. To identify the key parameters that influence the evaluation and selection of CQRS with ES architectural variations and to analyze the processes of the variations based on these parameters, synthesizing an overall assessment of each variation.
3. To compare the applicability of architectural variations to a software project based on their parameters and the project's requirements, priorities, and constraints.
4. To build an evolutionary roadmap for the CQRS with ES architecture and assess the effort required for architectural transitions between its evolutionary stages (variations).

2. Materials and Methods

2.1. Research strategy

2.1.1. The object and hypothesis of research

The object of this research is the process of selecting and evaluation of architectural solutions, aimed at selecting the optimal variation for software applications with defined requirements. Such decisions are relevant both at the system design stage and during the migration of a software application's architecture within the context of evolutionary architecture.

The main hypothesis of the study is that the decision-making process for selecting a CQRS with ES architectural variation can be improved through the following workflow:

- identifying and classifying use case types;
- defining processes for each class, and formally describing them using a formal activity-centric knowledge representation model;
- specifying algorithms and quantifying parameters for each activity;
- and applying a structured MCDA method.

It becomes possible to make a more objective selection of an architectural variation at the design stage, as well as to choose the appropriate next evolutionary step during migration.

Assumptions made in the study are:

1. The software systems under consideration follow a modular architecture and can be decomposed into distinct command and query processes.
2. The command and query processes can be described as unidirectional, non-branching chains and formalized using formal methods.
3. Key evaluation parameters such as complexity, performance, and maintainability can be quantified and normalized.

Simplifications adopted in the study are:

1. Only three architectural variations (Pure CQRS, Classical CQRS, mCQRS) are considered in the evolutionary roadmap.
2. Error handling and exception flows are not modeled, focusing exclusively on the successful execution paths of the process.
3. Performance evaluations are based solely on experiments with a Representative Test Project, rather than on statistical data obtained from real-world production systems.
4. External factors such as team skill level, infrastructure configurations, or organizational constraints are not considered.
5. The AHP method was selected for comparing applicability due to its simplicity and ease of implementation; however, in practice, it can be replaced with any other appropriate MCDA method.

To develop such an approach, the following strategy is proposed:

1. Building a model of the application under consideration using formal methods, considering all architectural variations. The goal is to

provide a strong foundation for the objective evaluation and comparison of the variations.

2. Defining quality attributes (parameters) for assessment of architectural variations, based on the proposed model.

3. Providing methods for evaluating architectural variations based on complexity and performance metrics.

4. Providing methods of choosing the appropriate variation for a certain project, considering the potential architectural evolution.

5. Providing examples of the practical use of the proposed approach. Conducting experiments and comparing results.

2.1.2. Modeling specifics

The methods of modeling the applications of this type can be divided into three categories. First is an informal architectural description represented in the form of diagrams focused on functional decomposition aspects of the application, aimed at forming the whole application view, identifying core units and their interaction mechanisms (e. g., informal textual and graphical languages, UML diagrams can be used). It's good for a superficial understanding of how the application works, but not enough to provide data for precise analysis and estimation of the solutions.

The second class of methods is related to formal system description using rigorous mathematical formalisms such as FSM, Petri Nets, and their combinations. These methods are used to describe such systems and are intended to understand the structure of the processes in terms of states, transitions, activities, and interaction specific.

The third one is based on knowledge description. Its main goal is to glue together the different pieces of knowledge presented by the above models. It helps to understand the details of the process construction, primarily focusing on the structure of activities, their classification and dependencies, and on the questions of constructing and maintaining the processes.

All three sets of interrelated models allow to see different aspects of the solutions provided, forming a foundation for their estimation and comparison.

Thus, the result is a certain hierarchy of models starting with informal analysis, from a mathematical point of view, and finishing with a detailed description of the activities-units of the processes. Activities-units are described using rigorous description formalisms, which makes the foundation of the provided decision-support approach. Both the formal and informal models of CQRS with ES architectural variations are discussed in detail in Section 2.2.

2.1.3. Defining quality attributes (parameters)

The method of architectural solution evaluation should be based on project requirements analysis and preliminary solution assessment (i. e., the method cannot rely on real code-oriented metrics).

While software design can be defined as an activity in which software requirements are analyzed to produce a description of the software's internal structure [35], software architecture is the foundation of this design. The architecture offers a blueprint (e. g., a set of patterns, principles, processes) that guides developers in structuring and implementing software components efficiently, ensuring coherence and clarity in the overall application design [36]. The goal of software architecture is to increase the quality of software, predicting and mitigating the risks of issues as early as possible (at the design and development phase). The quality of software depends on the quality of architecture.

Architecture evaluation is a milestone in the decision-making process, which aims at justifying the extent to which architecture design decisions meet application's quality requirements, considering operational uncertainties and changing requirements [25]. Different combinations of quality attributes are used to resolve this task. For example, in [37] evaluation process of monolith and microservice architectures considers Coupling, Testability, Security, Complexity, Deployability, and Availability quality attributes. According to a survey presented in [25], most of the software architecture evaluation studies have addressed multiple quality attributes, e. g., modifiability, portability, cost, etc.

The presented work focuses not on comparing different architectures, as in [37], but on comparing variations of the CQRS with ES architecture. This simplifies the task of identifying quality attributes. Most of the quality attributes used in other methods remain unchanged for variations within the same architectural approach. The two parameters that do vary when comparing architectural variations are Performance and Complexity (Modifiability) [38]. These parameters are selected as the quality attributes for applying the proposed method.

2.1.4. Evaluation method

The analytical methods of performance evaluation appropriate when the goal is to obtain approximate estimates for certain components. The overall application performance assessment should primarily be based on statistical data collected from the implementation of previous real applications. However, the systems can vary widely, being realized using different technological stacks and frameworks, according to the real tasks they solve, which can involve the correctness of estimation. Therefore, the best variant seemed to perform evaluations using Representative Test Projects (RTP) [39] derived from real projects, considering different situations, including complex scenarios (e. g., transactions, chained API calls, etc.), i. e., RTP can serve as a testing platform for the performance estimation.

The complexity of application implementation can be categorized into several key aspects: infrastructure complexity, component implementation complexity, and component maintenance/modification complexity.

The complexity of infrastructure depends on the tools (e. g., databases or servers, frameworks) and third-party services necessary for application implementation and includes a wide range of activities such as acquiring, studying, deploying, etc. Assuming that the infrastructure for all the compared variations is nearly identical, this type of complexity is disregarded in the evaluation.

For evaluating the complexity of component implementation and maintenance, there are two basic methodological categories. The first one includes methods for assessing already implemented components (e. g., [40–44]). These methods can be used for collecting statistical data (based on code analysis and estimation of its complexity) and forming typical functions and activities evaluation patterns. Another category focuses on estimating implementation complexity based on algorithm descriptions (e. g., in a form of diagrams or pseudo-code) and associated metadata (e. g., input/output parameters), for example, Object Points, Use-case Points [45] or Cognitive Functional Complexity [46].

However, the first category requires access to a large number of existing software applications based on different variations of CQRS with ES. The second approach requires the description of all functions, including their algorithms, at a certain level of abstraction. This requirement is difficult to implement in the context of real-world complex project development. At the beginning of the development process, all functions and their algorithms should be defined in advance.

It seems that a more effective way is to choose the optimal architectural variation based on the use-cases analysis. Of course, the level of description is higher than pseudo-code, and the use-cases description is the cornerstone of the development. Use-case descriptions does not require extra time and effort as in the case of algorithms, but the number of use cases can be huge, and their descriptions could be ambiguous and imprecise. To resolve these issues, the following method is proposed. In the first step, the use-cases should be classified by the specific of tasks and their realization. In the second step, the obtained classes should be mapped onto process model activities-units for each variation of CQRS with ES architecture. Then the models could be evaluated according to complexity and performance assessment criteria using the evaluation method.

Therefore, the evaluation of the performance and complexity of architectural variations is connected to the creation and maintenance of RTP derived from the real projects, considering the experience of the developers. The performance estimation is based entirely on statistics gathered using RTP. The evaluation of the architectural variation complexity is seemingly based on a combination of use cases estimation-oriented analytical, and RTP-based statistical methods.

2.1.5. Method of choosing the appropriate variation

To select an architectural variation based on metrics of performance and complexity obtained from evaluating variations and the requirements of the developed application, MCDA methods [47, 48], such as AHP or TOPSIS, can be used. It should be noted that these methods can also be applied based on expert judgment. However, in the case of comparing architectural variations, the differences in quality attribute values may be extremely small. Under such conditions, it is very difficult for experts to provide an objective evaluation, which makes the result less accurate.

Describing application processes using the proposed mathematical model enables the collection of complexity and performance metrics for architectural variations. Based on these metrics, along with application-specific data (such as the estimated number of command and query processes, read/write request ratio, etc.), evaluation parameters are calculated and normalized [49].

By applying the MCDA method with these parameters as input, an independence from a stakeholder assessment evaluation of architectural variations can be obtained.

Thus, the decision-making approach can utilize various MCDA method algorithms, including custom algorithms, considering stakeholder requirements. TOPSIS identifies the ideal solution and calculates the distance of each alternative from it. AHP evaluates each alternative relative to the others and establishes a hierarchy by giving the highest score to the best candidate. These two methods seemed to be well-suited for evaluating variations of the CQRS with ES architecture.

2.1.6. Examples and experiments

The main goal of the experimental part in this paper is to demonstrate the proposed decision-making approach for basic variations of CQRS with ES, describing the details of evaluation and decision-making specifics. Basic variations refer to those that address all major technical development issues. To make the assessment more illustrative, the selected variations are conceptually focused on different priorities – one maximizes software application performance, while the other minimizes development complexity to ensure the fastest possible implementation. The specific of the experiment is described in Section 3.

2.2. Informal description of the pure CQRS with ES architecture

This section provides the conceptual view of the pure CQRS with ES architecture, considering the specific features of CQRS presented in [7] and in the repository with the typical project [50].

From an architectural point of view, the software application is composed of three basic subsystems: The Write Model (WMS), the Read Model (RMS) and the Notification Subsystem (NS). WMS is responsible for command processing. RMS is responsible for query processing. NS is the event-based middleware between WMS and RMS. The essence of the approach can be represented by the diagram shown in Fig. 2. The command is processed by the Write Model. As a result, the WMS generates a bunch of events that come to the NS. NS is responsible for delivering the events to all the subscribers, among which are Read Model Projections. In addition to listening to events from the Notification Subsystem and responding to them, the Read Model also handles data read queries (queries), returning a Data Transfer Object (DTO) to the client.

The Write Model Subsystem is responsible for processing write requests, which include request validation, command creation, and invocation of a command handler, followed by command processing.

Typical actions performed by the command handler during command processing include retrieving aggregate, using one or more repositories, and calling a certain method of the aggregate to change its state. This method returns a set of events that should be applied to the current state of the aggregate in order to update it according to the instructions in the command. After that, the set of events is saved to the Event Store and posted to the Event Bus, i. e., passed to the Notification Subsystem.

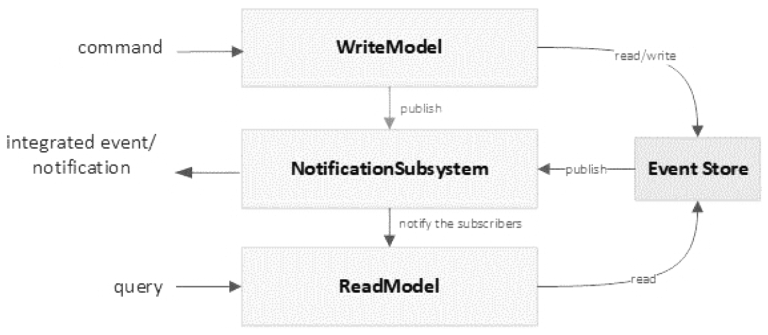


Fig. 2. Three basic subsystems and their interaction

The Notification Subsystem is a unit responsible for the delivery of the events published by command handlers of the Write Model to other subsystems and external environments (e. g., external services). It has its own gateway, which is called Event Bus, and a set of event handlers. Event handlers are responsible for the processing of events that appeared as a result of processing the command.

The interaction between Event Bus and event handlers is based on the publisher-subscriber pattern [51]. Unlike command handlers, event handlers are subscribed to certain types of events published to the Event Bus. Among several types of event handlers, the most noteworthy one is the Projection event handlers.

Projections (derived views, persistent read models) are denormalized, pre-calculated structural data representations that are specifically designed for efficient querying. The projections can be tailored to the specific needs of different read operations, improving the performance and responsiveness of the application. The process of converting (or aggregating) a stream of events into a Projection is called Projecting [52]. The implementation details of how the data is stored and accessed can vary according to the system's requirements, conditions, etc. There seem to be two polar approaches: persistence-oriented with the use of SQL, NoSQL, cloud storages; in-memory store-oriented, which is rebuilt from local event stream whenever a server is rebooted. Of course, in reality, different hybrids of these approaches are used. Thus, the Projections can be thought of more as services rather than persistent storage, tables, or databases.

The state of the projections is updated by event handlers that are subscribed to events received through the Event Bus. The event handler receives an event and updates the corresponding projection to the next version.

While Projections are being updated, the Notification Service notifies all clients about the update of the application's state.

Read Model subsystem (RMS) is responsible for query processing based on Projections. When the application receives a request to read data, the Query Handler requests data from the corresponding repository, which retrieves pre-prepared data from one or many Projections. In rare cases, the handler can work with multiple projections, aggregating the results into a user-friendly format. However, for performance optimization, the data in projections is usually stored in a denormalized form, eliminating the need for querying multiple tables and performing additional data aggregation or calculations to achieve the expected result. Additionally, caching of data at the query level can also be implemented to improve the performance, considering that the cache is invalidated after the projection is updated by the event handler.

2.3. The variations of CQRS with ES approach

2.3.1. Pure CQRS with ES using event versioning

This method is a variation of pure CQRS in which event versioning is implemented. The introduction of event versioning addresses the inability to modify event types as the system evolves. Having versions of event types gives the ability to separate older events from the newest and process each of them in a different proper way. The other reasons for the necessity of event versioning are described in greater detail in [15].

Such an application presents several complexities in development and maintenance, which were discussed in greater detail earlier in the article. Nevertheless, the performance of write operations in this application is relatively high, provided the Event Store is populated with events at a comparatively low intensity. Challenges arise when the Event Store accumulates so many events that replaying them to build an aggregate instance that is not present in the cache or to rebuild a projection starts to take considerable time. As for the Undo operation, it can be implemented relatively easily in such an application by adding a reverse event to the Event Store and resetting the cache.

2.3.2. CQRS with ES using snapshots with Event Store as a Source of Truth

This approach represents a pure CQRS approach with the application of all the standard solutions to its previously discussed issues. Hereafter, it will be referred to as the *classical CQRS* approach, which is commonly used in practice.

