

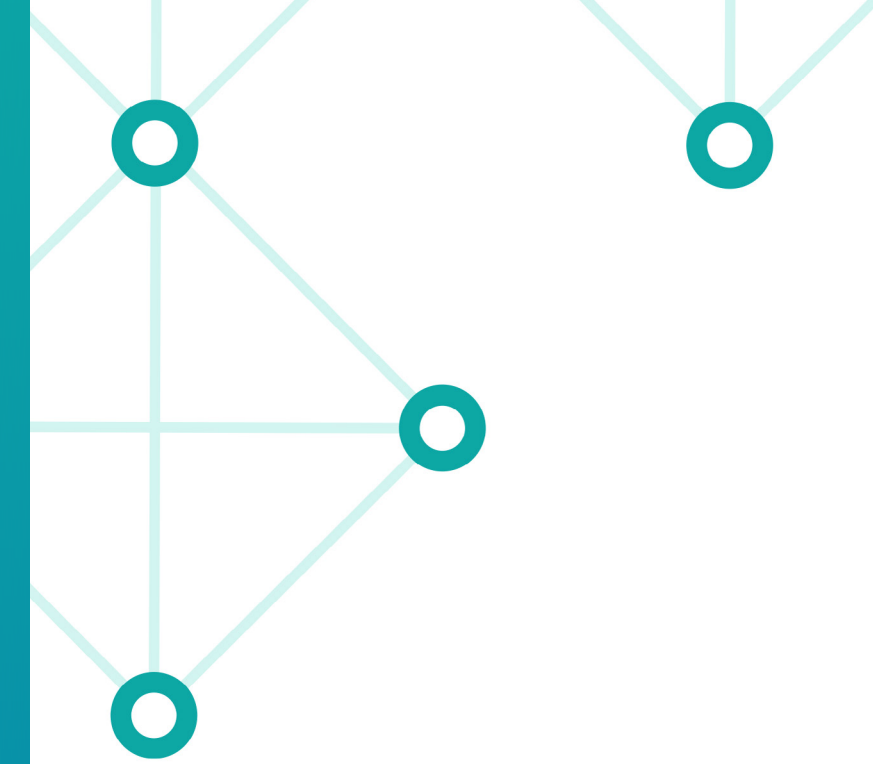
The CQRS/ES Playbook

Architecture Modernization Strategies
for CxOs, Architects and Development Teams

Allard Buijze & Vijay Nair



AxonIQ



It has been very difficult to ignore the twin patterns that seem to be all over the place of late - "[CQRS/ES](#)" (Command Query Responsibility Separation and Event Sourcing). They have taken a prominent mindshare at Conferences, you see a lot more literature coming out related to these patterns - Books, Blog Posts, Articles, Customer Case Studies and even within internal organizational summits these patterns make a beeline towards the front whenever the topic of Architecture Modernization comes up.

