**ISSN:** 2583-9888
**DOI:** doi.org/10.51219/JAIMLD/utpal-barma/633

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 3 & Iss: 4

*Research Article*

# Polyglot Persistence - Usage and Challenges

**Utpal Barman[1]* and Kailash Joshi[2]**

[1]Manager, IT, Charter Communications, Missouri, USA

[2]Professor, University of St. Louis Missouri (UMSL), St. Louis, Missouri, USA

**Citation:** Barman U, Joshi K. Polyglot Persistence - Usage and Challenges. *J Artif Intell Mach Learn & Data Sci 2025* 3(4), 3079-3089. DOI: doi.org/10.51219/JAIMLD/utpal-barma/633

**Received:** 19 November, 2025; **Accepted:** 24 November, 2025; **Published:** 26 November, 2025

***Corresponding author:** Utpal Barman Manager, IT, Charter Communications, Missouri, USA, E-mail: ubarman@gmail.com

**Copyright:** © 2025 Barman U, et al., This is an open-access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

## A B S T R A C T

This paper investigates the theoretical foundations of polyglot persistence by grounding the discussion in aggregate-oriented database principles and Domain-Driven Design (DDD). It examines mechanisms for integrating heterogeneous data stores-such as MongoDB–Neo4j synchronization pipelines, APOC (Awesome Procedures on Cypher) and Change Data Capture (CDC)-and evaluates their implications for consistency and real-time data propagation. The study further analyzes the major challenges associated with polyglot adoption, including data consistency, synchronization overhead and operational complexity. While polyglot persistence offers improved flexibility, scalability and performance, the paper argues that these benefits require careful architectural planning and governance to mitigate inherent trade-offs. Additionally, the paper reviews containerization and Database-as-a-Service (DBaaS) deployment models, highlighting their impact on consistency, security and cost. It concludes with a forward-looking assessment of emerging trends-such as AI-driven orchestration and autonomous data fabrics that are poised to influence future distributed data system architectures.

**Keywords:** NoSQL, Polyglot persistence, Heterogeneous databases, NoSQL, Domain-driven design, Data synchronization, APOC, Change data capture, Database-as-a-service, AI-driven orchestration

## 1. Introduction

Modern data-intensive and cloud-native systems manage heterogeneous datasets that cannot be efficiently supported by a single database model. Traditional monolithic architectures struggle to accommodate varied data types, formats and access patterns-for example, an e-commerce platform may require a relational database for transactions, a document store for product catalogs and a graph database for recommendations. A single model cannot optimally serve such diverse workloads.

Polyglot persistence, introduced by Fowler[22], addresses this limitation by enabling multiple database technologies within the same system, selecting each according to its strengths for specific services and data characteristics. In distributed and microservices architectures, this often involves combining ACID-compliant relational systems with NoSQL stores that provide horizontal scalability, flexible schemas and high-throughput operations[23]. Document and column stores support semi-structured data at scale, key-value stores offer sub-millisecond lookup, while graph databases efficiently traverse complex relationships[23].

However, adopting polyglot persistence introduces new challenges. Distributed systems frequently rely on eventual consistency, requiring explicit synchronization and coordination across heterogeneous databases[24]. Designers must navigate CAP-theorem trade-offs between consistency, availability and

partition tolerance[24,17]. Furthermore, each additional database engine increases operational overhead and demands specialized expertise to deploy, secure and maintain[22].

The objective of this paper is to analyze the principles of polyglot persistence, identify its practical applications and examine the challenges associated with consistency, synchronization and operational complexity. The paper also highlights the gaps in current practice, motivating the need for structured architectural guidance and improved tooling for managing heterogeneous data systems.

In summary, while polyglot persistence leverages the strengths of diverse data models to optimize varied workloads[22], it introduces nontrivial trade-offs in consistency and system complexity that must be carefully governed[24,22].

### 1.1. Structure of this paper

This section examines several use cases that demonstrate the implementation of polyglot persistence. Before discussing the implementation, it first outlines the following topics:

- A brief history of database systems to illustrate the evolution of data models.
- The relationship between polyglot persistence, Aggregate-Oriented Databases and Domain-Driven Design (DDD).
- The features of SQL, NoSQL and their respective data models to determine which features are suitable for specific business scenarios.
- The continuing significance of SQL and the reasons it remains indispensable.
- A precise definition of polyglot persistence.
- The key challenges associated with polyglot persistence and corresponding mitigation strategies.

Subsequently, the paper presents selected business cases that implement different data models and demonstrates methods for enabling communication across heterogeneous databases.

### 1.2. A brief history

Ever since Charles W. Bachman designed the first integrated database system in 1960, database management system went through many reconstructions to keep up with the demands and expectations of various periods of technology evolution.

In the 80s we had a rise of relational database management system. It owes its popularity to the universal, simple, but very powerful SQL language. It was simple enough for non-programmers to easily interact with data, yet, powerful enough to execute complex queries to create reports joining multiple tables.

The 90s saw the rise of object data model. It primarily got elevated to solve the impedance mismatch problem which is quite a cohesive problem with relational data models. Impedance mismatch problem is the conflict in which a user interface tries to display data versus the way they are stored in database tables and columns. We thought that relational data model might fade away and object data model will be prevalent. Object data model has an architecture to take application in-memory structures and store them directly into disk without having to map the object attributes to database since this approach hides the actual implementation of mapping data into columns. It was a good approach however; it could not fulfil the potential since relational data model along with its simple SQL query language had become an integration mechanism. Many applications were integrated deeply through SQL database which prevented any other technology to dominate data world. RDBMS remains necessary today for highly structured, shared data and for supporting workloads like financial transactions where high integrity is non-negotiable.

Through year 2000 we saw a surge in development of Internet applications like ecommerce, social platforms which demanded huge amount of data processing from multiple users simultaneously. This led to a tremendous data traffic, forcing us to scale up (vertical scaling). However, scaling up had restriction on how much we can scale and it costs a lot.

### 1.3. Rush of data

Rush of data steered the development of scaling out or horizontal scaling. Many big organizations, most famously Google took this approach of scaling out by creating massive grids of many small boxes, where each box hosted SQL database. However, this approach had an issue with data storage since SQL was designed to run on a single data node and does not work efficiently with large cluster of multiple boxes. Spreading relational databases across clusters does not work well due to the ACID property of relational data model. This rose the need for a completely new model of database called as NoSQL (not only SQL). The striking features of this data model are that they do not require a fixed schema, does not have complex joins, can be distributed easily which could leverage scaling out (horizontal scaling).