The implementation of such an application is more complex than the previous one, as it requires the development of a snapshot mechanism and introduces a new resource (a database for storing snapshots). In this variation, the snapshot is represented as a key-value pair or a record. The key is a combination of the aggregate ID and the snapshot version. The value is the aggregate, or even serialized aggregate, state at the specific moment in time. The generation of snapshots is based on various triggers, for example, after adding a certain number of events (reaching a threshold of the events window) or when the version of an event type changes.

The average performance of this application is slightly lower than the previous version during the initial stages, but proves advantageous once the Event Store accumulates a critical volume of events.

The Event Store remains the main source of data, which both Write and Read models rely on, and snapshots play only supporting roles, i. e., Event Store remains the Source of Truth.

2.3.3. CQRS with ES using snapshots with Snapshot database as a Source of Truth

This modification deviates from some principles of the classical approach: it violates the principle of events immutability, which means that events within the Event Store can be modified and even removed. It shifts the source of truth focus from the Event Store to the Snapshot Database and turning the Event Store into a system log. Thus, the construction of aggregate or projections rebuild procedures is unable to rely on the Replay events mechanism.

The advantage of this variation consists in reducing development complexity and excluding the issues connected with events versioning and events modification. Due to the presence of a relational database, one of the most complex mechanisms of an event-sourced application becomes unnecessary. Snapshot creation checks are eliminated. In case of a version mismatch, the latest projection version can be retrieved with a single query to the relational database, and the same applies to rebuild migrations.

At first glance, this variation is very similar to the standard approach used in DDD: fetching the data from the latest snapshot version with on-the-fly transformation. However, the principles of the command and queries segregation, projections, and events storing mechanisms remain untouched, making possible the potential evolution of the software application architecture towards classical CQRS with ES architectural solution.

In contrast to the previous method, the snapshot database in this approach closely resembles a relational database, although key-value storage with serialized aggregate state could also be used.

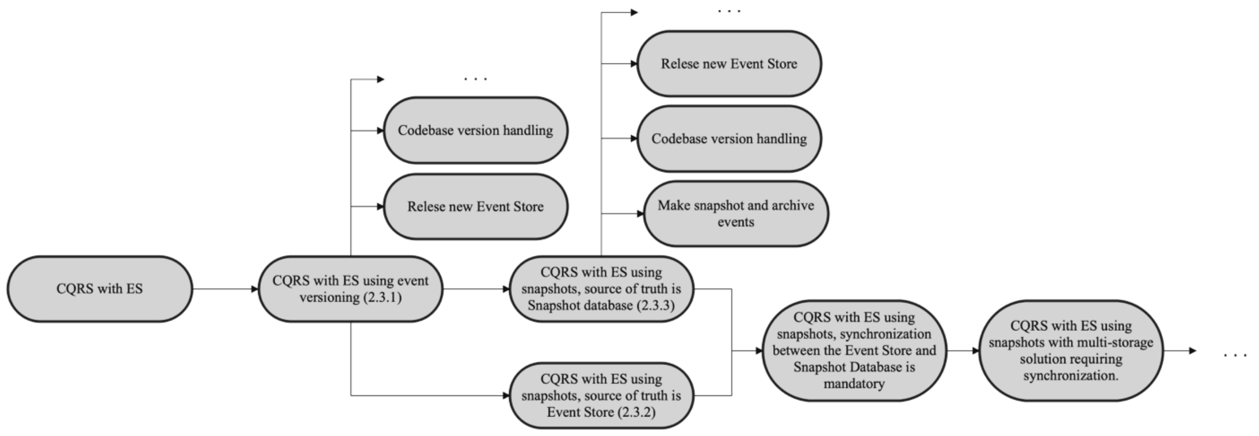
The performance of this method is lower than that of the previous ones, as each modification request requires not only storing the event in the Event Store but also updating the snapshot.

2.3.4. Evolution Roadmap

Apart from the variations discussed in Sections 2.3.2 and 2.3.3, numerous other variations exist for the CQRS with ES architecture. For example, each of the two options can evolve into variations where a new event store is released with every change in event type versions. Variations with event archiving after snapshot creation and their retention in the event store. Other variations include mandatory synchronization between the relational database and the event store, as well as multi-storage solutions requiring synchronization (for instance, using MongoDB to store snapshots for aggregate reconstruction and SQL for storing snapshots for projections and analytics).

All these variations and relations between them form a set of evolutionary branches – an evolutionary roadmap. An example of an evolution roadmap for the CQRS with ES architecture is shown in Fig. 3.

Intuitively, the key approaches are 2.3.2 and 2.3.3, as they address most of the issues inherent in the baseline approach (2.3.1). Many other approaches can be achieved as an evolution from either 2.3.2 or 2.3.3.



. . .
Relese new Event Store
Codebase version handling
Make snapshot and archive events
. . .
Codebase version handling
Relese new Event Store
CQRS with ES
CQRS with ES using event versioning (2.3.1)
CQRS with ES using snapshots, source of truth is Snapshot database (2.3.3)
CQRS with ES using snapshots, synchronization between the Event Store and Snapshot Database is mandatory
CQRS with ES using snapshots with multi-storage solution requiring synchronization.
. . .
CQRS with ES using snapshots, source of truth is Event Store (2.3.2)

Fig. 3. Evolution roadmap for CQRS with ES architecture

The roadmap is presented as a graph, where the nodes represent various variations of the CQRS with ES architecture approaches, and the edges represent the complexity of transitioning from one approach to another. A potential solution for calculating the complexity of such transitions is discussed later in the article.

The roadmap is developed during the planning phase, which may occur before each iteration. It helps visualize possible paths for building and evolving the software application architecture, estimate the effort needed for reaching a specific evolution stage, as well as assess the risks associated with choosing a particular evolutionary branch.

2.4. Formal modelling of the CQRS with ES architecture

2.4.1. State-driven formal description using FSM and Petri Nets

While informal modeling is intended for conceptual understanding of the specifics of CQRS with ES architecture and its variations, formal modeling provides the basis for precise analysis and estimation of the solutions.

Firstly, the comparison of different variations is based on formal descriptions of their processes, which requires a theoretical background allowing a comprehensive description of the applications under consideration.

Secondly, during the design phase of a software application development, such quantitative values can be derived from collected statistics from past projects. However, considering the uniqueness of each application, statistics may not provide accurate results. And it is better if such statistics are collected using a formal process description, which can provide a unification mechanism for different applications.

Therefore, it is necessary to explore the possibility of constructing a formal model of the software application to enable theoretical parameter calculations.

However, there are no publicly available options for the formal modeling of CQRS with ES applications. But, considering the experience in various fields of systems engineering and research, the formal modeling of the CQRS with ES applications processes can be based on variations of FSM [53] and Petri Nets [54] formalisms. Thus, this paragraph describes an attempt to apply standard formal methods to model CQRS with ES architecture-based applications, discussing their benefits and flaws.

In the case of CQRS with ES, the process of command or query request handling can be divided into at least two processes (WMS, RMS) interacting using the Notification Subsystem based on the Pub-Sub interaction pattern. The successful scenario of the WMS process is finishing with publishing events to the Notification subsystem. Then the processes of subscribed to the events receive the events from the Notification subsystem. Mainly, the subscribers are the processes of RMS, but there could also be processes responsible for different variants of notifications (Bounded Contexts, clients). It should also be considered that different processes can publish events of the same type (Fig. 4). For example, registering a patient with a hospitalization and adding a new hospitalization to an already existing patient produces the same event as registering a new hospitalization.

Thus, each process responsible for command request handling can be represented as a composition of at least two finite state machines (FSMs) interacted by the publisher-subscriber mediator (Notification service) [55, 56].

As a rule, to represent a reactive computer system, researchers use the hierarchical finite state machines (HFSMs) formalism. The concept of hierarchical nesting for managing complexity was introduced in [57]. Authors proposed an extension of FSMs called state charts, which allowed (though did not mandate) nesting of FSMs within states of other FSMs for the purpose of managing complexity through modularity. Such FSMs are now usually called hierarchical finite state machines (HFSMs) and used as a tool in software modelling, being a standardized part of the Unified Modelling Language (UML) [58].

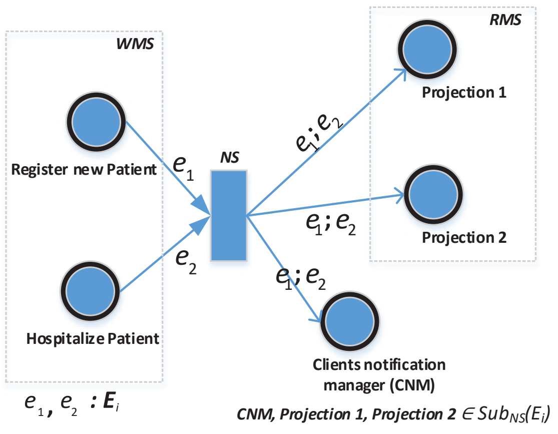


Fig. 4. Conceptual view of the CQRS processes interaction

In the case of CQRS with ES, the process of command handling is not based on a reactive model. It is like a pipeline comprised of different stages, each of which can be represented by different components connected to resources (services). From that point of view, HFSM formalism cannot be applied directly.

The most applicable variant is to use a recursive state machine (RSM) formalism, which is oriented to model control flow in typical sequential imperative processes with recursive procedure calls [59]. According to [60], RSM can be viewed as a variant of visual notations for HFSM, where concurrency is disallowed but recursion is allowed.

A recursive state machine consists of a set of component machines. Each component has a set of nodes (atomic states) and boxes (each of which is mapped to a component), a well-defined interface consisting of entry and exit nodes, and edges connecting nodes/boxes. An edge entering a box models the invocation of the component associated with the box, and an edge leaving a box corresponds to a return from that component.

Intuitively think of component state machines as procedures, and an edge entering a box at a given entry as invoking the procedure associated with the box with given argument values. Entry nodes are analogous to arguments, while exit nodes model return. An edge cannot start from a call or an exit state and cannot end at a return or an entry state.

Based on [60–62], the following definitions can be provided.

Definition. Recursive state machine (RSM) A over a finite alphabet Σ is given by a tuple $\langle A_1,\ldots,A_k\rangle$, where each component state machine A_i consists of the following pieces $(N_i, B_i, Y_i, \delta_i)$:

– finite set of boxes B_i;

– mapping $Y_i : B_i \mapsto \{1,\ldots,k\}$;

– finite set of nodes $N_i = In_i \cup En_i \cup Ex_i \cup Call_i \cup Ret_i$ partitioned into: internal nodes In_i, entry nodes En_i, exit nodes Ex_i, call nodes $Call_i = \{\langle b,e\rangle | b \in B_i, e \in En_{Y_i(b)}\}$, return nodes $Ret_i = \{\langle b,x\rangle | b \in B_i, e \in Ex_{Y_i(b)}\}$;

– transition relation $\delta_i = (In_i \cup En_i \cup \mathrm{Ret}_i) \times (In_i \cup Ex_i \cup Call_i)$.

There can also be defined a weight function $\omega_i : \delta_i \to D$, where $\langle D, \oplus, \otimes, \overline{0}, \overline{1}\rangle$ is a semiring [61], e. g., non-negative real numbers. $\oplus, \otimes$ – addition and multiplication operations, respectively, such that $\overline{0} \oplus a = a \oplus \overline{0} = a$, $\overline{1} \otimes a = a \otimes \overline{1} = a$ etc. As a rule, this function is used in weighted finite-state machines in which edges have weights. In the context of this study, the weight can be used to define the complexity of the transition.

Definition. A stack is a sequence of boxes $S = b_1,\ldots b_r$ where the first box denotes the top of the stack. ε is the empty stack. The height of S is $|S| = r$ – the number of boxes. bS denotes a push operation, which means a push of b onto the stack S.

Definition. A configuration of an RSM $\mathcal{R}$ is a tuple $\langle u,S\rangle$ where $u \in In \cup En \cup Ret$ is an internal, entry, or return node, and S is a stack. $u \in N_{Y_{j1}(b_1)}$, which means the case where the control is inside the module of node u, which was entered via box b_1 from module A_{j1}, which was entered via box b_2 from a module A_{j2} and so on.

Definition. Arrow $\Rightarrow$ denotes a transition relation over configurations and a weight function $:\Rightarrow\mapsto D$, such that $\langle u,S\rangle\Rightarrow\langle u',S'\rangle$ with $\omega(\langle u,S\rangle,\langle u',S'\rangle)=v$ if and only if there exists a transition $t\in\delta_i$ in $\mathcal{R}$ with $\omega_i(t)=v$ and one of the following holds:

– *Internal transition*: $u'\in In_i, t=\langle u,u'\rangle$, and $S=S'$;

– Call transition: $u'=e\in En_{Y_i(b)}$ for some box $b\in B_i$, $t=\langle u,\langle b,e\rangle\rangle$ and $S'=bS$;

– Return transition: $u'=\langle b,x\rangle\in R_i$ for some box $b\in B_i$ and exit node $x\in Ex_{Y_i(b)}$, $t=\langle u,x\rangle$, and $S=bS'$.

It's assumed that Call immediately enters the called module and a Return immediately returns to the calling module, i. e., the transition expenses in terms of time and resources are not suggested.

Definition. A computation of an RSM $\mathcal{R}$ is a sequence of configurations $\pi=c_1,\ldots,c_n$, such that $c_i\Rightarrow^* c_{i+1}$ for every $1\le i<n$. π is a computation from c_1 to c_n, which is denoted as $\pi=c_1\Rightarrow^* c_n$.

A computation $\pi=c_1\Rightarrow c_n$ is called non-decreasing if the stack height of every configuration of π is at least as large as that of c_1.

The weight of computation, according to [61], is defined as $\otimes(\pi)=$ $=\otimes_{i=1}^{n-1}\omega(c_i,c_{i+1})$.

For a set of computations Π, the weight is defined as $\oplus(\Pi)=$ $=\oplus_{\pi\in\Pi}\otimes(\pi)$.

A computation that cannot be extended by any transition is called a halting computation.

For example, there is an *RSM*, which consists of three component machines A_1, A_2, Ex (Fig. 5). Intuitively, this machine can be thought as a workflow management system (WFMS) logic that accepts a request σ by the handler represented by A_1 component-based machine. The request can be valid or invalid $\sigma\in\{\sigma_+,\sigma_-\}$, the validation logic is connected to a node e_1. In result of actions connected to the entry e_1 the transition edge is selected: transition to entry v of A_2 machines, which denotes a service in case of a valid request σ_+; transition to entry ex of Ex machine is responsible for handling an exception in case of an invalid request $\sigma\_$. A service represented by the A_2 machine reacts to correct data passed from e_1 node of A_1 machine by passing the result of processing to one of two exit nodes: r – in the case of a success scenario, ex – in the case of an exception. After the finishing of request handling by the A_2 – the result is passed from r exit node of A_2 to the t exit node of A_1 or from ex exit node of A_2 to ex entry node of the Ex machine. The result of Ex machine is passed from z exit node of Ex machine to the terminal node t of A_1.