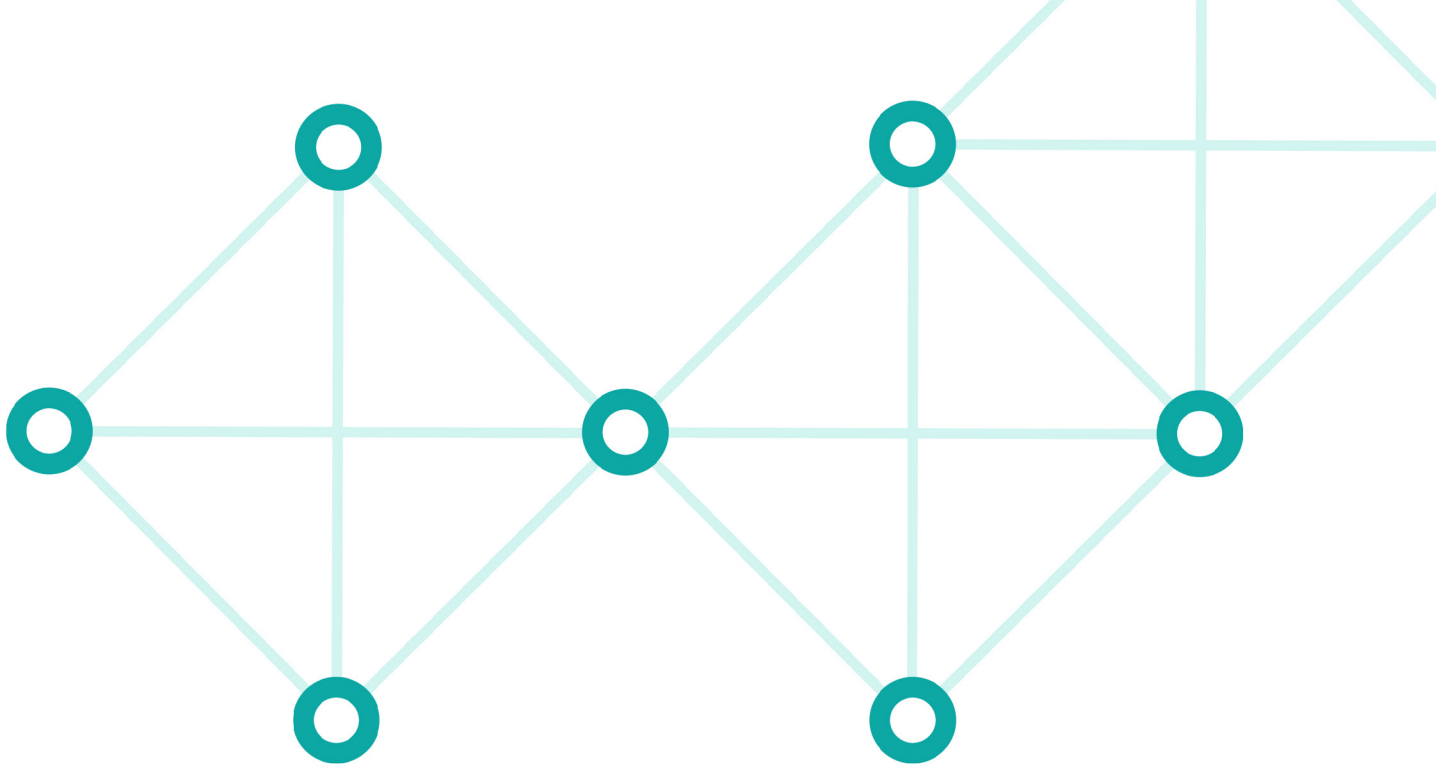
And as is the case with any other technology adoption, there are two paths that are generally taken - On one side, there are the "fence-sitters", organizations that are sufficiently intrigued by the promise of a new technology, have read a ton of literature, done some proof-of-concepts but are a bit hesitant to take the next step. Well, haven't we all been promised the holy grail for software development before? On the other side, you have the "fast-movers", organizations that have a culture of uptake of new technologies, apply it in smaller measures and then gradually expand its adoption by implementing a cycle of optimization (e.g. Best Practices/Lessons Learnt/Developer Experience/Automation).

Of course, not all technology adoption results in a happily ever after story - The adoption of these patterns do go awry and the trough of disillusionment (courtesy Gartner) is real. But are the patterns to blame? Probably not! There could be other factors at play too - misinterpretation of these patterns/poor organizational readiness/choice of a wrong tool to help do the job and so on.

A lot of factors go in to ensure successful adoption of any technology and particularly when you choose to implement something as fundamental as CQRS/ES which has an impact across all three main architectural characteristics - structural/operational and cross-cutting, you need to be extra prepared to get it right.

This paper lays down the proper foundational decisions that needs to be taken across all levels of the organization to help implement CQRS/ES successfully. Starting off with an holistic overview of these patterns, it provides a guided pathway for organizations that fall in both buckets - the fence sitters will understand what needs to be done for taking that leap of faith, while the fast movers will understand what needs to be done to optimize the adoption as well as possibly getting a derailed project back on track to successfully recoup the investment.

Let's begin.



So What Exactly is CQRS/ES?