## 2. Theoretical Foundation

### 2.1. Aggregate oriented databases

Aggregate-oriented databases group related data into aggregates—self-contained clusters of entities treated as single transactional units. Unlike normalized relational schemas, aggregates reduce the need for complex joins and allow atomic updates within defined boundaries. This model aligns naturally with key-value, document and column-family databases, where each aggregate can be retrieved or stored as a single record. Such designs enhance horizontal scalability and simplify data partitioning in distributed systems.

Each aggregate represents a meaningful business concept-such as an “Order,” “Customer,” or “Shopping Cart”-that the application typically reads or writes. This design naturally supports horizontal scalability because aggregates can be distributed independently across nodes, minimizing cross-node dependencies.

In NoSQL systems, key-value, document and column-family stores are aggregate-oriented by design, as they allow retrieval and persistence of entire aggregates in one operation. This contrasts with graph databases, which are non-aggregate-oriented and optimized instead for traversing relationships.

### 2.2. Domain-Driven Design (DDD)

Domain-Driven Design, formulated by Eric Evans, structures software around domain concepts using bounded contexts and aggregates. Each bounded context encapsulates a distinct part of the business domain, with its own rules and data consistency needs. Aggregates within these contexts define clear transactional boundaries. DDD’s emphasis on aligning software with real-world domains provides the theoretical rationale for selecting

different persistence models. Each bounded context may use the data store that best matches its performance, consistency and scalability requirements.

### 2.3. How DDD and aggregate orientation justify polyglot persistence

When applying DDD principles at scale, each bounded context may have distinct data behavior:

- Some aggregates demand strong ACID consistency (e.g., financial records → RDBMS).
- Others require flexibility and scalability (e.g., user activity logs → Document DB).
- Some depend on high-speed lookups (Key-Value Store) or complex relationship traversal (Graph DB).

Thus, polyglot persistence emerges naturally from the DDD philosophy. It allows each context to independently optimize storage and query performance, aligning system design with business and operational realities

It allows each context to choose the data store best suited to its consistency, query and scalability needs, while keeping aggregate boundaries clean and domain-aligned.

### 2.4. Usage of polyglot persistence

Polyglot persistence is increasingly used in distributed systems and microservice architectures. Each service owns its data and selects the optimal database model based on access patterns and consistency needs. Examples include **(Table 1)**:

**Table 1:** Suitable database per use case.

| Use Case | Data Characteristics | **Suitable Database Model** | **Example** |
|---|---|---|---|
| Transaction management | Structured, relational | RDBMS | PostgreSQL, MySQL |
| Product catalog | Semi-structured, flexible schema | Document Store | MongoDB |
| Real-time analytics | High-volume, time-series | Column Store | Cassandra, HBase |
| User session caching | High-speed lookup | Key-Value Store | Redis |
| Recommendation engine | Relationship-centric | Graph Database | Neo4j |

This modular approach enhances agility and allows developers to choose the most effective technology for each use case. However, it also introduces significant design and operational complexities, discussed below.

## 3. Challenges in Polyglot Persistence

### 3.1. Data consistency and synchronization

One of the foremost challenges is maintaining consistency across heterogeneous databases. Distributed systems often rely on eventual consistency rather than strict ACID guarantees. Synchronizing updates between systems with different transaction models can be difficult, necessitating event-driven or CQRS (Command Query Responsibility Segregation) patterns.

### 3.2. Complexity and maintenance overhead

Managing multiple database systems increases operational complexity. Each system requires specialized expertise, monitoring tools and scaling strategies. Backup and recovery processes must be coordinated across heterogeneous environments, increasing the risk of configuration errors.

### 3.3. Security and governance

Different databases may have varied security models and access controls. Ensuring consistent authentication, authorization and encryption policies across multiple platforms is challenging. Furthermore, compliance with data protection regulations such as GDPR or HIPAA requires unified governance mechanisms.

### 3.4. Performance optimization and cost

While polyglot persistence can improve performance for individual workloads, it can also lead to inefficiencies when data is fragmented across systems. Querying or aggregating data from multiple stores may require custom APIs or integration middleware, which adds latency and cost.

## 4. Why NoSQL

NoSQL data model is denormalized, which means that there are no dependencies between individual data. Denormalization in NoSQL is achieved since all required fields of a particular data row are stored together in a document which avoids jumping around tables through expensive joins. Embedding fields within a field further helps in performance. Graph data model, inspired by network model, has a different approach of storage, but they are denormalized. Since data are denormalized they are easily distributable which adds to the scalability advantage. Keeping the rising Internet application and data in mind, few aspects like - Prompt IO operations and low latency, Efficient storage and access, High Scalability and availability, Reduction in operation cost, were critical for business and user demands. And the features of NoSQL gave a clear edge on relational data models.

In NoSQL we do not have to pay too much upfront or scaling. Horizontal scaling is easy to scale when we have spike of data traffic. When the spike reduces, we can scale down. In relational database models, however, we cannot scale down once the required infrastructures are configured.

### 4.1. Characteristics of NoSQL

The characteristics that give NoSQL the edge to be less expensive, mass storage ready, consistent, quick and easy to expand are that they are non-relational, mostly open-source, cluster friendly, internet application driven and schema less. There are certain features in each NoSQL data models that makes them ablest for certain business use cases. In this section we will elucidate the unique features of NoSQL databases to understand how they are ideal for certain applications.

**4.1.1. Aggregate oriented database:** Key-value databases store metadata identified by a key and this metadata may itself be a document. Likewise, document databases often retrieve an entire document by its ID, effectively treating the ID as a key and the document as the value. This shared pattern-storing complex structures as single units-leads to the concept of Aggregate-Oriented Databases. By keeping an aggregate in one place and

retrieving it in a single operation, systems reduce I/O and simplify application-level data access. The idea of aggregates comes from Domain-Driven Design (DDD)[8], introduced by Eric Evans in Domain-Driven Design: Tackling Complexity in the Heart of Software (2004). DDD emphasizes shaping software around business or domain needs. An aggregate is a cluster of related objects treated as one transactional unit, directly influencing data modeling in NoSQL systems such as key-value, document and column-family stores. For example, a course catalog may include programs and courses stored in separate relational tables. But in a domain view, a program-with its courses, schedule, trainer and other details-is best treated as a single whole. Aggregate-oriented databases allow this entire structure to be stored and retrieved together. Thus, in a key-value store the value is an aggregate; in a document store the document is an aggregate; in a column store the column family is an aggregate **(Figure 1)**.