The success scenario of valid request handling can be described as follows

$$\langle e_1,\varepsilon\rangle\Rightarrow\langle v,b_1\rangle\Rightarrow\langle\langle b_1,r\rangle,\varepsilon\rangle\Rightarrow\langle t,\varepsilon\rangle.$$

In case of exception occurs while handling the request in A_2, the flow is

$$\langle e_1,\varepsilon\rangle\Rightarrow\langle v,b_1\rangle\Rightarrow\langle\langle b_2,ex\rangle,\varepsilon\rangle\Rightarrow\langle ex,b_2\rangle\Rightarrow\langle\langle b_2,z\rangle,\varepsilon\rangle\Rightarrow\langle t,\varepsilon\rangle.$$

Firstly, it is notable that the specifics of handling the input request or intermediate data that appeared during transition is hidden within the nodes which correlates to HFSM representation with OnEntry and OnExit logic connected to each state. Thus, the logic and its complexity, response/request, resources needed to realize a step of computation is not declared explicitly.

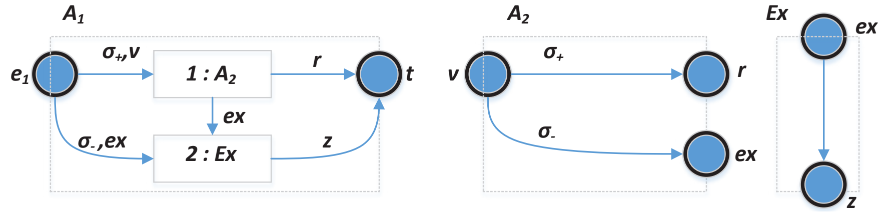


Fig. 5. An example of a simple RSM model of WFMS flow

Secondly, the machine accepting the input request could be used to describe a service composed of several handlers instead of only one. In this case, A_1 machine will have a set of entry nodes and a set of exit nodes, respectively, each entry node per request type.

Thirdly, RSM is formalism to model serial computation, which can be used to describe specific data processing by the WMS, NS, and RMS, but cannot show their integration based on the pub-sub pattern.

In paper [63] authors define the process as a scalable state machine (SSM) and use Petri Nets for several SSMs integration. SSM formalism is similar to recursive state machines shifting focus in case of handling request logic from OnEntry, OnExit nodes to arcs-activities enabling their representation by another FSM. A significant feature of the approach is that the description is based on the semantic equivalence of two different state machines seen at different levels of detail, which allows their comparison. Formally, SSM is defined as follows.

Definition. A scalable state machine (SSM) Σ is a finite state machine (S,s_{src},S_{dst},A,T):

– S – the set of all states;

– $s_{src}\in S$ – the source state, or entry state;

– $S_{dst}\subseteq S$ – the set of destination states, or exit states;

– A – the set of activities;

– $T: S\times A\rightarrow S$ – the state-transition function.

According to [63], the activity is defined as a tuple (role, action), where the role identifies a user or software agent responsible for realizing the action (i. e., execution of a task) which produces a deliverable, such as a document or a piece of software.

The action of the WMS state machine produces an output in the form of object, aggregate, and/or events using some resources such as services, repositories, databases etc. Considering this specific the definition of activity can be represented as follows.

Unfortunately, the authors provided a simple variant of machines integration which cannot be applied to model the integration of CQRS modules. The definition of the activity as a tuple (role, action) and component machine as a machine with exactly one source state and one destination state also could not be used directly to model CQRS based system.

Let's provide basic definitions:

Definition. An activity $\alpha\in A$ is a tuple defined as (resources, function), where the resources labels a set of resources used to realize the function (i. e., execution of a task), producing an intermediate or final result in a form of an object, aggregate of entities or event, or their composition.

Definition. A component scalable state machine CSSM is a scalable state machine with exactly one source state $s_{src}\in S$ and exactly one destination state $s_{dst}\in S$.

Definition. A SSM $\Sigma(S,s_{src},S_{dst},A,T)$ is semantically equivalent to another scalable state machine Σ where the transition $t=(s^t_{src},\alpha^t,s^t_{dst})\in T$ has been replaced by a CSSM $K=(S^K,s^K_{src},s^K_{dst},A^K,T^K)$ if and only if the following conditions hold:

– Source and destination states, respectively, are identical for the CSSM K and the transition t it replaces $s^K_{src}=s^t_{src}$, $s^K_{dst}=s^t_{dst}$.

– A meaningful overall resources r^K and a meaningful overall function a^K of the CSSM K that corresponds respectively to the resources r^t and action a^t in the activity $\alpha=(r^t,a^t)$ can be defined.

This definition of SSMs' equivalency is very important for comparing different variations of CQRS. It allows to define an abstract CQRS system module (WMS, RMS) with a certain number of common states which can be applied to all the variations and substitute the transitions between the states by the CSSMs. And comparing the variations of CQRS realizations in that case will be reduced to comparing the CSSMs.

It is also important to explicitly define the notion of a deliverable produced as an action result. It can be seen that the result, as well as the intermediate request that appeared when the transition occurred, are not explicitly described in RSM and SSM models. In the RSM case that requests and results are the elements of alphabet Σ, but the "request – response" transformations are not defined.

Thus, the transition from s to s' can be defined as a relation (s,x,y,a,s'), considering that activity is a composition of action, which is focused on what should be done and resources focused on how it should be done. In this work the following notation is used $s \xrightarrow[a]{x/y} s'$, to represent the relation (s,x,y,a,s'), where x – the request and y – the response-result of transition.

In such a way, the transition can be intuitively accepted as an interface (s,x,y,s') which allows to use different realizations of the activity a, i. e., its realization by the different CSSMs.

It also correlates with Petri Nets (place/transition nets or P/T nets), which are well-suited to represent complex discrete event processes. Formally, a Petri Net is defined as a triple (P,T,F) [64]:

– P – a finite set of places, containers (denoted by ○), which contain the tokens (denoted by ●);
– T – a finite set of transitions-actions ($P \cap T = \varnothing$), denoted by ▮;
– $F \subseteq (P \times T) \cup (T \times P)$ – a set of arcs (flow relation).

If a sufficient number of tokens are contained in place-container, an action is triggered. After firing an action, the tokens that helped to fire this action are removed, and some new tokens are generated. Thus, a place can be intuitively thought of as a container or a multi-set allowing multiple copies of the same value or a queue.

A place p is called an input place of a transition t if there exists a directed arc from p to t. Place p is called an output place of transition t if there exists a directed arc from t to p. Formally, for any node

$$x \in P \cup T, \bullet x = \{y | (y,x) \in F\} \text{ and } x \bullet = \{y | (x,y) \in F\}.$$

The state S (also known as marking) of the net is the distribution of tokens over places, i. e., $S : P \to \mathbb{N}$. Formally, a marking S, usually denoted by vector $(\mathbb{N})^{|P|}$. For example, the state with one token in place p_1 two tokens in p_2, one token in p_3, and no tokens in p_4 can be represented by a vector 1,2,1,0 or the following poset: $\langle (p_1,1),(p_2,2),(p_3,1),(p_4,0) \rangle$.

In case of CQRS, this formalism is very useful to describe the asynchronous processing by the module comprised of autonomous stages connected with queues, the number of tokens within the queue is the number of messages-objects to process. It can be used to assess the load on the module.

The module can be interpreted as a system function connected to a certain use case or a generic pipeline that is purposed to handle requests of several types.

The comparison of states in accordance with [64] is defined using partial ordering. For any two states S_1 and S_2, $S_1 \le S_2$ if for all $p \in P : S_1(p) \le S_2(p)$.

$S_1 \xrightarrow{t} S_2$ denotes the situation when the module goes from S_1 to S_2 after transition t occurred.

A state S_n is called reachable from S_1 (notation $S_1 \rightarrow^* S_n$) if there is a firing sequence $\sigma = t_1 t_2 t_3 \ldots t_{n-1}$ such that $S_1 \xrightarrow{\sigma} S_n$, where the firing of a transition removes tokens from its input place and adds tokens to its output place.

The classical Petri net is limited to describing complex systems, e. g., they do not allow for the modeling of data. To solve these problems, some extensions such as Colored Petri net to model data have been proposed. For example: the extension with color to model data, and the extension with hierarchy to structure large mode.

Colored Petri nets (CPNs) [65] is an extension of the classical model with typed or colored tokens. Thus, each place is described by two required attributes: its name and type. According to [66], CPNs are ordinary Petri Nets with the strengths of a high-level functional programming language called CPN ML. Tokens can be coded as data values of a rich set of types (called color sets), and arc inscriptions can be computed expressions.

To handle the complexity of a network, Hierarchical Petri Nets with a subnet construct can be used [67, 68]. A subnet is an aggregate of several places, transitions, and subsystems. At first level it gives a simple description of the process specifying more detailed behavior at other levels like HFSM.

Definition. A P/T net is called a workflow net (WF-net) if [64, 69]:

– there is one source place $in \in P$ and one sink place $out \in P$, $\bullet\, in = out \bullet = \varnothing$;
– if a transition t^* is added to P/T net which connects place out with in (i. e., $\bullet t^* = \{out\}$ and $t^* \bullet = \{in\}$), then the resulting Petri net is strongly connected, which means for every pair of nodes (i. e., places and transitions) x and y, there is a path leading from x to y.

The extension with color can be used to model workflow attributes, which provide a specific piece of information used for the routing of a case. The examples of workflow attributes are the age of the complainant, the department responsible for the complaint, or the registration date.

Petri Nets is an activity-centered formalism [70, 71], but it is primarily focused on flow analysis without analyzing transition specific: the resources used, the complexity of the transition, etc.

From that point of view, each transition can be represented as an RSM that accepts a command from the place-queue and posts the result to the output place-queue. Thus, it can be considered a pipeline comprised of RSM or SSM machines.

However, it cannot be used to describe the integration of modules (WMS, RMS) based on the publisher-subscriber pattern. There are several works devoted to describing the pub-sub pattern and its specifics. For example, in [72], authors describe the use of the Symmetric Petri Net Model of Generic Publish-Subscribe Systems applied to business process conformance checking, in [73], authors propose to use Predicate/Transition nets (PrT nets) [74] more generalized version of Petri Nets for analyzing Publisher Subscribe Architecture. The [75] describes a solution to causal event processing issues that arise in CQRS architecture applications.

For this study, it is not important how exactly the interaction based on the publisher-subscriber pattern would be realized, because the problems are not connected with its structure.

Thus, taking into account different facets of the CQRS architecture described by RSM, SSM and Petri Nets models, it can be concluded that the CQRS system as a composition of WMS and RMS subsystems interacting via Pub-Sub NS. Each subsystem comprises several RSM-SSM based modules-stages the compositions of which can be seen as transitions in terms of Petri Nets. These transitions are connected with message queues (i. e., Petri Net places).

2.4.2. Knowledge representation model

Modeling the application using different formalisms is effective because it provides a way to look at the different aspects of the application from different angles. But it is not convenient for the complexity analysis of the development or modification process this work is devoted to. Thus, an activity-centric CQRS application development process knowledge representation without losing connection to the models mentioned above is provided. The main idea is that the structure and complexity of the development process correlate with the structure and complexity of the function realization, i. e., the process of handling a request, which consists of several phases connected to different components developed by the development process.

The class of processes is connected to a certain class of use cases. At a high level of abstraction, all the use cases, and consequently the processes responsible for their handling, can be divided into two basic categories: write-oriented (including support-oriented use cases such as projection rebuilt), read-oriented use cases. These categories may also contain sub-categories, which can be divided into classes, etc.

The basic categories can be described by the two knowledge representation models, forming two basic architectural patterns (in the scope of CQRS with ES). Both categories of processes are comprised of activities and may result in exceptions, but write-oriented processes, in the case of successful handling, end processing with events publishing (i. e., the result of the command handling is a set of notifications), while the read-oriented processes results are responses represented by data transfer objects without events generation and publishing.

This statement can be formally represented as follows

$$U_W, P_W, U_W \subseteq U, P_W \in P, \tag{1}$$

where U – the whole set of use cases; P – the whole set of processes; U_W – the set of write-oriented use cases; P_W – the set of write-oriented processes.

At the high abstraction level, CQRS write-oriented function can be described as a process triggered by the request, which as a result provides a set of published events, while read-oriented function results in a conventional response without publishing any events, both function types can lead to exceptions.

By analogy with RSM [60], a process can be represented as a sequence of multiple activities. Each activity can represent either an atomic (node) or a complex functional component (box), accompanied by a context consisting of input, output, and auxiliary data. The characteristic feature of such processes is their similarity: that is, a generalized process frame [76] can be represented, consisting of a sequence of typical activities-slots. In general form, the set of write-oriented processes can be formally written as follows

$$P_W = \left(\left(A_W, <_A\right), Rq_W, IE_W, Ex_W\right), \tag{2}$$

where $A_W \subseteq A_{op}$ – a set of activities of write-oriented operations, while A_{op} is the total set of activities; $Rq_W \subseteq Rq_{op}$ – a set of write-oriented requests of operations; $IE_W \subseteq IE_{op}$ – a set of write-oriented integration events of operations; $Ex_W \subseteq Ex_{op}$ – a set of write-oriented exceptions of operations.

$$P_R = \left(\left(A_R, <_A\right), Rq_R, Rp_R, Ex_R\right),$$

where $A_R \subseteq A_{op}$ – a set of activities of read-oriented operations, while A_{op} – total set of activities; $Rq_R \subseteq Rq_{op}$ – a set of read-oriented requests of operations; $Rp_R \subseteq Rp_{op}$ – a set of read-oriented responses of operations; $Ex_R \subseteq Ex_{op}$ – a set of read-oriented exceptions of operations.

Each process from the subset P_{op} can be represented as an instance of this frame. Thus, for any process $p_i \in P_{op}$, the ordered set of activities $(A_i, <_A)$ will be a subset of $(A_{op}, <_A)$, such that $A_{op} = \langle A_1, A_2, \ldots, A_n \rangle$ and $p_i.A_{op} = \langle a_1, a_2, \ldots, a_n \rangle$, where the notation $p_i.A_{op}$ represents the poset of activities of the i-th process, $a_1 \in A_1, a_2 \in A_2, \ldots, a_n \in A_n$. If $A_1, A_2, \ldots, A_n$ are tuples, and the index of an element in each corresponds to the index of the process, then it holds that

$$p_i.A_{op} = \langle \pi_i(A_1), \pi_i(A_2), \ldots, \pi_i(A_n) \rangle.$$

This definition resembles the concept of a relation schema in Codd's relational algebra but differs in that the order of activities is important, and the type of components may be complex. It also resembles the logic of processes, described as a path of states in [77]. But, in our opinion, the logic of processes is more general and does not allow for a sufficiently complete description of the architecture under consideration.