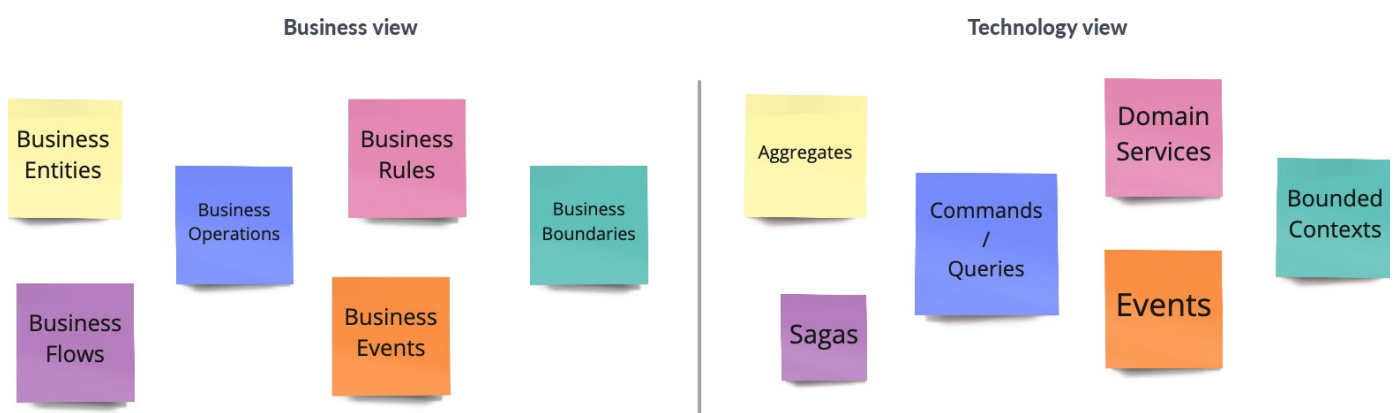
Before we deep dive into it, a bit of history. The genesis of these patterns can be traced back to [DDD or Domain Driven Design](#) as we all know it. In a nutshell, DDD aims to provide answers to two fundamental questions that have plagued the software development field for a long time.

- How do we describe any Business Domain (e.g. Banking/Retail/Hospitality) in a language that is understandable across the Enterprise – Business and Technology?
- How do we decompose software systems into cohesive modules (or components and microservices) that are loosely coupled?

DDD provides a systematic approach to Application Design by providing a set of strategic and tactical patterns to breakdown the complexity associated with Business Domains. DDD advocates the breaking down of any Business Domain into a set of problems that you would like to solve for the Domain (e.g. in Retail Banking you have to solve the problem of opening Checking Accounts). DDD helps teams then come up with something known as a “Domain Model” that can be

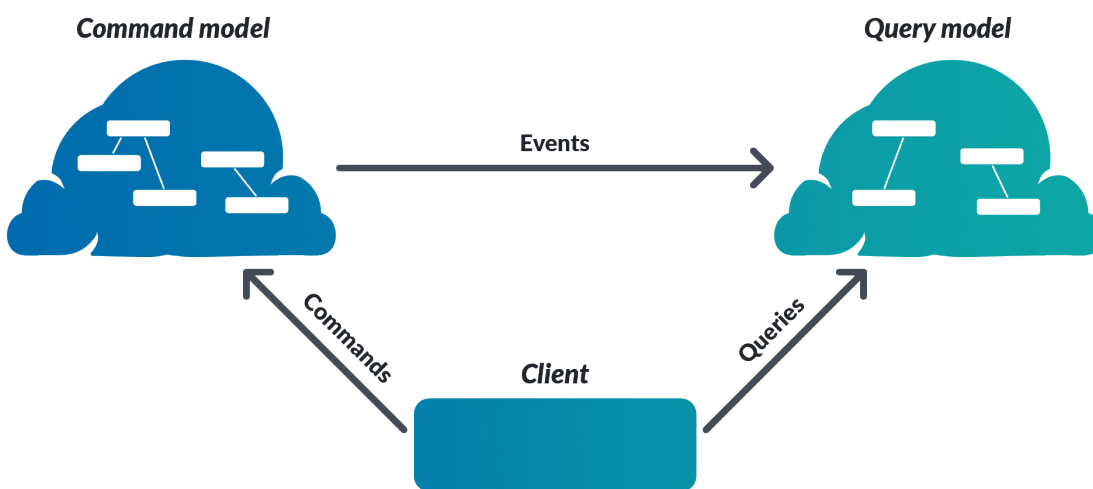
applied to solve the problem. The Domain Model is an artifact that teams could use to communicate a common language as well as be a unit of deployment (e.g. as a [Microservice](#)) for that specific problem.