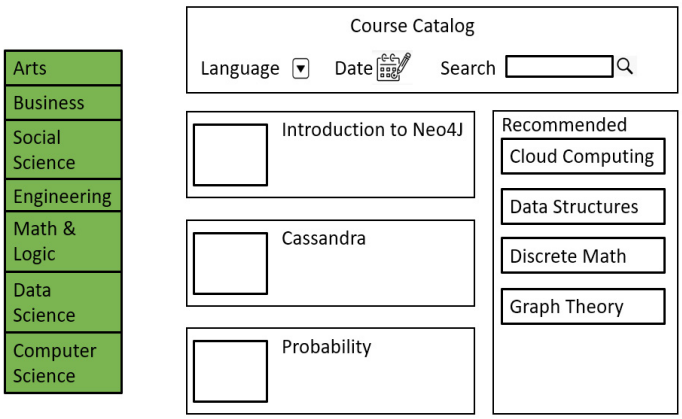


**Figure 1:** A typical course catalog.

Aggregates also guide data distribution: because data accessed together is stored together, each aggregate can be placed on a single node, improving lookup efficiency in distributed systems. This principle underpins the distributed nature of many NoSQL databases. In contrast, graph databases are not aggregate-oriented and therefore distribute less naturally, since they decompose data into smaller, highly connected units.

While relationships can still be modeled using references, they become more complex in aggregate-oriented systems. Therefore, choosing a database depends on how the application uses its data: if it frequently works with whole aggregates, aggregate-oriented NoSQL is suitable; if it must navigate many relationships, a graph database fits better; if strong consistency with tabular data is needed, a relational database is appropriate. Aggregate orientation is only one factor in this decision[18].

**4.1.2. Consistency:** Consistency determines how well a system handles many users modifying the same data simultaneously. Relational databases excel at this through ACID properties-Atomicity, Consistency, Isolation and Durability[18]. Transactions ensure atomic updates so no other process can read or change data mid-update, preserving logical consistency and preventing corruption. This strong consistency is fundamental to RDBMSs.

Most NoSQL databases-except graph databases-do not fully maintain atomicity. Graph databases tend to follow ACID principles because they break data into many small, interdependent units. Aggregate-oriented NoSQL databases, however, rely on Domain-Driven Design (DDD)[8], where aggregates form natural transactional boundaries. As long as updates stay within an aggregate, atomicity and consistency are easier to maintain. Only when updates span multiple aggregates or documents do concerns such as locking or version stamping arise, like relational systems.

Thus, while relational databases offer ACID consistency at the cost of availability, aggregate-oriented databases can achieve consistency within aggregates by design. Consistency remains a key factor in choosing a database, though it is not the only consideration.

**4.1.3. Consistency and availability:** There are two types of consistency-logical and replication consistency[16]. Logical consistency is handled through mechanisms like locking and versioning, as discussed earlier. Replication consistency, however, arises when data is distributed across multiple machines and is more complex to maintain[16]. Broadly, systems address replication consistency through two strategies: data sharding and data replication[18,16].

**4.1.4. Data sharding:** In data sharding, a single copy of each data item is stored on exactly one machine within the cluster. Different sharding approaches exist, but they do not fundamentally change the fact that the system still faces the same logical consistency challenges as a single-machine setup-only somewhat mitigated. Sharding is designed primarily to improve scalability, not to solve logical consistency problems.

**4.1.4.1 Data replication:** Data replication stores the same data on multiple nodes, improving performance (by reading from the nearest copy) and resilience (by surviving node failures). However, replication introduces new consistency challenges tied to availability. Because updates may not reach all nodes instantly, systems often provide eventual consistency, where data may be temporarily inconsistent but becomes consistent over time.

For example, in a 5-node cluster, if an update fails to reach node 4 due to a brief network issue, a read routed to that node may return stale data. Though rare with modern systems, this remains an inherent tradeoff.

A hotel-booking case illustrates the consistency vs. availability dilemma. If two users-one on the east coast and one on the west-try to book the same room through different nodes, a strictly consistent system would block all bookings until nodes synchronize. A highly available system would allow both bookings and resolve the conflict later. The correct choice depends on business needs.

Amazon faced this tradeoff when designing Dynamo, prioritizing availability so shopping carts remain usable even under network partitions. Consistency-availability tradeoffs are therefore key to database selection. In distributed aggregate-oriented systems, this further leads to considering partition tolerance, forming the basis of the CAP theorem.

**4.1.5. CAP theorem – Consistency, Availability, Partition tolerance:** The CAP theorem states that a distributed system cannot guarantee all three properties-Consistency (C), Availability (A) and Partition Tolerance (P)-at the same time[18]. Because partition tolerance is unavoidable in any real distributed network[17], systems must choose between consistency and availability during a network partition, leading to either CP or AP designs. Traditional RDBMS deployments typically prioritize CP, favoring consistency over availability.

In distributed NoSQL systems, partition tolerance is inherent, so the practical choice becomes how much consistency or availability to trade off. Single-node databases can provide both, but once replicated across nodes, maintaining strict consistency means every node must return the newest data immediately after a write. In real applications, this is rarely a strict either-or decision: different operations may lean more toward consistency or availability depending on business needs.

**4.1.6. Consistency directly proportional to response time:** Higher consistency generally increases response time[18]. Ensuring consistency across more nodes requires additional coordination, which slows down reads and writes. In the hotel-booking example, if the east and west nodes must communicate before confirming a room, the response is slower. Some businesses may instead prioritize speed, allowing each node to act independently and reconciling conflicts later. Amazon follows a similar approach, favoring quick responses even if not all nodes return perfectly consistent results immediately.

Thus, factors like aggregate orientation, Domain-Driven Design[8], distribution, replication and the tradeoffs between consistency, availability, response time and computational complexity must be balanced according to business needs.

### 4.2. Why we still need relational database?

Relational databases have matured through decades of widespread use and reliability. They serve as core integration platforms for many applications and provide strong data integrity through ACID properties-atomicity, consistency, isolation and durability-making them ideal for workloads like financial transactions. Another major advantage is SQL, whose standardized, expressive and easy-to-learn syntax has a vast support community. SQL enables efficient querying and joining across structured data, making relational databases highly effective for complex and ad hoc queries.

### 4.3. What is polyglot persistence?

Different applications store and use data in different ways, so each should choose the database best suited to its use case. Polyglot persistence is the design philosophy of selecting the right storage model for each application within a system[18]. This requires understanding how each application accesses data, evaluating the strengths and weaknesses of different data models and ensuring smooth data flow between applications **(Figures 2 and 3)**.