Based on formulas (1) and (2) a description of a specific process p_i corresponding to the system function u_i can be represented as:

$$\langle u_i, p_i \rangle \in \langle U_{op}, P_{op} \rangle,\ u_i \in U_{op},\ p_i \in P,$$

$$p_i = \left(\left(p_i.A_{op}, <_A\right), Rq_i, IE_i, Ex_i\right),$$

$$p_i.A_{op} \subseteq A_{op},\ IE_i \subseteq IE_{op},\ Rq_i \subseteq Rq_{op},\ Ex_i \subseteq Ex_{op}.$$

In this case, activities are a broader concept than in the description using scalable state machines [63]. An activity is a container that includes the description of a software function, as well as various core and auxiliary resources, components, and characteristics associated with it. From this perspective, activities resemble pipeline stages, and this allows for the adaptation of the process to an asynchronous request processing model by establishing queues between stages and refining the response return mechanism.

It is important to note that the function of an activity is partial, and its execution may be associated with side effects, specifically exceptions. Therefore, the minimal description of an activity should include: the function, types of input data, types of output data (including events), and the set of possible exceptions. A formal representation of an activity ($a_i \in A$) can be its description as a relation (tuple), which represents the relation of components of different types. It should also be noted that upon completion of an activity, the process is in one of the states, which corresponds to the process description using Process Logic [77].

In type theory and relational algebra, each tuple has a product type that fixes the underlying types of each component (called domains in relational algebra)

$$a = \left(c_1, c_2, \ldots, c_n\right) : C_1 \times C_2 \times \ldots \times C_n. \tag{3}$$

Otherwise, a tuple can be represented as an ordered set of projections onto domain-types, which can be represented as

$$a = \left(\pi_1(a) : C_1, \pi_2(a) : C_2, \ldots, \pi_n(a) : C_n\right).$$

Thus, the activities of the i-th process $(p_i.A_{op}, <_A)$ are represented as a poset of tuples that has a corresponding structure, which aligns with the concept of a relation in relational algebra. This poset can easily be represented using Petri nets [64] or as a path (directed acyclic graph), as shown in [77], but with vertices being tuples that describe the structural and functional characteristics of the activity. The case of representing a set of activities as a cyclic, branched, directed graph is not considered in this work, as it does not correspond to the architecture under consideration.

It is important to note that the structure of the tuple, i. e., the types of tuple components, will be common for the activities of all processes. That is, the structure A_{op} is represented as a frame-schema, with each of its slots linked to a specific type of component. Formally, this can be written as

$$\forall a_j \in p_i.A_{op}.a_j = \left(c_1, c_2, \ldots c_k\right) : C_1 \times C_2 \times \ldots \times C_k,$$

where $C_1, C_2, \ldots, C_k \subset \mathcal{A}x$ – the types of slot-components describing the activity, and $\mathcal{A}x$ – the set of all possible component types. The types of slots can be represented as sets of all possible values associated with this slot (similar to formula (3)).

Slots may vary, and the structure of the frame describing the activity can change. However, the root (mandatory) slots for the software application are the slots for input, output data, and exceptions, which describe the contract, and the function slots associated with the transformation algorithm.

Here, it is important to emphasize again the distinction between a function and an activity. An activity serves to describe the features of the transformation contract, but it does not describe the transformation mechanism itself. It can be compared to a process descriptor, a frame that describes the knowledge about the transformation, the necessary resources for the transformation to occur, but does not describe the algorithm of the transformation, which is the prerogative of the function. Some components of the activity may describe architectural features of the function, such as the layer, class, or method in which the function resides, i. e., having no direct relation to the arguments or results of the function.

Let's define the set of operations for working with processes and activities within the framework of the considered model:

– *creating a process*: Defining its input, output, and auxiliary parameters;
– *modifying a process*: Changing its input, output, or auxiliary parameters;
– *adding an activity*: Inserting a new activity into the set of activities that describe the process;
– *modifying an activity*: Modifying the parameters of a specific activity;
– *removing an activity*: Deleting an activity from the set of activities that describe the process;
– *decomposing activities into sub-activities*: Similar to "unfolding boxes" in RSM and using CSM as part of SSM.

When performing all of these operations, the activities sequence consistency rule must be followed. Based on the definition of an activity, a tuple can contain many optional parameters, but it must always contain a descriptor of the function (f_{ij}), as well as the input and output data (In_{ij}, Out_{ij}). By analogy with transition replacement conditions from SSM to CSM [63], the consistency rule intuitively states that for activities to be linked sequentially, the type of outgoing data of the preceding activity must match the type of incoming data of the subsequent activity.

Mathematically, this can be described as the existence of a mandatory meet between adjacent activities, $a_{ij} \cap a_{ij+1}$, which occurs if the output type of the preceding function Out_{ij} (the codomain of the function f_{ij}) matches the input type of the following function In_{ij} (the domain of the function f_{ij+1}). This can be formally written as follows

$$S\left(a_{ij}\right) \cap S(a_{ij+1}), \text{ if } Out_{ij} : T_k,\ In_{ij+1} : T_k,$$

where $S\left(a\right)$ denotes the set of values in an ordered tuple.

Alternatively, if the activities are adjacent, and the argument of the subsequent activity and the result of the preceding activity are represented by the types T_k and T_l, then these data types must be equal

$$\text{if } Out_{ij} : T_l,\ In_{ij+1} : T_k \wedge S(a_{ij}) \cap S(a_{ij+1}), \text{ then } T_l = T_k.$$

This set of operations and the activities sequence consistency rule allows to create and manipulate processes within a software application.

Now, let's confine the types of processes appropriate to the scope of this article. Since the systems under consideration do not provide a generative response form, only processes with the following scenarios are considered:

– basic-successful scenario, which is executed when the final activity, which is the supremum of the activities poset, returns as a result a tuple (rp_i, IE_i) consisting of a response and, in the case of a command pattern (P_c), a set of integration events;
– alternative scenario, which leads to the generation of an exception during the execution phase in one of the activities in the sequence A_i. In this case, not all activities will participate in the processing. While this scenario cannot be ignored when describing software application processes, it does not affect the calculation of process parameters in this article

$$p_i\left(rq_i\right) = \begin{cases} p_i^+\left(rq_i\right), \text{ if } \sup\left(\pi_f\left(A_i\right)\right) = \left(rq_i, IE_i\right), \\ ex_{ip}, \text{ if } \exists f_{ik} \in \pi_f\left(A_i\right).\ f_{ik} : In_{ik} \rightarrow ex_{ip} \in Ex_i. \end{cases}$$

It should be noted that the basic-successful scenario of process execution represents a composition of activities (functions with associated components), which, according to [77], corresponds to the operation of concatenation. This composition is applied to process the request and generate a result in the form of a response and a set of events

$$p_i^+\left(rq_i\right) = \left(a_{im} \circ a_{im-1} \circ \ldots \circ a_{i1}\right)\left(rq_i\right) = \left(rq_i, IE_i\right).$$

However, while representing activities as a *poset of tuples* provides a formal description of the process, it does not allow for a clear evaluation of process parameters. To analyze specific parameters of the process, such as performance or complexity, let's identify the components of activities.

To extract the components, the projection operation from relational algebra can be used on a specific type of tuple components associated with the activity. Consider the following example:

– Let S be defined as a poset of tuples

$$S = \left\langle \left(x_1, y_1\right), \left(x_2, y_2\right), \ldots, \left(x_n, y_n\right) \right\rangle, x_i \in X,\ y_i \in Y, n \in \mathbb{Z}.$$

– Then, the projection on type X of S is expressed as

$$\pi_X\left(S\right) = \left\langle x_1, x_2, \ldots x_n \right\rangle, x_i \in X, n \in \mathbb{Z}.$$

Using a modified join operation by the ordinal key of elements in the sequence, the projections can be merged, yielding the following equality

$$\left\langle x_1, x_2, \ldots x_n \right\rangle \bowtie \left\langle y_1, y_2, \ldots y_n \right\rangle = \left\langle \left(x_1, y_1\right), \left(x_2, y_2\right), \ldots \left(x_n, y_n\right) \right\rangle = \\ = S, x_i \in X,\ y_i \in Y, n \in \mathbb{Z}.$$

Thus, the ordered set of relations describing the process can be represented as the join of projections of relations using the modified join operation

$$p_i.A_{op} = \pi_{x_1}\left(A_i\right) \bowtie \pi_{x_2}\left(A_i\right) \bowtie \ldots \bowtie \pi_{x_n}\left(A_i\right).$$

To evaluate any parameter of a process related to a domain $ax_j \in \mathcal{A}x$, an evaluation function φ_{ax_j} can be used. The evaluation function takes a set of qualitative assessments of a parameter for each process activity and returns a quantitative assessment of this parameter for the entire process, which can be described as

$$\varphi_{ax_j} : \cup S\left(\pi_{ax_j}\left(p_i.A_{op}\right)\right) \rightarrow \mathbb{R},\ ax_j \in \mathcal{A}x, \quad (4)$$

where $\cup S(\pi_{ax_j}(p_i.A_{op}))$ – represents the union of subsets formed by the elements of the projection.

It is assumed that the subsets, which can be elements of the set formed from the projection, contain elements belonging only to a single domain. For example, for the activities of process $A_i = a_{i1}, a_{ik}$, the projection $\pi_R(A_i) = \{r_a, r_b, r_c\}, \{r_a, r_b\}$ and $\cup S(\pi_R(A_i)) = \{r_a, r_b\}$. Thus, the evaluation function $\varphi_R(\cup S(\pi_R(A_i))) = \varphi_R(\{r_a, r_b\})$.

Thus, the formal model for describing a typical process of an architectural approach provides an opportunity to obtain an objective formal representation of knowledge about software application processes and enables the calculation of various process parameters at the design stage of software application development. These parameters can be used as input data for MCDA methods and subsequent decision-making on the selection of an architectural strategy. Since activity parameters can represent not only calculated values for individual implementations but also statistical values derived from corresponding activities across multiple typical software applications, this ensures greater objectivity in the comparison.

Another application of formula (4) is the calculation of the distance between variations of CQRS with ES architectures (Fig. 3). In this context, distance refers to the similarity between two variations, the complexity of transitioning from the source approach to the target one. Each variation can be represented as a set of processes. The distance from one variation to another is equal to the sum of the transition complexities of all processes within the software application.

Distance assessment comes down to comparing the activities of the compared processes. Some of the activities can be considered of the same type, they could be called matched activities. For activities that don't match (i.e., those that need to be added or removed), the complexity of addition/removal is calculated. Matched activities can either be identical or require modification. Identical activities are excluded from the complexity assessment. To evaluate the complexity of modifying matched activities, they are further divided into sub-activities until all activities are identical, intended to be removed or added.

Thus, similarly to [78], the process transition complexity (ω_{p_i}) could be calculated as the sum of the complexities of adding new activities and removing activities that are no longer needed in the new architectural variation of the system. The qualitative parameter used to assess the complexity of adding or removing activities is a certain representation of the activity's algorithm π_{Alg} (e. g., in the form of pseudocode, flowcharts, etc.)

$$\omega_{p_i} = \varphi_{adding}\left(\cup S\left(\pi_{Alg}\left(A_{new}\right)\right)\right) + \varphi_{removing}\left(\cup S\left(\pi_{Alg}\left(A_{remove}\right)\right)\right),$$
$$A_{new} \subset p_i.A_{op},\ A_{remove} \subset p_i.A_{op},\ Alg \in \mathcal{A}x, \tag{5}$$

where φ_{adding} – an evaluation function that calculates the complexity of adding activities; $\varphi_{removing}$ – an evaluation function that calculates the complexity of removing activities; A_{new} – a set of activities that should be added to the target process; A_{remove} – a set of activities that should be removed from the target process; π_{Alg} – a collection of activity algorithms used as the basis for calculating the operation's complexity.

The total transition complexity (ω) is defined as the cumulative sum of individual process transition complexities. While the complexity associated with migrating multiple processes of the same type may decrease over time as the developer gains experience, this reduction typically converges toward a saturation threshold. The saturation-level complexity, which reflects the steady-state efficiency of a skilled developer, is generally lower than the upper-bound complexity values used in this study. However, given the presence of numerous additional risk factors and uncertainties not captured in this complexity model, the upper-bound estimates are used to ensure robust planning

$$\omega = \sum_i c_{p_i} \cdot \omega_{p_i}, \tag{6}$$

where c_{p_i} – the number of processes of a certain type in the considered software application.

3. Results and Discussion

3.1. System Parameters for Experimentation

This section aimed is to demonstrate the application of the proposed approach. As an example, an RTP for which the CQRS with ES architecture has been chosen can be considered. The process of determining the applicability of the *classical CQRS* (alias for option 2.3.2) and *mCQRS* (alias for option 2.3.3) variations to the RTP under different conditions is described. These variations were chosen because:

1) they both address most of the issues inherent in the baseline approach (2.3.1);

2) they have different priorities (one maximizes software application performance, while the other minimizes development complexity to ensure the fastest possible implementation), making the assessment more illustrative.

Additionally, an example of calculating the distances between the examined variations for the evolution roadmap of the RTP is provided.

Thus, the experiment consists of the following stages:
– defining RTP parameters;
– formal description of typical processes for the proposed variations;
– evaluation of parameters for typical processes of the considered variations;
– application of the MCDA method to an application under various conditions, using measured process parameters;
– calculation of distances and creating the software application evolution roadmap;
– application parameters for experimentation.

Let's determine the parameters of the application on which the experiment is conducted. These parameters include the number of processes required before the first release, as well as the ratio of new process additions to modifications of existing processes during subsequent stages of development. These parameters are defined as fixed values based on statistical data collected from real projects (e. g., DBB Software projects). Based on the statistics, the MVP version of the project includes 41 command processes and 19 query processes on average. After the release of the MVP version, it is assumed that tasks for modifying existing functionality and adding new features are evenly distributed.

3.2. Formal description of typical processes

This section focuses on the preparation of source data for parameters calculation. It includes the description of algorithms for applications based on each variation and the creation of RTPs for basic architectural variations. Algorithms are used to compute complexity. RTPs enable the measurement of performance metrics.

The highest level of abstraction for describing functionality at the system level is a use case. Use case describes the basic and alternative scenarios of actor-system interactions connected to a certain system function [79]. The use cases in accordance with the CQRS with ES approach can be divided into two basic types: read-oriented queries and write-oriented commands. The write and read-oriented requests are handled by the processes which can be described by the models using the proposed modelling approach for each architectural variation, considering the most complex (pessimistic) variant of the task (i. e., the variant of the process is intended to cover all the variants, involves all possible activities). It is worth noting that while the activities of the compared processes realize the same contracts (i. e., responsible for the same functionality of the process), which provide the basis for their comparison, their realizations could be different. For the processes divided into activities, complexity parameters can be calculated using selected evaluation methods (Section 2.1.4). Given the known number of processes of each type within the application, these parameters are then used to derive the overall characteristics of architectural variation.

Let's describe the two main processes (command processing and query processing) for a software application based on the classical CQRS architecture and mCQRS. This provides a comprehensive view of the software application and allows for an objective operation with knowledge about it.