However over a period of time as the exploration of the domain started expanding it posed a challenging problem - that of Model Complexity. The model started to become big and unwieldy which resulted in it becoming difficult to maintain, “Model Fatigue” started becoming a real thing and finally different parts of the model started to have different non-functional requirements (e.g. scalability).



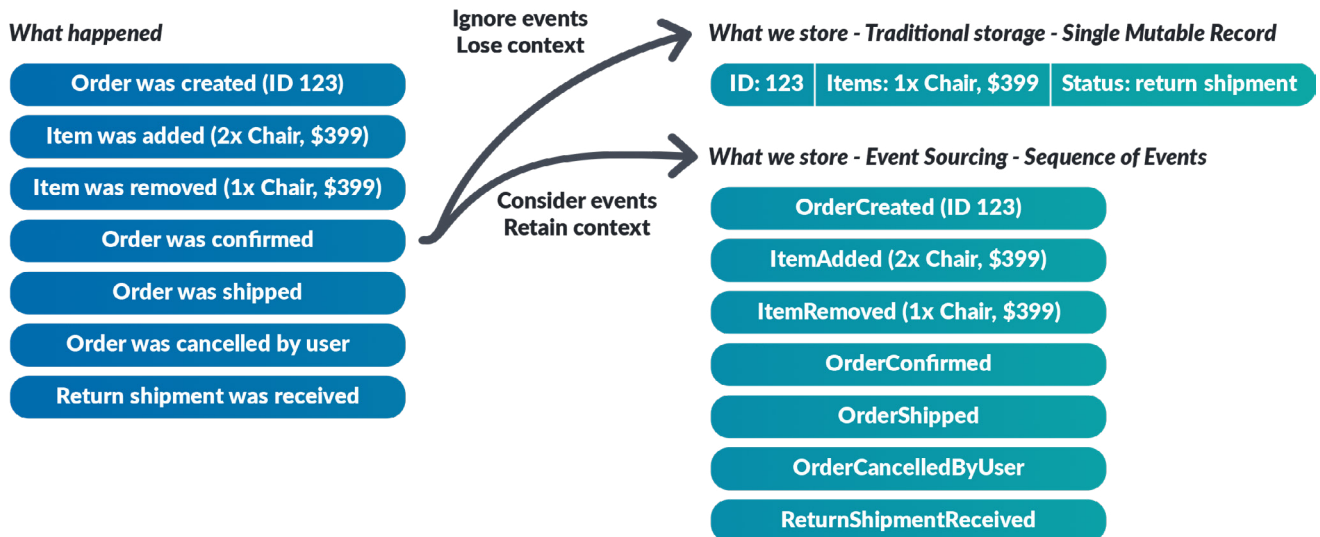
Enter CQRS - [CQRS or Command Query Responsibility Separation](#) was proposed with the primary aim to tackle Model Complexity by dividing and conquering the Domain Model. Simply put, it advocates the vertical split of your Domain Model into two distinct partitions - a Command Model and a Query Model. The Command Model is focused on executing Tasks, is primarily expressed in operations (e.g. Open an Account) and only contains the data necessary for task execution and decision making. The Query Model (aka Projections) is focused on delivering information with data stored the way it is used (e.g. Relational/NoSQL). Suddenly the complexity associated with bloated Domain Models disappear due to the optimization that CQRS provides. The split also helps you focus on different types of NFRs for each side of the Domain Model e.g. certain problems might have more reads than writes while certain problems might be the exact opposite - CQRS offers a great way to help address these kind of problems.