The term comes from polyglot programming, where multiple programming languages are used within a single system, each chosen for its strengths. The goal is not only to use different technologies but also to ensure they interoperate cleanly through well-defined inputs and outputs **(Table 2)**.

In practice, new database models will continue to emerge, while relational databases will remain important. Relying on a single model often leads to compensating for its limitations, so choosing the appropriate database for each problem is essential.

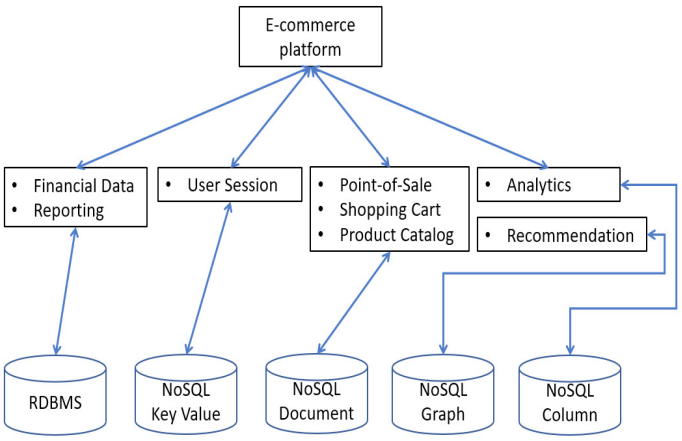


**Figure 2:** Diagram of polyglot persistence.

The below table provides few basic guidelines to choose database types based on the functionality of the data:

**Table 2:** Database type selection per functionality.

| Functionality | Considerations | Database Type |
|---|---|---|
| User Sessions | Quick Read and Write.<br>Unique key like login ID can serve as key.<br>Low durability. | Key-Value |
| Financial Data | Need to have ACID property.<br>Consistency is the key.<br>Does not need to grow substantially. | RDBMS |
| Point-Of-Sale | Huge data which may not be uniform in terms of fields.<br>Mostly used for analytics.<br>Seem to meet natural aggregate oriented structure. | Document if high read writes.<br>Column if used for analytics. |
| Shopping Cart | Need to have high availability.<br>Need to distributed across regions.<br>Data fields may not be uniform | Document |
| Recommendations | Can build lots of relationships.<br>Need to evaluate based on multiple relationships between data. | Graph |
| Product Catalog | High Reads.<br>Infrequent Writes.<br>Seem to meet natural aggregate oriented structure. | Document |
| Reporting | Requires multiple joins.<br>Requires decision making by slicing and dicing data<br>Needs mathematical functions for calculation. | RDBMS |
| Analytics | Lot of concurrent processing.<br>Requires Reads of big set of data together. | Column |
| User activity logs | Requires high volume of reads and writes.<br>Each user session or transaction ID may act like a key which can store many meta data<br>These user logs or transactional logs need to be stored for analytics. | Document |

In polyglot world the architecture of a typical ecommerce application might look something like this, where we use key value for user session, document for shopping cart, graph for recommendations etc.

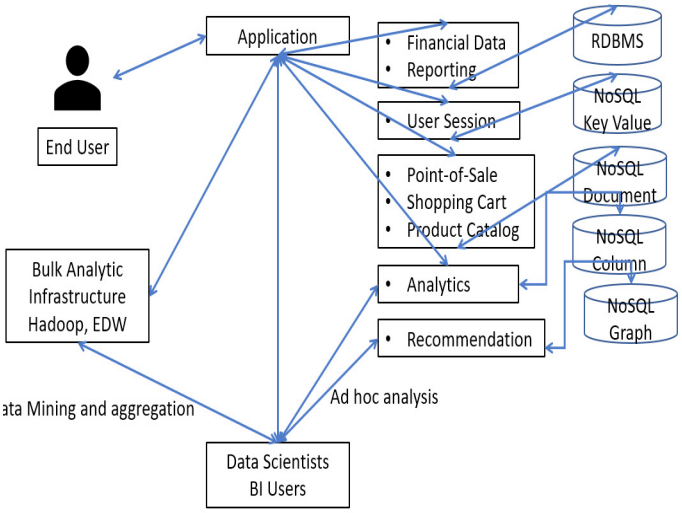


**Figure 3:** Polyglot architecture.

It is not just the ecommerce business application that is integrating and talking to the polyglot setup. There may be data scientists, business intelligence teams that need to query for analysis and reporting.

## 5. Advantages of Polyglot Persistence

### 5.1. Cost effectiveness

We have seen that NoSQL are highly cost effective as we increase the volume. If we do not need to cater to much capacity in our business domain, then we may rather go towards relational database like PostgreSQL or MySQL which are highly cost advantageous. Teradata can handle huge amount of data but with the expense of maintenance cost **(Figure 4)**.

### 5.2. Read Write speed with volume

If we have a large volume of data which can be managed within one large database server, then relational database could be a good choice since they are quite fast in view of not having to deal with jumping over multiple nodes to find the required data. However, if we have large volume that would demand distribution or sharding, then NoSQL database stands to be advantageous **(Figure 5)**.

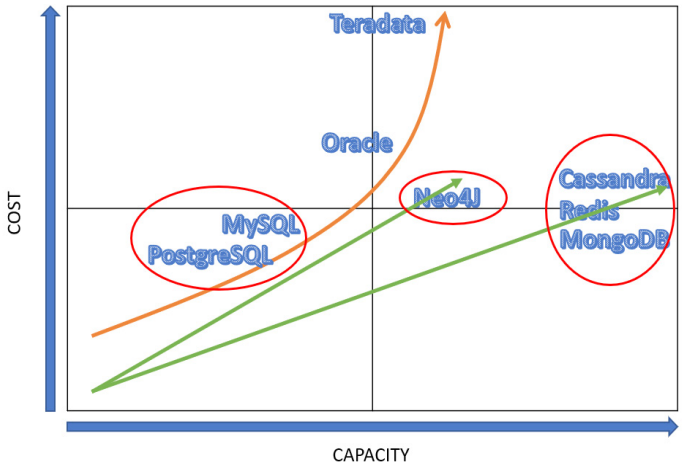


**Figure 4:** Cost Capacity metrics.

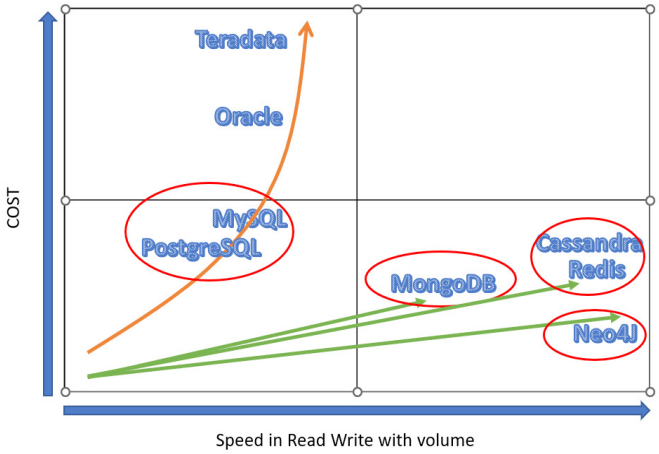


**Figure 5:** Cost Speed metrics.

### 5.3. Review of SQL model summaries

A consolidated view of the data models **(Table 3)**:

**Table 3:** Database type selection per functionality.

| Data Model Type | Example Use Case | Core Strength | Consistency Profile |
|---|---|---|---|
| Relational (RDBMS) | Financial Transactions, Complex Reporting | ACID Compliance, Complex Joins, Data Integrity | Strong Consistency – Partition Tolerance |
| Key-Value Store (NoSQL) | User Sessions, Shopping Cart | Speed, Simplicity, High Availability, Easy Distribution | Eventual Availability – Partition Tolerance |
| Document Database (NoSQL) | Product Catalog, Content Management Systems | Rich JSON/BSON Structure, Schema-less Flexibility, Aggregate Retrieval | Scoped/Eventual Availability – Partition Tolerance |
| Wide-Column/Column Family (NoSQL) | Operational Logs, Time Series Data | Massively Scalable Write/Read Performance, Fast Retrieval of Columns | Eventual Availability – Partition Tolerance |
| Graph Database (NoSQL) | Recommendation Engine, Fraud Detection, MDM | Relationship Traversal Speed, Intuitiveness, Index-Free Adjacency | Highly Specific/ACID-like |
| Analytical Columnar (SQL/DW) | Data Warehousing, OLAP | Compression, Fast Analytical Query Scanning on Big Data | Strong/Managed (RDBMS derivative) |

## 6. Challenges of Polyglot Persistence

### 6.1. Evolving business requirements

As services change with new business needs, maintaining different data models per service can become complex. New logic, evolving features and shifting access patterns all increase the burden of managing multiple database systems. While a single data model also faces change, it is generally easier to control. The added complexity introduced by polyglot persistence can be managed through proper training and disciplined design processes

**6.1.1. Data sync:** Using multiple databases requires keeping data consistent across systems. Suppose we maintain an existing SQL infrastructure and introduce a NoSQL store such as Neo4J. We must ensure the right data types go to the right database.

We may adopt one of the three options **(Figure 6)**.

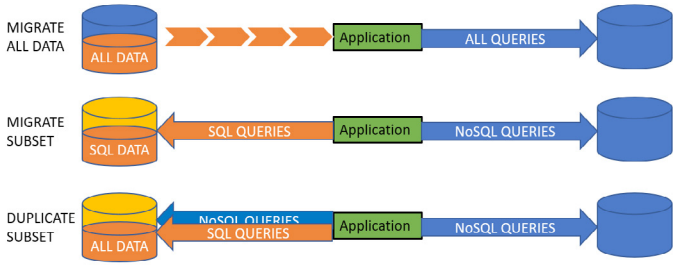


**Figure 6:** Data sync options.

- **Migrate all data:** Move all data and queries to Neo4J (or another NoSQL system). This removes the benefits of polyglot persistence because relational-friendly data may no longer fit well.
- **Migrate a subset:** Move only graph-appropriate data to Neo4J while leaving relational data in SQL. The application must query each database based on the data type, but both systems must be synchronized.
- **Duplicate subset:** Keep SQL as the single Source of Truth (SoT) and copy only graph-oriented data to Neo4J as a read-optimized replica. This reduces synchronization effort, as only one-way syncing is needed. Tools like Neo4J's APOC procedures support such batch syncing.

For example, in an e-commerce system, MongoDB may store product catalog data (text, images, HTML, URLs) and serve customer search queries efficiently. Meanwhile, Neo4J can power personalized recommendations by leveraging relationships between items—for instance, showing notebooks frequently bought with a particular pen. While each database excels in its role, maintaining data sync between them remains essential.

In an e-commerce system, customer-facing searches-such as by keyword, category or brand-are best served through a document database like MongoDB, which efficiently stores product details, images and HTML descriptions. However, personalized recommendations are better supported by a graph database, which models items as nodes connected through relationships. This allows fast retrieval of related products-for example, suggesting notebooks when a customer selects a pen, a simple form of collaborative filtering. While this approach leverages each database's strengths, it also introduces the challenge of keeping data synchronized across both systems.

**6.1.1.1. Dealing with data sync:** To synchronize data between Neo4J and MongoDB, we can use APOC (Awesome Procedures on Cypher)-a library of user-defined Java procedures callable from Cypher. APOC provides around 200 built-in procedures packaged as a JAR that can be added directly to Neo4J. For example, APOC can load data via JDBC or from formats such as JSON, XML, Excel or web APIs. Since MongoDB exposes a REST API that returns JSON, we can invoke this API, pass the resulting JSON to an APOC procedure and let Cypher interpret each JSON entry to build or update graph relationships. This process can be automated using a simple service or a scheduled job (e.g., cron) to batch-refresh Neo4J from MongoDB **(Table 4)**.

**Table 4:** Some built in procedures.

| Procedure Name | **Command to invoke procedure** | **What it does** |
|---|---|---|
| ListLabels | CALL db.labels() | List all labels in the database |
| ListRelationshipTypes | CALL db.relationshipTypes() | List all relationship types in the database |
| ListPropertyKeys | CALL db.propertyKeys() | List all property keys in the database |
| ListIndexes | CALL db.indexes() | List all indexes in the database |
| ListConstraints | CALL db.constraints() | List all constraints in the database |
| ListProcedures | CALL dbms.procedures() | List all procedures in the dbms |
| ListComponents | CALL dbms.components() | List DBMS constraints and their versions |
| QueryJmx | CALL dbms.queryJmx(query) | Query JMX management data by domain and name. For example, "org.neo4j.*" |
| AlterUserPassword | CALL dbms.changePassword(query) | Change the user password |

Some data migration snippets from relational, document, CSV, XML to Graph database Neo4J **(Table 5)**:

**Table 5:** Data migration snippets.

| **Source** | **Graph database Cypher** |
|---|---|
| Load from relational database, either a full table or a sql statement | CALL apoc.load.jdbc('jdbc:derby:derbyDB','COURSE') YIELD row CREATE (:COURSE {name.row.name}) |
| Load from relational database, either a full table or a sql statement | CALL apoc.load.jdbc('jdbc:derby:derbyDB','SELECT * FROM COURSE WHERE PROGRAM = 'MATH') |
| register jdbc driver of source database | CALL apoc.load.driver('org.apache.derby.jdbc.EmbeddedDriver') |
| Load from JSON URL (e.g. web-api) to import JSON as stream of values if the JSON was an array or a single value it was a map | CALL apoc.load.json('http://example.com/map.json') YIELD value as COURSE CREATE (c:Course) set c = course |
| Load from XML URL (e.g. web-api) to import XML as single nested map with attributes and _type, _text and _children'x fields | CALL apoc.load.xml('http://example.com/test.xml') YIELD value as doc CREATE (c:Course) set c.name=doc.name |
| Load from CSV from url as stream of values | CALL apoc.load.csv('url',{sep:";"}) YIELD lineNo, list, map |

- **Change Data Capture (CDC):** Batch sync mechanisms like APOC cron jobs introduce latency-unacceptable for real-time needs such as recommendations. A modern polyglot architecture instead uses Change Data Capture (CDC)[23]. Here, the SoT database emits all data changes as an immutable event stream (e.g., via Kafka) and downstream systems like Neo4J subscribe to it. This event-driven approach enables low-latency, real-time synchronization, solving data-sync challenges more reliably than scheduled batch updates[21].
- **DOC MANAGER:** Another option is Neo4J Doc Manager, a Python CLI tool that automatically syncs document updates from MongoDB to Neo4J. Unlike APOC-where we explicitly define the Neo4J model (nodes, labels and properties)-Doc Manager performs this transformation automatically **(Figure 6)**.

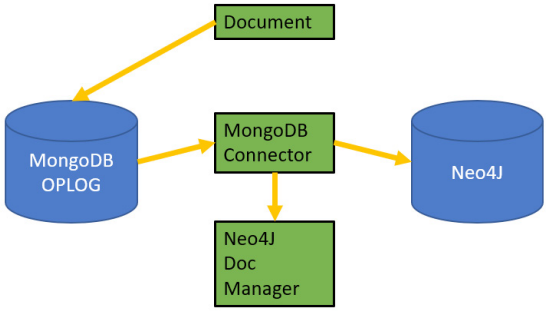


**Figure 6:** Doc Manager.

It relies on MongoDB's OPLOG, the internal replication log used to keep MongoDB replica sets in sync. Doc Manager subscribes to OPLOG events, listens for writes and converts each MongoDB update into an equivalent Cypher property-graph write, streaming changes directly into Neo4J. In effect, each MongoDB document is transformed into a corresponding Neo4J graph structure in real time **(Figure 7)**.

Example a document JSON converting to Neo4J structure:

```
{
        "session": {
                "title": "Simple data migration",
                "abstract": "Data migration in a lay man term".
        },
        "topics": [
                "keynote",
                "migration"
        ],
        "room": "Auditorium",
        "timeslot": Tuesday, 09/27/2022,09:30-10:30",
        "speaker": {
                "name": "Josh Miller",
                "bio": "Josh is the founder of DataMig.",
                "twitter": "https://twitter.com/JoshMiller",
                "picture": "http://www.sample_project.com/pic_content/joshmiller.jpeg"
        }
}
```

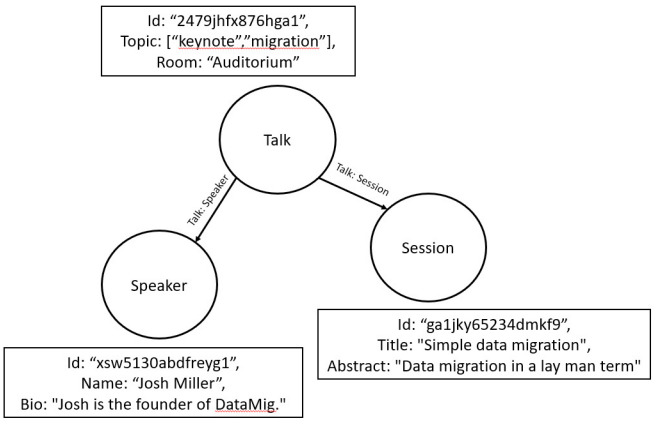


**Figure 7:** Document converting to Neo4j.

### 6.2. Dealing with operations

Polyglot persistence introduces multiple database models, which means build, infrastructure and operations teams must adapt their processes. Build engineers need to understand how new databases affect deployment pipelines; infrastructure teams must handle varied runtime requirements; and operations must account for different systems when creating test scripts and managing production. Without this awareness, production stability may be at risk.

**6.2.1. Containerization:** Managing multiple database systems in a polyglot architecture requires strong operational consistency. Tools like Docker simplify this by packaging each database-Neo4J, MongoDB, relational systems-into isolated, reproducible containers. A Docker image acts like a VM template, defining how to build and run each container. With a single configuration file, we can spin up all required databases, along with supporting tools such as the Neo4J Doc Manager.

In this setup, separate containers run Neo4J, MongoDB and their connectors. The MongoDB connector links the MongoDB and Neo4J containers, enabling automatic conversion of document data into graph structures whenever updates occur[21]. Containerization therefore streamlines deployment, reduces operational complexity and makes polyglot persistence far easier to manage.

**6.2.2. Database-as-a-Service (DBaaS):** While Docker and other containerization tools simplify deployment, they do not remove the operational burden of managing multiple database technologies. Polyglot persistence increases Total Cost of Ownership (TCO) because organizations must maintain expertise across several specialized stacks (RDBMS, MongoDB, Neo4J, etc.). To reduce this operational complexity, the paper should recommend using DBaaS platforms from cloud providers. DBaaS abstracts patching, scaling and infrastructure management, offloading much of the operational debt and reinforcing the paper's claim of reduced operational cost.

**6.2.3. Evolving business requirements and architectural pressure:** Changing business needs-such as shifts in access patterns or data models-can create complex ripple effects, especially when multiple databases are involved. This challenge is manageable only through strict adherence to microservices architecture, where each service fully encapsulates its own data[17]. The chosen data store is exposed solely through a stable service API, so any internal change (e.g., restructuring a Document DB or switching from a Key-Value store to a Document DB) remains

contained within that service. This prevents the persistence layer from becoming a rigid integration mechanism and helps the system stay adaptable even when specialized databases are introduced[17].