For the RTP, it is proposed to account for two classes of use cases: command use cases (U_c) and query use cases (U_q). U_c, $U_q \subset U$. These types of use cases correspond to process types P_c, $P_q \subset P$. To formally describe the processes using the mathematical model proposed in Section 2.4.2, the most comprehensive process of each type is decomposed into its constituent activities. Some processes may belong to a certain type but lack the full set of activities (e. g., in some cases, the unnecessary command validation step may be skipped). However, for the formal description of a typical process, the complete set of activities is used. This approach allows to calculate the maximum complexity value for a typical process.

Thus, the process handling a write operation (command) can be formally represented as follows

$$\forall p_i \in P_c,\ P_c \subset P_{CQRS} \mid \left(p_i.A_j, <_A\right) =$$
$$= \Big\langle \left(\text{Create command}\right), \left(\text{Validate command}\right), \left(\text{Route command}\right),$$
$$\left(\text{Fetch aggregate CQRS}\right), \left(\text{Update aggregate's state}\right),$$
$$\left(\text{Save aggregate CQRS}\right), \left(\text{Dispatch events}\right), \left(\text{Route event}\right),$$
$$\left(\text{Handle update projection event CQRS}\right), \left(\text{Notify client}\right)\Big\rangle.$$

Round brackets denote the activity slot. The structure of the activity depends on the application's architectural specific. The information required to perform the evaluations:

- in – input data of the activity;
- out – output data which is produced by the activity;
- description of the activity's functionality (Algorithm).

$$\forall p_i \in P_c, P_c \subset P_{mCQRS} \,|\left(p_i.A_j, <_A\right)=$$
$$=\left\langle\left(\text{Create command}\right),\left(\text{Validate command}\right),\right.$$
$$\left(\text{Route command}\right),\left(\text{Fetch aggregate mCQRS}\right),$$
$$\left(\text{Update aggregate's state}\right),\left(\text{Apply event onto aggregate}\right),$$
$$\left(\text{Save aggregate mCQRS}\right),\left(\text{Dispatch events}\right),\left(\text{Route event}\right),$$
$$\left.\left(\text{Handle update projection event mCQRS}\right),\left(\text{Notify client}\right)\right\rangle.$$

The process of a read operation (query)

$$\forall p_i \in P_q, P_q \subset P \,|\left(p_i.A_j, <_A\right)=$$
$$=\left\langle\left(\text{Create request}\right),\left(\text{Validate query}\right),\left(\text{Fetch projection from DB}\right),\right.$$
$$\left.\left(\text{Map projection to DTO}\right),\left(\text{Return DTO}\right)\right\rangle.$$

3.3. Evaluation of parameters for typical processes

3.3.1. Complexity metrics

The complexity of an activity is the combined complexity of its functionality and the infrastructure that allows its implementation. In this case, the infrastructure for all the compared activities is nearly identical, except for the projection snapshot and aggregate snapshot databases. These resources exist in the classical CQRS approach and are described as parts of a common snapshot database in the mCQRS one. Therefore, the complexity of the infrastructure can be disregarded in the evaluation.

There are various methods for evaluating the complexity of software functional components, and some of them may show that one option is simpler, while others might indicate the opposite. Since the modification presented in this work is intended to simplify the development and maintenance of applications based on CQRS with Event Sourcing architecture, it is appropriate to compare the cognitive complexity of the algorithm, i. e., how difficult it is for a human to comprehend. Both the development complexity of the process and the complexity of its modification should be considered.

To obtain the most objective assessment of complexity for an activity, it should be calculated based on statistics. There are tools available for automatically evaluating the complexity of software code. For a variety of existing typical applications, the complexity of individual components corresponding to the activities in the process description template can be assessed. The more accumulated statistics, the more accurate the evaluation will be. Due to the lack of access to a large number of existing software applications based on the classical CQRS architecture and its mCQRS modification, it is suggested to conduct the evaluation using the RTP and Use Case flowcharts of typical functions.

CQRS with ES based applications typically decompose logic into multiple thin layers, each responsible for a specific concern. While this decomposition facilitates better code organization and maintainability, it also imposes additional cognitive load on developers during implementation. Accordingly, process development complexity in such applications can be decomposed into two distinct components: design complexity, which pertains to the allocation of responsibilities across architectural layers, and realization complexity, which reflects the actual coding effort.

It is important to note that code segments implementing complex business logic are not considered in this evaluation, as they are typically unique to each application. Such segments contribute to the overall application complexity rather than reflecting the complexity introduced by a particular architectural variation.

The algorithm for computing the implementation and modification complexity of CQRS with ES architectural variation processes is presented in Fig. 6.

In this experiment, design complexity is estimated using the Cognitive Functional Complexity method [46], applied to representative use-case flowcharts. This technique quantifies the mental effort required by developers by assigning cognitive weights to Basic Control Structures (parts of the flowchart) within the flowcharts. The resulting complexity is computed using formula (7) and expressed in Cognitive Weight Units (CWU)

$$S_f=\left(N_i+N_o\right)\cdot\sum_{i=1}^{l}W_i, \tag{7}$$

where N_i – number of input arguments; N_o – number of output arguments; W_i – the weight of the i-th basic control structure; l – number of basic control structures in the flowchart.

Attachment 1 in [80] provides a calculation of Cognitive Functional Complexity for all activities (Tables 2–4) using formula (7). The complexity parameters of Query and Command processes for the software application are calculated using formula (4). The evaluation function takes a set of flowcharts representing the activity algorithms, assesses each of them using the Cognitive Functional Complexity method, and returns their sum. The evaluation results for each flowchart, along with the overall assessments, are presented in Table 5.

To assess realization complexity, the McCabe's Cyclomatic Complexity metric was adopted. This metric quantifies the number of linearly independent execution paths within a program module and serves as an established indicator of code structural complexity

$$CC=E-N+2P, \tag{8}$$

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.

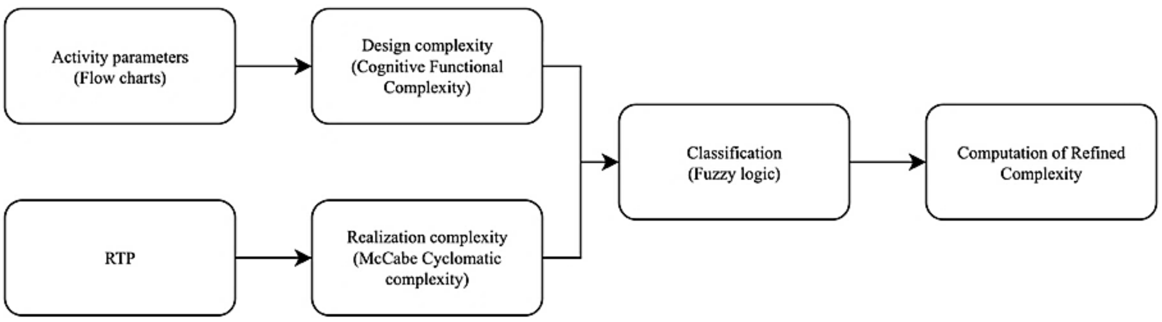

Activity parameters (Flow charts)
Design complexity (Cognitive Functional Complexity)
RTP
Realization complexity (McCabe Cyclomatic complexity)
Classification (Fuzzy logic)
Computation of Refined Complexity

Fig. 6. The algorithm for estimating implementation and modification complexity of processes

Table 2

Command process for CQRS

Activity	Description	In	Out
Create command	Receive a mutation request and transform it into a command	req	command
Validate command	Validate the command type and properties	command	command
Route command	Call the appropriate command handler, providing the command as a parameter	command	command
Fetch aggregate CQRS	Retrieve the aggregate from cache or rebuild it by replaying events onto a new aggregate or the latest snapshot	command	aggregate
Update aggregate's state	Call the appropriate aggregate method to generate events for modifying the aggregate's state	aggregate	events
Save aggregate CQRS	Save the events generated during the aggregate's state update to the Event Store. Periodically save the latest aggregate snapshot to the Aggregate Snapshot DB (once per N calls)	events	events
Dispatch events	Publish events to the Event Bus	events	events
Route event	Call the appropriate event handler, providing the event as a parameter	events	event
Handle update projection event CQRS	Process the event and update projections	event	event
Notify client	Notify clients about changes in the system	event	notification

Table 3

Command process for mCQRS

Activity	Description	In	Out
Create command	Receive mutation request and transform it into a command	req	command
Validate command	Validate the command type and properties	command	command
Route command	Call the appropriate command handler, providing the command as a parameter	command	command
Fetch aggregate mCQRS	Retrieve the aggregate from cache or create a new aggregate using the latest snapshot	command	aggregate
Update aggregate's state	Call the appropriate aggregate method to generate events for modifying the aggregate's state	aggregate	events
Apply events onto aggregate	Update the aggregate state by replaying newly generated events	events	aggregate, events
Save aggregate mCQRS	Save the aggregate snapshot to the Snapshot DB. Store the events generated during the aggregate's state update in the Event Store	aggregate, events	events
Dispatch events	Publish events to the Event Bus	events	events
Route event	Call the appropriate Event Handler, providing the event as a parameter	events	event
Handle update projection event mCQRS	Process the event and update projections	event	event
Notify client	Notify clients about changes in the system	event	notification

Table 4

Query process

Activity	Description	In	Out
Create query	Receive a read data request and transform it into a query	req	query
Validate query	Validate the query type and properties	query	query
Fetch projection from DB	Retrieve the projection from the Projection DB	query	projection
Map projection to DTO	Convert the projection data into a DTO	projection	DTO
Return DTO	Send the response containing the DTO to the client	DTO	DTO

Table 5

Cognitive Functional Complexity of development and modification

Activity	Classical CQRS (CWU)		mCQRS (CWU)	
	Development	Modification	Development	Modification
Command process activities				
Create command	2	2	2	2
Validate command	8	2	8	2
Route command	8	4	8	4
Fetch aggregate	22	8	8	4
Update aggregate's state	8	4	8	4
Apply events onto aggregate	–	–	3	0
Save aggregate	7	1	8	0
Dispatch events	6	0	6	0
Route event	8	4	8	4
Handle event (Update projection)	14	6	10	2
Notify client	12	2	12	2
Command process complexity	95	33	81	24
Query process activities				
Create query	2	2	2	2
Validate query	8	2	8	2
Fetch projection from DB	4	4	4	4
Map projection to DTO	2	2	2	2
Return DTO	2	0	2	0
Query process complexity	18	10	18	10

The metric is computed automatically using the SonarCloud static analysis tool, targeting the code repository containing RTP implementations. To ensure robustness of the evaluation, the final complexity value is calculated as the mean cyclomatic complexity across several aggregation roots (Table 6).

The realization complexity of an activity is calculated by summing the Cyclomatic Complexity values of the classes associated with the activity (those that require changes during the development or modification of the activity). The results are summarized in Table 7.

Table 6

McCabe Cyclomatic Complexity of RTP's classes

Class	Classical CQRS	mCQRS	Class	Classical CQRS	mCQRS
Command	1	1	Query handler	2	2
Query	1	1	Aggregate	9	6
Command handler	2	2	Repository	14	10
Event handler	2	2	Projections Repository	23	16
Event	2	2	Controller	13.5	13.5

Table 7

McCabe Cyclomatic Complexity by activities

Activity	Classical CQRS		mCQRS	
Create command	Command	1	Command	1
Validate command	Controller	13.5	Controller	13.5
Route command	Command handler	2	Command handler	2
Fetch aggregate	Command handler Aggregate Repository	25	Command handler Aggregate Repository	18
Update aggregate's state	Aggregate	9	Aggregate	6
Apply events onto aggregate	–	–	Command handler Aggregate	8
Save aggregate	Command handler Repository	16	Command handler Repository	12
Dispatch events	Command handler	2	Command handler	2
Route event	Event handler	2	Event handler	2
Handle event (Update projection)	Event handler Projections Repository	25	Event handler Projections Repository	18
Notify client	Event handler	2	Event handler	2
Create query	Query	1	Query	1
Validate query	Controller	13.5	Controller	13.5
Fetch projection from DB	Query handler Projections Repository	25	Query handler Projections Repository	18
Map projection to DTO	Query handler	2	Query handler	2
Return DTO	Controller	13.5	Controller	13.5

The calculated values for Cognitive Functional Complexity (representing design complexity) and Cyclomatic Complexity (representing realization complexity) are classified into two or more ordinal classes (e. g., five). Each class is assigned a specific weight coefficient (Table 8). The classification approach and corresponding weights are inspired by the use case size point methodology [81].

Table 8

Fuzzy classes

Number (i)	Name	Value
0	Very simple	4
1	Simple	6
2	Average	8
3	Complex	12
4	Very complex	16

The classification is performed using fuzzy logic [82]. Each complexity class is a fuzzy set, which is represented on the coordinate plane as a trapezoidal membership function. The horizontal distance between the top segments of adjacent trapezoids (l_a) is equal to the length of the top segment of each trapezoid, ensuring smooth linear transitions. The upper bound of the scale corresponds to the maximum observed metric values (22 for design complexity and 25 for realization complexity)

$$l_a = \frac{\max_{x \in X}(x)}{n_c \cdot 2 - 1},$$

where X – a set of complexity values; n_c – a number of classes, in the current case $n_c = 5$; x – source complexity value.

A portion of the complexity values lies entirely within a single class, while others fall within overlapping regions between adjacent classes. In such cases, the complexity value is proportionally distributed between the two neighboring sets based on the degree of membership. For example, as illustrated in Fig. 7, a complexity value of 16 falls partially into two adjacent sets: it has a 0.33 membership degree in the "Average" class and a 0.67 degree in the "Complex" class.