However, the adoption of CQRS does present one problem that needs to be taken care of. How do we keep the two models in sync i.e. changes in the Command Model should (eventually) be reflected in the Query Model? An obvious answer would be “Events” which could act as the glue between the Command Model and the Query Model and since Events are the result of a task (e.g. Open an Account results in an Account Opened Event), they are tied to the Command Model.



Which brings us to the final piece of the puzzle - Event Sourcing which deals with the concept of immutable data. Essentially ES proposes to exclusively store the state of your model as a series of immutable events in a datastore rather than as a mutable set of records which is the more traditional way. Event Sourcing has a natural alignment to CQRS. Commands typically result in the Events that need to be stored and the same Events are also consumed to help in construction of various projections of the state that can then be served by Queries. In short, Event Sourcing without CQRS is quite difficult.

A side by side comparison of Event Sourcing Based Storage v/s Traditional Storage is shown below. As an example we have taken the lifecycle of an order being placed, confirmed, shipped and finally cancelled by the user. While in the Traditional Storage mechanism we store just one record which reflects the current state of the order, in an Event Sourcing based storage mechanism we store it as a series of immutable Events. Suddenly you have complete context of why your system is in a particular state



because you have the complete sequence of events that led to it - An extremely powerful feature and the fundamental selling point of Event Sourcing.

A summary of the benefits of Event Sourcing:

- Immutability
- Reliable source of truth
- Naturalized Audit Trail
- Data Mining/Analytics
- Design Flexibility
- Temporal Reporting
- Easier to Debug
- Enhanced User Experience
- Event Driven Distributed Applications (i.e. Microservices)
- Reactive and Scalable
- Predictable Development Model

To summarize, the CQRS/ES twin patterns help build a completely new class of applications which are inherently Reactive, Scalable, Distributed and Event Driven - Essentially they become the foundational patterns for your Organization's Architectural Modernization effort.



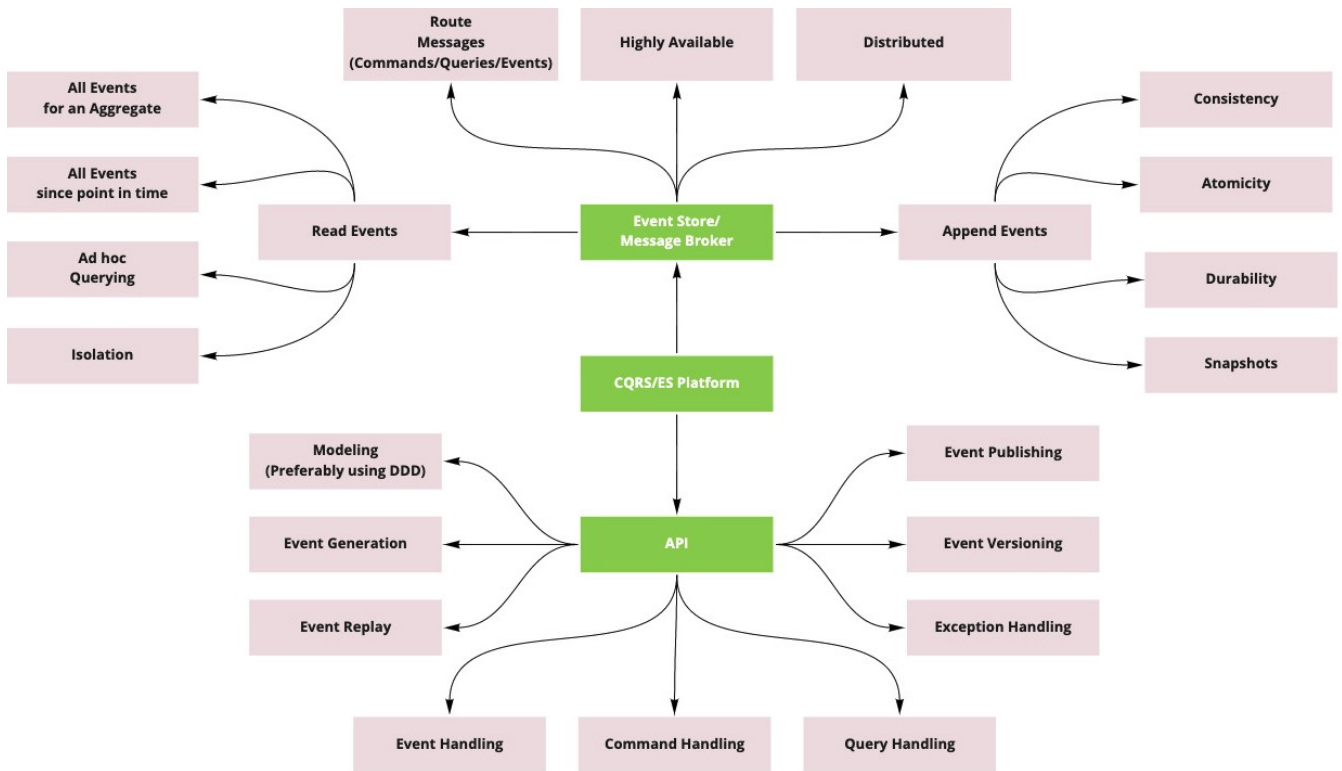
Rolling out a CQRS/ ES Platform within your organization