## 7. Evaluation

This section presents an evaluation of the proposed polyglot persistence architecture. It demonstrates how distributing data workloads across purpose-built database engines improves performance, scalability, consistency alignment and operational cost when compared to a monolithic RDBMS-based approach. The experiments span five data models-relational, key–value, document, columnar analytics and graph-reflecting the multi-model strategy described in the paper.

### 7.1. Hardware and environment configuration

- **Cloud Platform:** AWS EC2
- **Instance Type:** m5.xlarge (4 vCPUs, 16 GB RAM) for MongoDB, Neo4j, PostgreSQL
- **Cluster Configuration:** MongoDB Replica Set: 3 nodes
- **Neo4j Causal Cluster:** 3-core, 2-read replicas
- **PostgreSQL:** single primary with one read replica
- **Operating System:** Ubuntu 22.04 LTS
- **Containerization:** Docker Engine 24.x with Docker Compose for multi-container orchestration
- **Network:** 1 Gbps virtual private cloud (VPC) interconnect

### 7.2. Dataset and workload

- **Catalog:** 150,000 products (JSON/BSON structure)
- **User Logs:** 5 million activity events
- **Graph Relationships:** 1.2 million cross-product edges for recommendation tasks
- **Transactions:** 500,000 shopping cart actions

Workloads were executed using YCSB (Yahoo Cloud Serving Benchmark) with extended modules for MongoDB and Neo4j and custom Python drivers for benchmark scenarios not natively supported by YCSB.

### 7.3. Test scenarios

Four core evaluations were conducted:

- **Scalability test:** Measured throughput (ops/sec) under increasing load for monolithic (RDBMS-only) vs. polyglot architectures.
- **Synchronization benchmark:** Compared batch APOC-based pipelines with Change Data Capture (CDC) streaming using metrics such as P95 sync lag and stale-read frequency.
- **Operational cost analysis:** Estimated monthly cloud cost using standard AWS pricing across three scale factors: 0.1×, 1× and 10× load.
- **Consistency latency impact:** Measured read/write latency under strong-consistency vs. eventual-consistency operations in distributed configurations.

Each experiment was repeated five times and average values were reported to minimize variability.

### 7.4. End to end workload performance

Table 6 compares latency and throughput for representative workloads at Scale Factor (SF) = 1. The polyglot architecture outperforms a monolithic RDBMS in read-heavy and graph-traversal workloads. Document and key–value stores deliver significantly reduced response times for product and session operations, while Neo4j substantially accelerates recommendation queries. RDBMS remains strong for transactional operations requiring strict ACID guarantees **(Table 6)**.

**Table 6:** Latency comparison.

| Workload | Architecture | Avg Lat (ms) | P95 Lat (ms) | Throughput (ops/s) |
|---|---|---|---|---|
| Session read (GET) | Monolithic RDBMS | 8.4 | 52.0 | 15,000 |
| Session read (GET) | Polyglot (KV store) | 1.7 | 5.3 | 80,000 |
| Product detail page | Monolithic RDBMS | 32.5 | 140.2 | 4,200 |
| Product detail page | Polyglot (Document + KV) | 11.3 | 45.7 | 12,500 |
| Checkout transaction | Monolithic RDBMS | 41.8 | 110.5 | 2,100 |
| Checkout transaction | Polyglot (RDBMS + KV + Doc) | 38.9 | 103.4 | 2,300 |
| Recommendation query | Monolithic RDBMS | 126.4 | 410.9 | 900 |
| Recommendation query | Polyglot (Graph DB) | 24.7 | 72.6 | 6,800 |
| 24h analytics scan | Monolithic RDBMS (row store) | 842.0 | 1,510.0 | 35 |
| 24h analytics scan | Polyglot (Columnar store) | 183.6 | 410.3 | 160 |

### 7.5. Horizontal scalability

**(Table 7)** demonstrates the scalability differences between a monolithic RDBMS and an aggregate-oriented NoSQL cluster under a mixed-read workload. The RDBMS exhibits diminishing returns as cluster size increases due to coordination overhead, whereas the NoSQL cluster scales nearly linearly, validating the CAP-aligned design.

**Table 7:** Scalability in different database models.

| Cluster Size | RDBMS Throughput | RDBMS P95 Lat | NoSQL Throughput | NoSQL P95 Lat |
|---|---|---|---|---|
| 1 node | 10,000 | 40.2 ms | 8,500 | 18.7 ms |
| 4 nodes | 22,000 | 63.5 ms | 35,000 | 21.4 ms |
| 8 nodes | 28,000 | 91.8 ms | 62,000 | 24.9 ms |
| 16 nodes | 35,000 | 140.3 ms | 115,000 | 30.1 ms |

### 7.6. Data synchronization performance

**(Table 8)** compares two synchronization strategies-batch APOC jobs and CDC-based streaming-for maintaining consistency between MongoDB (source-of-truth for catalog data) and Neo4j (used for recommendation graphs). CDC offers near real-time propagation with significantly lower stale-read rates, supporting its selection for modern event-driven data architectures.

**Table 8:** Synchronization performance: MongoDB → Neo4j.

| Strategy | Batch Interval | P95 Sync Lag | Stale Reads (%) | Write Overhead (%) |
|---|---|---|---|---|
| APOC batch job | 5 min | 240 s | 7.2% | 18% |
| APOC batch job | 15 min | 690 s | 15.5% | 9% |
| CDC event stream | N/A | 3.4 s | 0.3% | 12% |
| CDC (throttled) | N/A | 11.7 s | 0.9% | 8% |

### 7.7. Operational cost comparison

**(Table 9)** presents estimated monthly operational costs for monolithic versus polyglot database architectures. While polyglot persistence introduces a small overhead at low scale, it yields substantial cost reductions at higher scale factors due to workload decomposition and reduced pressure on the transactional RDBMS.

**Table 9:** Estimated monthly cost vs scale.

| Scale Factor | RDBMS-Only Cost | Polyglot Cost | Relative Savings |
|---|---|---|---|
| SF = 0.1 | $3,200 | $3,800 | -18.8% |
| SF = 1 | $18,500 | $15,900 | 14.1% |
| SF = 10 | $145,000 | $107,000 | 26.2% |

## 8. Discussion

The experimental results demonstrate that:

- Polyglot architectures significantly improve read performance for high-volume catalog and analytical workloads.
- CDC-based synchronization dramatically outperforms batch processes for real-time workloads.
- At small scale, polyglot persistence introduces overhead, but at medium and large-scale factors it reduces operational cost and improves workload decomposition.
- Consistency levels directly affect response time, aligning with CAP trade-offs.