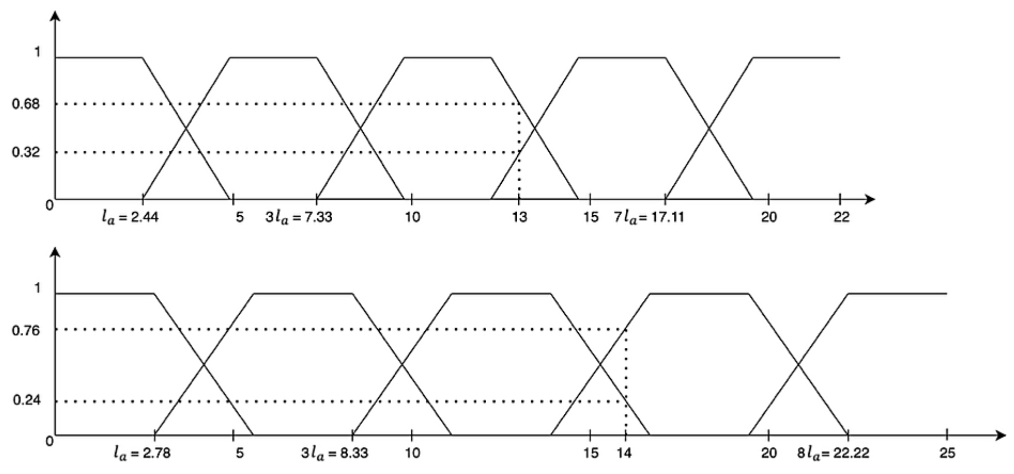


Fig. 7. Example of fuzzy membership distribution across complexity classes for design and realization complexity

Thus, the classified complexity can be computed using the following membership function $\mu(x)$, resulting in normalized values expressed in consistent lexical units across both datasets

$$\mu(x)=\begin{cases} v_i, \\ v_{i+1}\sqrt{\dfrac{2}{l_a}\displaystyle\int_0^{x-(2i+1)l_a}\dfrac{z}{l_a}\mathrm{d}z}+v_i\sqrt{\dfrac{2}{l_a}\displaystyle\int_{x-(2i+1)l_a}^{l_a}1-\dfrac{z}{l_a}\mathrm{d}z}\end{cases}=$$

$$=\begin{cases} v_i, \\ x\in\left[2i\cdot l_a;(2i+1)l_a\right]\subset\mathbb{R}_+,\ i\in\left[0;4\right]\subset\mathbb{N}, \\ v_i*\dfrac{(2i+2)l_a-x}{l_a}+v_{i+1}*\dfrac{x-(2i+1)l_a}{l_a}, \\ x\in\left((2i+1)l_a;(2i+2)l_a\right)\subset\mathbb{R}_+,\ i\in\left[0;3\right]\subset\mathbb{N}, \end{cases} \quad (9)$$

where l_a – the horizontal distance between the top segments of adjacent trapezoids; i – class number; x – a source complexity value; v_i is a class weight.

Next, the coefficients are introduced, and the refined complexity is calculated using Formula (10). In this experiment, both coefficients are set to 0.5, as design complexity and realization complexity are considered equally important. The results are summarized in Table 9

$$\chi_{refined}=w_r\cdot\chi_r+w_d\cdot\chi_d, \quad (10)$$

where w_r, w_d – weights for realization and design complexities; χ_r, χ_d – realization and design complexities.

3.3.2. Performance metrics

Another important parameter of application is performance. For the most objective evaluation, these metrics, like complexity metrics, are best obtained by collecting statistics from multiple applications. Given the absence of such a dataset, it is proposed to use the request processing time measurements (commands and queries response time, and eventual data update time) taken from the RTPs. Response time measurements were conducted for both applications. One hundred read and write requests were sent. The results are presented in Fig. 8 and Table 10.

Table 9

Refined Complexity Calculation

Activity	Classical CQRS				mCQRS			
	Development		Modification		Development		Modification	
	R	*D*	*R*	*D*	*R*	*D*	*R*	*D*
Command process activities								
Create command	4	4	4	4	4	4	4	4
Validate command	8	6.54	8	4	8	6.54	8	4
Route command	4	6.54	4	5.28	4	6.54	4	5.28
Fetch aggregate	16	16	16	6.54	12	6.54	12	5.28
Update aggregate's state	6.48	6.54	6.48	5.28	6	6.54	6	5.28
Apply events onto aggregate	–	–	–	–	6	4.46	0	0
Save aggregate	11.04	6	11.04	4	8	6.54	0	0
Dispatch events	4	6	0	0	4	6	0	0
Route event	4	6.54	4	5.28	4	6.54	4	5.28
Handle update projection event	16	10.92	16	6	12	8	12	4
Notify client	4	8	4	4	4	8	4	4
Command process complexity	77.52	77.08	73.52	44.38	72	69.70	54	37.12
Refined complexity	77.30		58.95		70.85		45.56	
Query process activities								
Create query	4	4	4	4	4	4	4	4
Validate query	8	6.54	8	4	8	6.54	8	4
Fetch projection from DB	16	5.28	16	5.28	12	5.28	12	5.28
Map projection to DTO	4	4	4	4	4	4	4	4
Return DTO	4	4	0	0	4	4	0	0
Query process complexity	36	23.82	32	17.28	32	23.82	28	17.28
Refined complexity	29.91		24.64		27.91		22.64	

Notes: *R* – realization complexity, *D* – design complexity

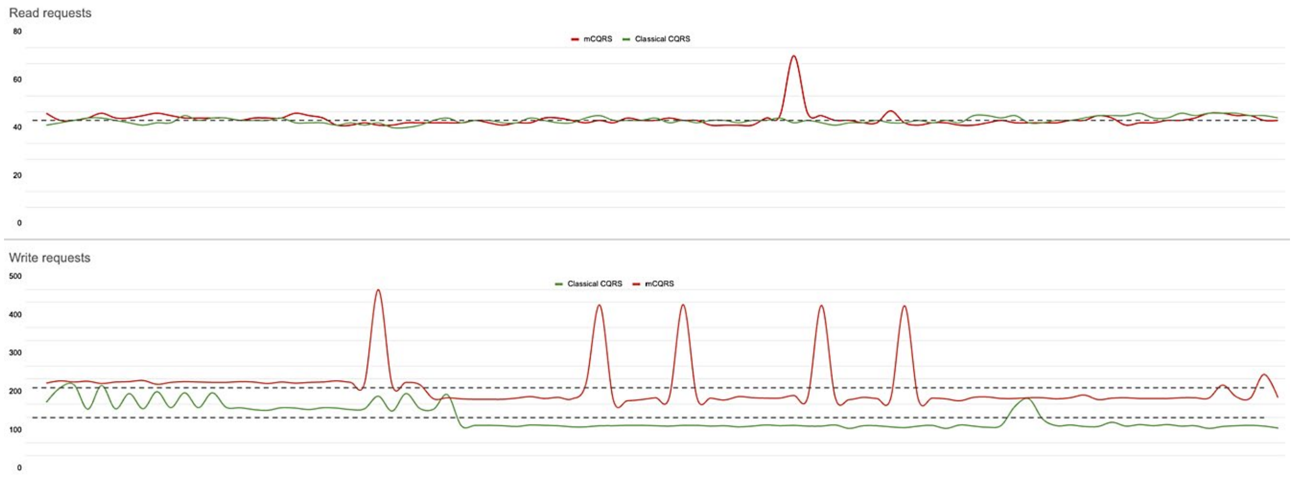


Fig. 8. CQRS and mCQRS response time metrics

Table 10

Software application commands and queries response time

Variation	Read requests, ms	Write requests, ms	Write requests with cache enabled, ms
Classical CQRS	43	132	90
mCQRS	43	210	168

The response time for write requests was measured both with and without aggregation root caching enabled. As seen from the results, caching aggregation roots considerably improves the average application performance. To obtain more accurate performance metrics of the algorithms, subsequent calculations use metrics obtained with aggregation root caching disabled.

In cases of uncertainty, when performance metrics obtained from different experiments are inconsistent, the results can also be classified by acceptability levels (ranging from Minimal to Excellent). To enable this, stakeholders define acceptable response time thresholds for each class. Once thresholds are established, the collected performance metrics are normalized in accordance with the defined acceptability levels, using a fuzzification algorithm.

3.4. Application of the MCDA method

During previous steps, metrics for complexity and performance for individual processes of the considered architectural approaches were collected. At this stage, MCDA method can be applied to obtain quantitative characteristics of how well an approach aligns with the application being developed.

The Analytical Hierarchy Process method is proposed for use as the MCDA method. This method is based on constructing a pairwise comparison matrix of alternative parameters for each evaluation criterion (*B*) and a weight vector of evaluation criteria (*W*). Multiplying the matrix by the vector yields a set of values corresponding to the weight of each alternative (*R*). The optimal choice (r_{opt}) is calculated as the maximum value from this set:

$$R = B \times W,\ r_{opt} = \max\left| r_1; r_2; \ldots; r_n \right|.$$

Let's evaluate the applicability of the considered architecture variations to the same software application under varying conditions. Software application parameters are specified in Table 11.

Table 11

Software application parameters

Name	Value
Commands, pcs	41
Queries, pcs	19
Adding new processes, %	50
Updating existing processes, %	50

The applicability of a particular variation is influenced by the estimated read-to-write request ratio and business priorities (whether the goal is to develop a functioning application as quickly as possible or to achieve maximum performance). These parameters are unique to each application. Therefore, for the experiment, calculations were performed, considering the variation of these two parameters. The expected percentage of read/write operations, represented along the *Requests* axis in Fig. 9. A value of 0 corresponds to the absence of read requests, while 100 indicates that the application will receive only read requests with no write operations. The prioritization coefficient of application performance versus development speed (simplicity of development). This parameter is represented along the *Priority* axis, where 0 corresponds to a complete priority on development speed, while 100 assumes unlimited development time with the requirement to achieve maximum possible performance.

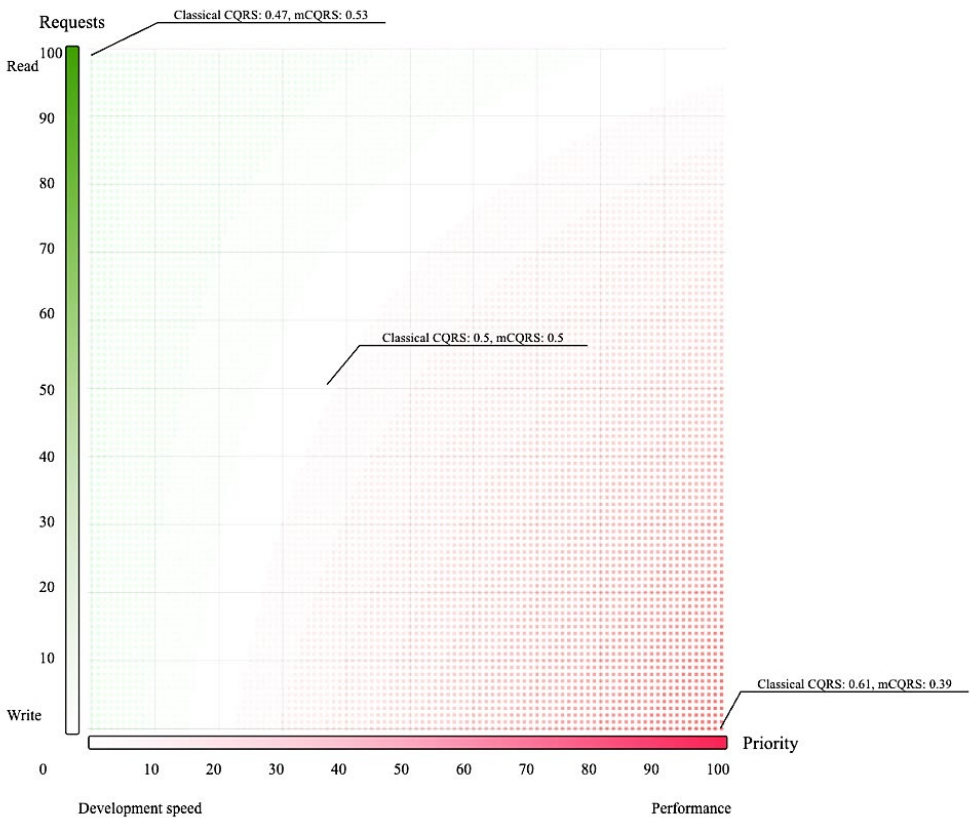


Fig. 9. CQRS and mCQRS approaches applicability to software applications with different Requests and Priority parameters

Each parameter evaluation is based on gathered and calculated metrics rather than expert judgment. The applicability assessment is performed for each combination of dynamic parameters.

The data transformation for application of the AHP method includes several steps:

– Define key parameters of the application under development:

1) the approximate number of command and query processes that need to be implemented for the minimum valuable product (MVP) version of the application;

2) the estimated ratio of new process additions to modifications of existing ones after the MVP release to assess application maintainability.

– Calculate and normalize the complexity coefficients of the software application.

– Define the measured parameters of the approaches (classical CQRS and mCQRS):

1) performance parameters (average request duration for read and write operations);

2) complexity parameters (development and modification complexity of command and query processes).

– Normalize the given parameters using cost criteria [49], as for all the listed parameters, a lower value is considered better when evaluating the approach.

– Introduce dynamic parameters:

1) the prioritization coefficient of development speed versus system productivity;

2) the expected read/write operation ratio that the application should support during operation.

– In an n-dimensional loop (e. g., a 2-dimensional one), the application parameters are modified considering the dynamic parameters for each value from 1 to 100%. The modification consists of the following steps:

1) multiplying the application's complexity parameters by the development speed prioritization coefficient;

2) adding the average duration parameters of read/write operations, multiplied by the application productivity prioritization coefficient.

– AHP method application. As a result of the, an n-dimensional array of applicability values is obtained for each considered method. The data from these arrays can be visualized on a bitmap chart to determine under which conditions each approach is more suitable.

Example of the visualization, as well as the implementation of the algorithm are available on the GitHub repository [80]. The bitmap visually represents the regions where each architectural variation is most suitable (Fig. 9). Thus, this enables the ability to objectively select one of the architectural variations for the initial application implementation.

Fig. 9 shows that in 37% of cases, where the application development priority leans towards simplicity and the expected number of read operations exceeds write operations, the AHP method suggests using the mCQRS approach. In 50% of cases, where the priority shifts towards application performance, the classical CQRS approach is recommended. In the remaining percentage of cases, the applicability evaluation of both approaches is equal.

These calculations are not entirely precise, as they are based on analyzing the performance of Classical CQRS and mCQRS variations based on RTPs (Table 10). To obtain more accurate results, it is necessary to collect statistics from as many applications as possible. Also, it should be noted that when calculating development complexity using the cognitive functional complexity method (as demonstrated in this example), the level of detail in flowcharts and their alignment with the actual program code play a crucial role. To ensure a sufficient level of detail and consistency, it is advisable to apply control methods such as cross-reviews and team demonstrations of the work results.

3.5. Example of calculating the transition complexity between Architecture Variations

Thus, the previous example demonstrates how, based on theoretical calculations and statistical data, the choice of an evolutionary branch for the developed software application can be objectively justified at the design stage of development.

The next step is to provide a way for the assessment and expenses evaluation of the application transition from one architectural variation to another. The first question is how to prove the necessity of the transition. To prove that, the approach for selecting the variation described in the previous example can be used. The second question is measuring the expenses for transitioning between architectural variations and the associated risks.

According to the suggested approach (Section 2.4.2) the first step is to define the differences between variations. The differences can be defined using the formal descriptions of the same process in different solutions: some activities won't need any modification. After identifying the differences, the next step is to evaluate the distance between the realizations of the activities in terms of complexity of sub-activities that should be added or removed.

Attachment 2 in [80] presents the transition design complexity calculated using (5). These estimations cover three architectural variations included in the evolutionary roadmap: Pure CQRS, Classical CQRS, and mCQRS. The transition complexity is evaluated for migrations from Pure CQRS to both Classical CQRS and mCQRS, as well as for transitions between mCQRS and Classical CQRS.