While the concepts are easy to understand and the benefits of adoption quite obvious, a big question is how do you roll this out within your organization? Before we layout the process, let us talk about something that we have coined here at AxonIQ called the CQRS/ES Capability Map.

The Capability Map details out the essential features that are required for rolling out an enterprise grade CQRS/ES infrastructure across your application portfolio wanting to adopt these new paradigms.

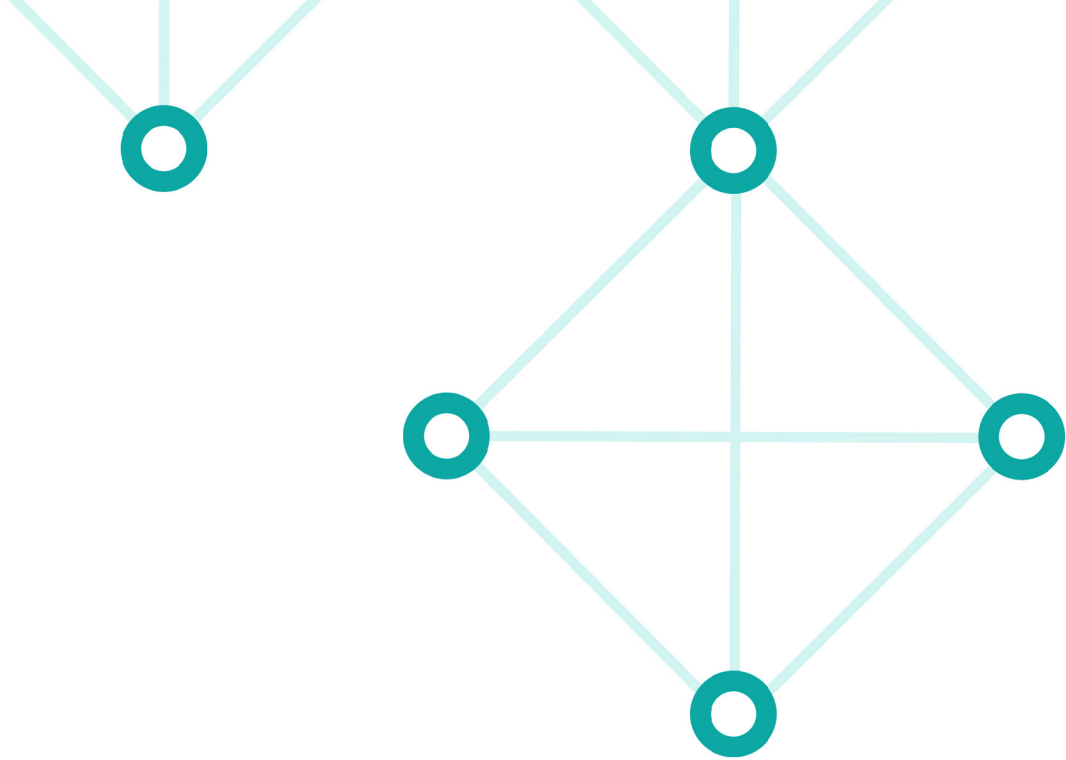
The Capability Map consists of two main areas:

- A logical infrastructure i.e. an API to support CQRS/ES operations preferably using DDD concepts.
- A physical infrastructure to support Event Persistence/Retrieval and Message Routing (for Commands/Queries and Events).



The Map guides you in figuring out the various components that would be needed to be put in place to roll out the required infrastructure. A very important point that needs to be considered is the incorporation and implementation of a strong modeling and governance process as a complimentary aspect to the CQRS/ES component set. Event Storming/Event Modeling/Storyboard Mapping are good examples of modeling processes that work very well with the CQRS/ES design paradigms.

To summarize, rolling out a CQRS/ES platform requires the implementation of a set of Logical/Physical Infrastructural components complimented by a robust modeling/governance process.



The Market for Purpose-Built CQRS/ES Platforms

Why a Purpose-Built platform?

As can be seen above, implementing the infrastructure CQRS/ES requires is complex. While the first tendency often is to build out these patterns utilizing custom built frameworks and tools, it quickly spirals out of control with a very high chance of failure as the various aspects of these patterns start to become clearer. This in turn translates into a messy architecture, poor developer experience, complex operational aspects resulting in a high monetary cost for organizations without much to show for the effort spent.

A wiser decision is to always utilize a purpose-built CQRS/ES platform. This helps accelerate the adoption of these patterns quickly and effectively by providing a complete ecosystem of tools, single infrastructure and techniques.

This enables technology teams to adopt these patterns rapidly:

- Architects can implement a streamlined/enforceable architecture.
- Developers can focus on what they do best i.e. writing business code.
- Operators have a single physical infrastructure to monitor, optimize and automate.
- CxOs can realize significant savings while at the same time showcase real architecture modernization progress.

The Axon Platform

Axon is the leading purpose-built CQRS/ES platform to help rollout enterprise grade CQRS/ES infrastructures. Axon elevates the concept of traditional CQRS/Event Sourcing architectures by treating every operation within an application (Commands/Queries and Events) as “Messages”. Axon processes these Messages using the Location Transparency Pattern, enabling teams to focus on getting right the correct boundaries for their applications, allowing them to be split into microservices as the requirements evolve.

The Axon Platform provides two main components:

- **[Axon Server \(Physical Infrastructure\)](#)**
A highly scalable, distributed, and purpose-built Event Store and zero-configuration Message Router. It is available in two editions Standard and Enterprise (for more advanced capabilities like Clustering).
- **[Axon Framework \(Logical Infrastructure\)](#)**
Implements the full range of API capabilities required for Event Sourcing/Message Routing operations. It provides the building blocks required to deal with all the non-functional requirements, allowing developers to focus on the functional aspects of their application instead.

AxonFramework

AxonServer



Event Driven
Microservices



DDD



CQRS



Event Sourcing



Routing



Event Store



Observability



High
Availability

[The Axon Platform](#) has multiple customers across a wide range of industries who use it to roll out an Enterprise-Grade CQRS/ES infrastructure. An important aspect here is that the Axon Platform is an “Application Platform” not a “Deployment Platform”. Applications built with the Axon Platform can be deployed on any existing cloud infrastructure (e.g. [K8s](#)/[OpenShift](#)/[Tanzu](#)) that organizations currently have invested in.

Conclusion

To conclude, the CQRS/ES patterns can serve as the foundation for your architecture modernization efforts within your organization. The adoption of these patterns requires implementing [a set of capabilities](#) that these patterns prescribe. Utilizing a purpose-built CQRS/ES platform like Axon helps organizations reap the significant benefits that these patterns offer in a faster/cost optimized/streamlined and efficient way.