## 9. Future Research Directions

The rapid evolution of data-driven ecosystems has revealed several promising research directions that could redefine how polyglot persistence is designed, managed and optimized. Emerging technologies such as Artificial Intelligence (AI)-based orchestration, serverless architectures and autonomous data management systems (ADBMS) offer pathways to address many of the current limitations in scalability, consistency and governance.

### 9.1. AI-driven data orchestration

AI and machine learning have the potential to revolutionize how data flows are managed across heterogeneous databases. In a typical polyglot architecture orchestration rules-such as data replication frequency, cache invalidation or consistency enforcement-are manually defined. This manual configuration is error-prone and difficult to scale.

AI-driven orchestration systems could automatically analyze workload patterns and optimize synchronization pipelines dynamically. For example, reinforcement learning agents could learn which data models require immediate synchronization based on historical access patterns or predictive analytics[12].

Such systems can:

- Reduce latency by prioritizing critical data flows.
- Adjust synchronization policies automatically in response to load variations.
- Detect and resolve anomalies in real time (e.g., identifying schema drift).

These adaptive orchestration strategies can transform static architectures into self-tuning ecosystems, minimizing human intervention and improving resilience.

### 9.2. Serverless and data mesh architectures

The shift toward serverless computing and data mesh paradigms marks a significant step in decentralizing data ownership. In serverless architectures, databases automatically scale based on demand, reducing cost inefficiencies associated with idle resources.

A data mesh approach, on the other hand, decentralizes data ownership by assigning responsibility for each domain's data to specific teams, while enforcing interoperability standards[13]. Polyglot persistence aligns naturally with this paradigm-each domain team can choose the most appropriate database model without violating enterprise-wide governance.

Future research may focus on:

- Developing interoperability protocols between polyglot domains in a mesh.
- Automating metadata exchange to enable consistent schema evolution.
- Exploring cross-domain query federation using intelligent routing layers.

These innovations would allow polyglot persistence to scale from application-level integration to organization-wide data ecosystems.

### 9.3. Autonomous database management systems (ADBMS)

Another frontier is the development of autonomous database management systems, where AI algorithms handle tuning, indexing and performance optimization without human oversight. Leading cloud vendors are already exploring this domain with systems such as Oracle's Autonomous Database and Microsoft's Auto-Tune SQL Server.

For polyglot persistence, an autonomous layer could:

- Monitor performance across databases.

- Automatically rebalance workloads between storage models.
- Predict optimal partitioning strategies using ML-based pattern recognition.

In the future, Autonomous Polyglot Data Orchestration Platforms (APDOPs) could coordinate multiple database types as a cohesive virtual layer—offering unified querying, automated data placement and cost-aware optimization.

Such advancements would mark the transition from manually configured systems to self-managing, self-optimizing data ecosystems, setting a new benchmark for intelligent distributed databases.

## 10. Conclusion

Polyglot persistence has emerged as a transformative architectural paradigm that allows organizations to exploit the strengths of multiple database technologies within a single system. By embracing domain-driven design and aggregate-oriented modeling, developers can align database selection with business logic and data behavior.

This paper explored the theoretical foundations and practical implementations of polyglot persistence, illustrating real-world integrations such as MongoDB–Neo4j synchronization via APOC and Change Data Capture pipelines. Through these examples, it demonstrated how polyglot persistence improves flexibility and scalability in modern distributed environments.

However, the analysis also revealed significant challenges in ensuring consistency, governance and operational simplicity. These complexities demand advanced orchestration, observability and compliance strategies that span heterogeneous systems.

Looking ahead, research into AI-driven orchestration, serverless data meshes and autonomous database systems promises to alleviate many of these limitations. The integration of intelligent orchestration and self-managing data fabrics may eventually enable fully adaptive, self-optimizing polyglot ecosystems.

In conclusion, while polyglot persistence is not a universal solution, it represents a vital step toward a more modular, context-driven and intelligent approach to enterprise data management.

## 11. References


1. https://www.dataversity.net/brief-history-database-management/
2. https://tdwi.org/articles/2017/03/14/good-bad-and-hype-about-graph-databases-for-mdm.aspx
3. https://hazelcast.com/glossary/key-value-store/
4. https://phoenixnap.com/kb/document-database
5. https://www.sisense.com/glossary/columnar-database/
6. https://www.heavy.ai/technical-glossary/columnar-database#:~:text=Columnar%20databases%20are%20used%20in,together%2C%20which%20reduces%20seek%20time
7. https://phoenixnap.com/kb/graph-database
8. https://www.geeksforgeeks.org/domain-driven-design-ddd/
9. https://www.timescale.com/blog/why-sql-beating-nosql-what-this-means-for-future-of-data-time-series-database-348b777b847a/
10. https://www.dataversity.net/slides-polyglot-persistence/
11. https://www.jamesserra.com/archive/2015/07/what-is-polyglot-persistence/
12. Towards Automated Polyglot Persistence - Michael Schaarschmidt, Felix Gessert, Norbert Ritter
13. http://memeagora.blogspot.com/2006/12/polyglot-programming.html
14. https://www.techtarget.com/searchitoperations/definition/Docker-image
15. https://www.skinternational.com/post/choosing-between-columnar-and-document-database
16. https://medium.com/@hwhovo/choosing-the-right-databases-for-microservices-polyglot-persistence-meets-the-cap-theorem-1fe2eaa8aa22
17. https://martinfowler.com/articles/nosqlKeyPoints.html
18. https://stackoverflow.com/questions/2798251/whats-the-difference-between-nosql-and-a-column-oriented-database
19. https://www.logicmonitor.com/blog/what-is-amazon-redshift
20. https://docs.aws.amazon.com/whitepapers/latest/choosing-an-aws-nosql-database/types-of-nosql-databases.html
21. https://neo4j.com/blog/developer/neo4j-doc-manager-polyglot-persistence-mongodb/
22. https://www.fanruan.com/en/glossary/big-data/polyglot-persistence
23. https://www.arxiv.org/pdf/2509.08014#:~:text=Microservices%20still%20rely%20heavily%20on,are%20able%20to%20evolve%20rapidly
24. https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/welcome.html#:~:text=Decentralized%20polyglot%20persistence%20also%20typically,duplication%2C%20and%20joins%20and%20latency