Additionally, the realization complexity is computed for the code difference between RTP's aggregate using the Halstead complexity measures method [44]. These metrics are used to quantify the complexity of software by analyzing the composition of code within program modules. The approach calculates three primary complexity metrics of a program: volume (*V*), difficulty (*D*), and effort (*E*). The formula for calculating the effort is as follows:

$$V = (N_1 + N_2) \cdot \log_2 (n_1 + n_2),$$

$$D = (n_1 / 2) \cdot (N_2 / n_2),$$

$$E = D \cdot V,$$

where n_1 represents the count of distinct operators; n_2 represents the count of distinct operands; N_1 – the total number of operators; N_2 – the total number of operands.

Using the proposed method, both the design and realization complexity values are classified and aggregated to derive the refined complexity estimate. Finally, the total transition complexity is calculated using (6), based on application parameters defined in Table 11. A summary of the results is provided in Table 12.

Table 12

Software application transition complexity metrics

Transition	Design complexity		Target Realization complexity		Refined complexity	c_{p_r}, pcs	Total transition complexity
	Cognitive	Classified	Halstead	Classified			
Pure CQRS to Classical CQRS	12	6	857,750	16	11	41	451
Pure CQRS to mCQRS	26	12	376,671	7.9	9.95		407.95
Classical CQRS to mCQRS	28	12	161,418	5.38	8.69		356.29
mCQRS to Classical CQRS	39	16	687,773	12.88	14.44		592.04

The table does not include transition complexity calculation results for query processes, as they remain the same across all three considered variations, making the transition complexity equal to zero.

As a result, an evolutionary architecture roadmap can be constructed, allowing to obtain a more precise understanding of the probable evolution ways considering the expenses needed to implement them (Fig. 10).

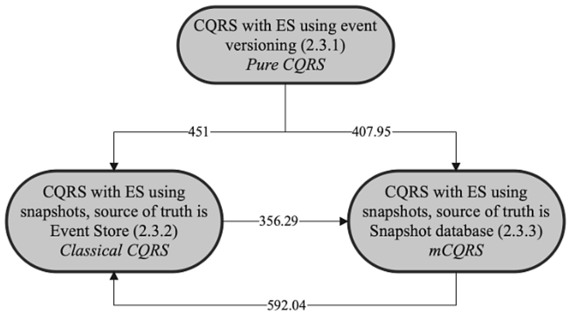

CQRS with ES using event versioning (2.3.1) Pure CQRS
451
407.95
CQRS with ES using snapshots, source of truth is Event Store (2.3.2) Classical CQRS
356.29
CQRS with ES using snapshots, source of truth is Snapshot database (2.3.3) mCQRS
592.04

Fig. 10. Example software application estimated Evolutionary roadmap

It is important to note that the distances between architectural variations were computed using a modified Cognitive Functional Complexity method and are expressed in the same units as the process development and modification estimates presented in Table 5. This alignment allows for meaningful comparison between the values in Tables 5 and 12, thereby supporting informed decision-making during planning. For example, the estimated complexity of realizing a single command-handling process in the Classical CQRS architecture variation is 95 CWU, while the complexity of migrating this process to the mCQRS variation is 28 CWU. Consequently, the total complexity of implementing the process followed by migration amounts to 123 CWU.

Therefore, the proposed approach does not account for data migration when calculating transition complexity, which can be considered a limitation.

3.6. Discussion

The main results of the approach application are:

– a bitmap chart that demonstrates the applicability of each architectural variation in different scenarios, depending on the development priorities of the application. This information assists in selecting the most appropriate architectural variation at the early stages of software application development;

– an evolutionary roadmap that provides a quantitative assessment of the distance between architectural variations. This assessment enables the estimation of the complexity involved in transitioning the functional part of an application from one architectural variation to another.

The bitmap chart shows that the Classical CQRS variation is more suitable when application performance is a higher priority than development speed, or when the application is expected to handle a high volume of write requests. This is explained by the fact that Classical CQRS provides higher write performance, although it is more complex to implement.

In contrast, the mCQRS variation is better suited for applications that do not require high write operation performance (typically when the percentage of write operations is very low). This is due to its simpler structure, which leads to reduced development complexity and lower implementation costs compared to Classical CQRS.

It can be seen that the evolution complexity for the selected RTP from Pure CQRS to mCQRS (407.95) is lower than that to Classical CQRS (451). This is explained by the fact that Classical CQRS requires more resources and places higher demands on developer expertise during implementation. The relatively small difference between the values (451 – 407.95 = 43.05) is due to the structural similarity between Pure CQRS and Classical CQRS.

The roadmap also shows that the distance between Classical CQRS and mCQRS is direction-dependent. The transition from Classical CQRS to mCQRS has a complexity value of 356.29, as the change simplifies the architecture. In the opposite direction, mCQRS to Classical CQRS, the complexity is higher – 592.04.

By evaluating the development and migration complexity of a single typical write process, the following results are obtained:

– 77.3 (Classical CQRS implementation) + 8.69 (transition from Classical CQRS to mCQRS) = 85.99;

– 70.85 (mCQRS implementation) + 14.44 (transition from mCQRS to Classical CQRS) = 85.29.

This leads to the conclusion that, when choosing between these two architectural variations at the design stage and their applicability scores are equal, it is more efficient to initially implement mCQRS. If a transition to Classical CQRS is later required, the overall cost per typical write process would be lower by 85.99 – 85.29 = 0.7 units.

Existing architecture evaluation methods such as SAAM, ATAM, and parametric-based approaches like AHP and its modifications (e. g., LiVASAE) are primarily designed to compare fundamentally different architectural styles. Consequently, their evaluation criteria are defined at a coarse-grained level, targeting general architectural characteristics rather than fine-grained distinctions between architectural variations. These methods lack the level of detail required to differentiate between variations of the same architectural family, such as those within the CQRS with ES paradigm. When such methods are applied to architectural variations, a significant portion of evaluation parameters becomes irrelevant or indistinguishable across alternatives, leading to reduced accuracy and limited practical value.

Furthermore, scenario-based approaches like SAAM and ATAM are heavily reliant on expert judgment, which introduces subjectivity and the risk of human error. Their outputs are largely qualitative, consisting of narrative assessments, risk descriptions, and supporting documentation. While valuable in certain contexts, these results do not provide quantitative criteria for selecting between closely related architectural options.

Parametric-based methods such as AHP and its modifications (e. g., LiVASAE) are typically do not include a mechanism for obtaining objective measurement data to be used as input for the multi-criteria analysis. Moreover, the outputs of such methods usually do not include design documentation, such as formal descriptions of processes, their parameters, or flowcharts, which are produced as part of the proposed approach.

In contrast, the approach proposed in this study is specifically tailored to evaluate architectural variations within a single architectural style (CQRS with ES). It addresses the limitations mentioned above by providing quantitative assessment based on measurable complexity and performance metrics. The provided approach relies on statistical and computational metrics instead of expert judgment, making the assessment results more objective and applicable to the task of selecting an architectural variation.

Thus, the practical application of the proposed method reduces uncertainty when making informed decisions about selecting a CQRS with ES architectural variation or transitioning to another evolutionary level or branch of the software system's architecture. Subsequently, it would help to improve long-term planning and avoid unnecessary expenses related to resolving additional technical issues or migrating the application's architecture to a different variation.

However, a key limitation of the approach is its dependency on the accuracy of metric calculations. The assessment process is based on predefined algorithms that are inherently linked to use cases. To minimize errors, use cases are classified and clustered according to their characteristics. Without such clustering, the risk of errors increases significantly. Even with clustering, a thorough examination is required to ensure reliable results.

Based on the results of the study, the assessment outcome is strongly influenced by the quality and level of detail of the flowcharts developed during the planning phase, as these directly impact the complexity metric calculations. To enhance accuracy, rigorous data quality control in both flowchart development and metric calculations is essential. However, the issues and their solutions connected to developing and testing flowcharts fall outside the scope of the present work and represent a prospect for further research.

A research limitation is the lack of quantitative comparison between proposed approach and other software architecture evaluation methods. Testing the method on several individual applications does not provide objective results, as each project may differ in technology stack, team composition, and implementation context. A method that works well for one application may yield different outcomes in another due to these variations. To overcome this limitation, a Representative Test Project was used. It reflects not a single application but an entire class of applications. In addition, the evaluation was supported by empirical data and observations from previously developed real-world applications based on practical experience.

Provided example of evaluating the complexity of implementation, modification and functional transitions between architectural variations shows (Attachments 1 and 2 [80]) the specifics of practical use of the proposed decision-making approach. Having this information allows for planning a long-term development of the software application, taking into account the risks and the effort costs associated with changing the architectural variation.

4. Conclusions

1. The command and query processes of CQRS with ES architectural variations have been formalized and described. To support this, a formal activity-centric knowledge representation model based on scalable finite state machines and Petri nets was proposed. This allows for the analysis of each activity individually and the subsequent aggregation of metrics for the entire process and, ultimately, for the corresponding architectural variation.

2. The study provides an analysis of key parameters influencing the selection of CQRS with ES architectural variations, including implementation complexity, maintainability, and performance. A design complexity estimation example is presented using an activity-centric model, along with a performance evaluation based on the development of an RTP and experimental measurements. To account for both design and realization complexity, the refined complexity metric was calculated using a method based on fuzzy logic. The refined complexity of the command process was 77.30 for Classical CQRS and 70.85 for mCQRS. The average command processing time was 132 ms for Classical CQRS and 210 ms for mCQRS.

3. To compare the applicability of architectural variations to a software project, the AHP method was employed. To use the metrics calculated in the previous steps as input parameters for AHP, they were transformed using the proposed algorithm. This transformation algorithm was automated, and applicability calculations were performed for each considered variation across various project priority scenarios. The results showed that the mCQRS approach is recommended in 37% of cases, while the classical CQRS approach is preferred in 50%. In the remaining cases, both approaches demonstrated equal applicability.

4. As an example, an evolutionary roadmap that includes the Pure CQRS, Classical CQRS, and mCQRS variations was defined. Based on the RTP and the metrics obtained during the evaluation of architectural variations, the effort required for architectural transitions between evolutionary stages (variations) was assessed. For instance, transitioning a single command-handling process from Classical CQRS to mCQRS has a refined complexity of 8.69, while transitioning in the opposite direction results in a complexity of 14.44.

Acknowledgments

We sincerely appreciate DBB Software company [23] for providing their proprietary platform, which served as the foundation for our experiment. This platform offered essential capabilities for our research, ensuring the accuracy and reliability of our experimental results.

We also want to express our deep appreciation to Volodymyr Khandetsky, Head of Electronic Department, for his valuable comments and suggestions.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this research, whether financial, personal, authorship or otherwise, that could affect the research and its results presented in this paper.

Financing

The research was performed without financial support.

Data availability

Manuscript has associated data in a data repository

Use of artificial intelligence

The authors have used artificial intelligence technologies within acceptable limits to provide their own verified data, which is described in the research methodology section.

ChatGPT was used for improving the translation quality and structuring of individual paragraphs.

References

1. Fowler, M., Rice, D., Foemmel, M., Hieatt, E., Mee, R., Stafford, R. (2002). *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 560. Available at: https://dl.ebooksworld.ir/motoman/Patterns%20of%20Enterprise%20Application%20Architecture.pdf
2. Hohpe, G., Woolf, B. (2011). *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Boston: Addison-Wesley. Available at: https://ptgmedia.pearsoncmg.com/images/9780321200686/samplepages/0321200683.pdf
3. Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley. Available at: https://fabiofumarola.github.io/nosql/readingMaterial/Evans03.pdf
4. Zhong, Y., Li, W., Wang, J. (2019). Using Event Sourcing and CQRS to Build a High Performance Point Trading System. *Proceedings of the 2019 5th International Conference on E-Business and Applications*. Bangkok, New York, 16–19. https://doi.org/10.1145/3317614.3317632
5. Betts, D., Dominguez, J., Melnik, G., Simonazzi, F., Subramanian, M. (2012). *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*. Microsoft patterns & practices. Available at: https://download.microsoft.com/download/e/a/8/ea8c6e1f-01d8-43ba-992b-35cfcaa4fae3/cqrs_journey_guide.pdf
6. Fowler, M. (2011). *CQRS*. Available at: https://martinfowler.com/bliki/CQRS.html
7. Young, G. (2010). *CQRS Documents by Greg Young*. Available at: https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
8. Young, G. (2017). *Event Centric: Finding Simplicity in Complex Systems*. Boston: Addison-Wesley Professional, 560.
9. Taylor, H., Yochem, A., Phillips, L., Martinez, F. (2009). *Event-Driven Architecture: How SOA Enables the RealTime Enterprise*. Boston: Addison-Wesley, 272.
10. Vernon, V. (2013). *Implementing Domain-Driven Design*. Boston: Addison Wesley, 656.
11. Ford, N., Parsons, R., Kua, P., Sadalage, P. (2022). *Building evolutionary architectures*. Sebastopol: O'Reilly Media, 262.

12. *Event Sourcing pattern*. Microsoft. Available at: https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing
13. Comartin, D. (2021). *Snapshots in Event Sourcing for Rehydrating Aggregates*. CodeOpinion. Available at: https://codeopinion.com/snapshots-in-event-sourcing-for-rehydrating-aggregates/
14. Evsyukov, O. (2020). *Bermudskyi Ahrehat. I spasenye utopaiushchykh*. Domain-Driven Design Injection. Available at: https://youtu.be/Br4TL-486ZM?t=1500
15. Young, G. (2017). *Versioning in an Event Sourced System*. Available at: https://leanpub.com/esversioning/read
16. Kleanthous, S. (2021). *Event immutability and dealing with change*. Kurrent. Available at: https://www.eventstore.com/blog/event-immutability-and-dealing-with-change
17. Zheng, Z., Xie, S., Dai, H., Chen, X., Wang, H. (2017). An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends. *2017 IEEE International Congress on Big Data (BigData Congress)*. Honolulu, 557–564. https://doi.org/10.1109/bigdatacongress.2017.85
18. Regulation (EU) 2016/679 of the European Parliament and of the Council of 27 April 2016 on the protection of natural persons with regard to the processing of personal data and on the free movement of such data, and repealing Directive 95/46/EC (General Data Protection Regulation) (Text with EEA relevance). *Official Journal of the European Union, 119*, 4.5.2016, 1–88. Available at: https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng
19. Vasconcellos, P. R. G., Bezerra, V. M., Bianchini, C. P. (2018). Applying Event Sourcing in a ERP System: A Case Study. *2018 XLIV Latin American Computer Conference (CLEI)*. São Paulo, 80–89. https://doi.org/10.1109/clei.2018.00019
20. Korkmaz, N., Nilsson, M. (2014). *Practitioners' view on command query responsibility segregation*. [Master's thesis; Lund University]. Available at: https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=4864802&fileOId=4864803
21. Lytvynov, O., Hruzin, D., Frolov, M. (2024). On the migration of domain driven design to CQRS with event sourcing software architecture. *Information Technology: Computer Science, Software Engineering and Cyber Security, 1*, 50–60. https://doi.org/10.32782/it/2024-1-7
22. Pandiya, D. K., Charankar, N. G. (2024). Optimizing Performance and Scalability in Micro Services with CQRS Design. *International Journal of Engineering Research & Technology, 13 (4)*. Available at: https://www.ijert.org/optimizing-performance-and-scalability-in-micro-services-with-cqrs-design
23. *DBB Software's*. Available at: https://dbbsoftware.com/
24. *ISO/IEC/IEEE 24748-1:2024(en) Systems and software engineering – Life cycle management – Part 1: Guidelines for life cycle management* (2024). ISO. Available at: https://www.iso.org/obp/ui/en/#iso:std:iso-iec-ieee:24748:-1:ed-2:v1:en
25. Sobhy, D., Bahsoon, R., Minku, L., Kazman, R. (2021). Evaluation of Software Architectures under Uncertainty. *ACM Transactions on Software Engineering and Methodology, 30 (4)*, 1–50. https://doi.org/10.1145/3464305
26. Bahsoon, R., Emmerich, W. (2003). Evaluating software architectures: development, stability, and evolution. *ACS/IEEE International Conference on Computer Systems and Applications*. Tunis, 47. https://doi.org/10.1109/aiccsa.2003.1227480
27. Kazman, R., Bass, L., Abowd, G., Webb, M. (1994). SAAM: a method for analyzing the properties of software architectures. *Proceedings of 16th International Conference on Software Engineering*. Sorrento, 81–90. https://doi.org/10.1109/icse.1994.296768
28. Kazman, R., Klein, M., Clements, P. (2000). ATAM: Method for Architecture Evaluation. *Technical report CMU/SEI-2000-TR-004. Carnegie Mellon Software Engineering Institute. Pittsburgh*. Available at: https://www.sei.cmu.edu/documents/629/2000_005_001_13706.pdf
29. Kazman, R., Jai Asundi, Klein, M. (2001). Quantifying the costs and benefits of architectural decisions. *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. Toronto, 297–306. https://doi.org/10.1109/icse.2001.919103
30. Faniyi, F., Bahsoon, R., Evans, A., Kazman, R. (2011). Evaluating Security Properties of Architectures in Unpredictable Environments: A Case for Cloud. *2011 Ninth Working IEEE/IFIP Conference on Software Architecture*. Washington, 127–136. https://doi.org/10.1109/wicsa.2011.25
31. Zarghami, M., Szidarovszky, F. (2011). Introduction to Multicriteria Decision Analysis. *Multicriteria Analysis*. Berlin, Heidelberg: Springer, 1–12. https://doi.org/10.1007/978-3-642-17937-2_1
32. Brunelli, M. (2015). Introduction to the Analytic Hierarchy Process. *SpringerBriefs in Operations Research*. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-12502-2
33. Al-Naeem, T., Gorton, I., Babar, M. A., Rabhi, F., Benatallah, B. (2005). A quality-driven systematic approach for architecting distributed software applications. *Proceedings of the 27th International Conference on Software Engineering – ICSE '05*. St. Louis, 244–253. https://doi.org/10.1145/1062455.1062508
34. Kim, C.-K., Lee, D.-H., Ko, I.-Y., Baik, J. (2007). A Lightweight Value-based Software Architecture Evaluation. *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2007)*. Washington, 646–649. https://doi.org/10.1109/snpd.2007.507
35. Bourque, P., Fairley, R. E. (2014). *Guide to the Software Engineering Body of Knowledge – SWEBOK V3.0*. Piscataway: IEEE and IEEE Computer Society Press. Available at: https://www.researchgate.net/publication/342452008_Guide_to_the_Software_Engineering_Body_of_Knowledge_-_SWEBOK_V30
36. Nivedhaa, N. (2024). Software architecture evolution: Patterns, trends, and best practices. *International Journal of Computer Sciences and Engineering, 1*, 1–14. Available at: https://www.researchgate.net/publication/384019495_SOFTWARE_ARCHITECTURE_EVOLUTION_PATTERNS_TRENDS_AND_BEST_PRACTICES
37. Milić, M., Makajić-Nikolić, D. (2022). Development of a Quality-Based Model for Software Architecture Optimization: A Case Study of Monolith and Microservice Architectures. *Symmetry, 14 (9)*, 1824. https://doi.org/10.3390/sym14091824
38. *ISO/IEC/IEEE 24765:2017 Systems and software engineering – Vocabulary* (2017). ISO. Available at: https://www.iso.org/standard/71952.html
39. Mohapatra, S. K., Prasad, S. (2015). Finding Representative Test Case for Test Case Reduction in Regression Testing. *International Journal of Intelligent Systems and Applications, 7 (11)*, 60–65. https://doi.org/10.5815/ijisa.2015.11.08
40. Mens, T. (2016). *Research trends in structural software complexity*. arXiv: 1608.01533v1. https://doi.org/10.48550/arXiv.1608.01533
41. Sarala, S., Abdul Jabbar, P. (2010). Information flow metrics and complexity measurement. *2010 3rd International Conference on Computer Science and Information Technology*. Chengdu, 575–578. https://doi.org/10.1109/iccsit.2010.5563667
42. Beyer, D., Häring, P. (2014). A formal evaluation of DepDegree based on weyuker's properties. *Proceedings of the 22nd International Conference on Program Comprehension*. Hyderabad, 258–261. https://doi.org/10.1145/2597008.2597794
43. McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering, SE-2 (4)*, 308–320. https://doi.org/10.1109/tse.1976.233837
44. Halstead, M. H. (1977). *Elements of Software Science*. New York: Elsevier Science Inc., 128.
45. Stepien, B. (2003). Software development cost estimation methods and research trends. *Computer Science, 5 (1)*, 67–86. Available at: https://www.researchgate.net/publication/50365764_Software_Development_Cost_Estimation_Methods
46. Wang, Y., Shao, J. (2003). Measurement of the cognitive functional complexity of software. *Proceedings of the 2nd IEEE International Conference on Cognitive Informatics (ICCI '03)*. Washington, 67–74. https://doi.org/10.1109/COGINF.2003.1225955
47. Zlaugotne, B., Zihare, L., Balode, L., Kalnbalkite, A., Khabdullin, A., Blumberga, D. (2020). Multi-Criteria Decision Analysis Methods Comparison. *Environmental and Climate Technologies, 24 (1)*, 454–471. https://doi.org/10.2478/rtuect-2020-0028
48. Jahanshahi, H., Alijani, Z., Mihalache, S. F. (2023). Towards Sustainable Transportation: A Review of Fuzzy Decision Systems and Supply Chain Serviceability. *Mathematics, 11 (8)*, 1934. https://doi.org/10.3390/math11081934
49. Vafaei, N., Ribeiro, R. A., Camarinha-Matos, L. M. (2016). Normalization Techniques for Multi-Criteria Decision Making: Analytical Hierarchy Process Case Study. *Technological Innovation for Cyber-Physical Systems*. Costa de Caparica, 261–269. https://doi.org/10.1007/978-3-319-31165-4_26
50. Young, G. (2023). *GitHub: EventStore repository*. Available at: https://github.com/gregoryyoung/EventStore
51. Driscoll, M. (2017). *The Publish-Subscribe Pattern. WxPython Recipes*. Berkeley: Apress, 43–50. https://doi.org/10.1007/978-1-4842-3237-8_4
52. *CQRS. Practical and focused guide for survival in post-CQRS world: Projections*. Available at: http://cqrs.wikidot.com/doc:projection
53. Hierons, R. M., Türker, U. C. (2017). Parallel Algorithms for Generating Distinguishing Sequences for Observable Non-deterministic FSMs. *ACM Transactions on Software Engineering and Methodology, 26 (1)*, 1–34. https://doi.org/10.1145/3051121
54. Wang, J., Tepfenhart, W. (2019). *Petri Nets. Formal Methods in Computer Science*. Chapman and Hall, CRC, 201–243. https://doi.org/10.1201/9780429184185-8
55. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M. (2010). Learning Communicating Automata from MSCs. *IEEE Transactions on Software Engineering, 36 (3)*, 390–408. https://doi.org/10.1109/tse.2009.89
56. Brand, D., Zafiropulo, P. (1983). On Communicating Finite-State Machines. *Journal of the ACM, 30 (2)*, 323–342. https://doi.org/10.1145/322374.322380
57. Harel, D. (1987). Statecharts: a visual formalism for complex systems. *Science of Computer Programming, 8 (3)*, 231–274. https://doi.org/10.1016/0167-6423(87)90035-9
58. Booch, G., Rumbaugh, J., Jacobson, I. (1999). *The Unified Modeling Language User Guide*. Addison Wesley Longman Publishing Co., Inc., 512. Available at: https://patologia.com.mx/informatica/uug.pdf
59. Alur, R., Etessami, K., Yannakakis, M. (2001). *Analysis of Recursive State Machines. Computer Aided Verification*. Berlin, Heidelberg: Springer-Verlag, 207–220. https://doi.org/10.1007/3-540-44585-4_18
60. Alur, R., Benedikt, M., Etessami, K., Godefroid, P., Reps, T., Yannakakis, M. (2005). Analysis of recursive state machines. *ACM Transactions on Programming Languages and Systems, 27 (4)*, 786–818. https://doi.org/10.1145/1075382.1075387

61. Chatterjee, K., Kragl, B., Mishra, S., Pavlogiannis, A. (2017). Faster algorithms for weighted recursive state machines. *Proceedings of the 26th European Symposium on Programming, ESOP 2017 held as Part of the European Joint Conferences on Theory and Practice of Software.* Uppsala: Springer, 287–313. https://doi.org/10.48550/arXiv.1701.04914

62. Dubslaff, C., Wienhöft, P., Fehnker, A. (2024). Lazy model checking for recursive state machines. *Software and Systems Modeling, 23 (2),* 369–401. https://doi.org/10.1007/s10270-024-01159-z

63. Simon, E., Stoffel, K. (2009). State machines and petri nets as a formal representation for systems life cycle management. *Proceedings of the International Conference Information Systems.* Barcelona, 275–282. Available at: https://www.researchgate.net/publication/228721890_State_machines_and_petri_nets_as_a_formal_representation_for_systems_life_cycle_management

64. Van Der Aalst, W. M. P. (1998). The Application of Petri Nets to Workflow Management. *Journal of Circuits, Systems and Computers, 8 (1),* 21–66. https://doi.org/10.1142/s0218126698000043

65. Jensen, K. (1996). Coloured Petri Nets. *Monographs in Theoretical Computer Science. An EATCS Series.* Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-662-03241-1

66. Ullman, J. D. (1998). *Elements of ML Programming.* New Jersey: Prentice-Hall. Available at: https://www.scribd.com/doc/221508984/Elements-of-Ml-Programming

67. Fehling, R. (1993). A concept of hierarchical Petri nets with building blocks. *Advances in Petri Nets 1993,* 148–168. https://doi.org/10.1007/3-540-56689-9_43

68. Farwer, B., Misra, K. (2002). Modelling with hierarchical object Petri nets. *Fundamenta Informaticae, 55 (2),* 129–147. Available at: https://www.researchgate.net/publication/220445187_Modelling_with_Hierarchical_Object_Petri_Nets

69. Chistikov, D., Czerwinski, W., Hofman, P., Mazowiecki, F., Sinclair-Banks, H. (2023). Acyclic Petri and Workflow Nets with Resets. *Proceedings of the 43rd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science.* Dagstuhl Castle, Leibniz Center for Informatics, 284, 1–18. https://doi.org/10.4230/LIPIcs.FSTTCS.2023.16

70. Lomazova, I. A., Mitsyuk, A. A., Rivkin, A. (2021). *Soundness in Object-centric Workflow Petri Nets.* arXiv:2112.14994v1. https://doi.org/10.48550/arXiv.2112.14994

71. Blondin, M., Mazowiecki, F., Offtermatt, P. (2022). The complexity of soundness in workflow nets. *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science.* New York, 1–13. https://doi.org/10.1145/3531130.3533341

72. Meyer, T. (2023). A Symmetric Petri Net Model of Generic Publish-Subscribe Systems for Verification and Business Process Conformance Checking. *Proceedings of the International Workshop on Petri Nets and Software Engineering (PNSE '23).* Lisbon: CEUR, Aachen, 88–109. Available at: https://ceur-ws.org/Vol-3430/paper6.pdf

73. Ding, J., Zhang, D. (2015). Modeling and Analyzing Publish Subscribe Architcture using Petri Nets. *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering, 2015.* Pittsburgh: KSI Research Inc., 589–594. https://doi.org/10.18293/seke2015-232

74. Genrich, H. J. (1991). *Predicate / Transition Nets.* High-Level Petri Nets. Berlin, Heidelberg: Springer-Verlag, 3–43. https://doi.org/10.1007/978-3-642-84524-6_1

75. Lytvynov, O. A., Hruzin, D. L. (2024). Critical causal events in systems based on cqrs with event sourcing architecture. *Radio Electronics, Computer Science, Control, 3,* 119–143. https://doi.org/10.15588/1607-3274-2024-3-11

76. Minsky, M. (1974). *A Framework for Representing Knowledge. MIT Research Lab Technical Report.* Cambridge: Massachusetts Institute of Technology. Available at: https://courses.media.mit.edu/2004spring/mas966/Minsky%201974%20Framework%20for%20knowledge.pdf

77. Harel, D., Peleg, D. (1985). Process logic with regular formulas. *Theoretical Computer Science, 38,* 307–322. https://doi.org/10.1016/0304-3975(85)90225-7

78. Levenshtein, V. (1965). Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Doklady Akademii nauk SSSR, 10,* 707–710. Available at: https://nymity.ch/sybilhunting/pdf/Levenshtein1966a.pdf

79. Cockburn, A. (2000). *Writing Effective Use Cases.* Boston: Addison-Wesley Professional, 304. Available at: https://kurzy.kpi.fei.tuke.sk/zsi/resources/CockburnBookDraft.pdf

80. Hruzin, D. (2025). *GitHub: CQRS-variations-test repository.* Available at: https://github.com/dmitryhruzin/CQRS-variations-test

81. Braz, M., Vergilio, S. (2006). Software Effort Estimation Based on Use Cases. *30th Annual International Computer Software and Applications Conference (COMPSAC'06).* Chicago, 221–228. https://doi.org/10.1109/compsac.2006.77

82. Zadeh, L. A. (1965). Fuzzy sets. *Information and Control, 8 (3),* 338–353. https://doi.org/10.1016/s0019-9958(65)90241-x

✉***Oleksandr Lytvynov,*** *PhD, Associate Professor, Department of Electronic Computing Machinery, Oles Honchar Dnipro National University, Dnipro, Ukraine, e-mail: lytvynov_o@365.dnu.edu.ua, ORCID: https://orcid.org/0000-0001-7660-1353*

Dmytro Hruzin, *PhD Student, Department of Electronic Computing Machinery, Oles Honchar Dnipro National University, Dnipro, Ukraine, ORCID: https://orcid.org/0009-0004-8534-2559*

✉*Corresponding author*