aws marketplace

EBOOK

Modern data architectures for cloud-native development

Applying version control, testing, and CI/CD to data changes to break the data monolith and create a unified source of truth

aws marketplace

Table of contents

A modernized database is efficient, agile, and scalable 3
Modern applications 4
What is cloud-native? Why does it matter? 6
Modern data management for cloud-native development 7
Introducing modern application architectures 9
Purpose-built databases: NoSQL 12
DevOps for databases 22
Key takeaways for Database DevOps 25
Implementing modern data management processes and tools 26
Ingesting logs and metrics 36
Recap: Modern data architecture 38
Implement your modern data management strategy today 39
Conclusion 40

A modernized database is efficient, agile, and scalable

Our goals as developers are to deliver business value faster and create products that help us stay competitive in the market. To achieve these goals, we need to modernize by adopting new technologies and practices.

In this chapter, you'll learn the importance of test automation, data management, and data democratization in enterprises, and how to integrate data changes into your Continuous Integration/Continuous Delivery (CI/CD) pipeline.

Modern applications

Modern applications were born out of a necessity to deliver smaller features faster to customers. While this directly addresses only the application architecture aspect, it also forces other teams to build and execute in a similar manner. In order to continuously deliver features, organizations need all cross-functional teams to operate as “One Team.”

Key aspects of modern applications:

- Use independently scalable microservices such as serverless and containers
- Connect through APIs
- Deliver updates continuously
- Adapt quickly to change
- Scale globally
- Are fault tolerant
- Carefully manage state and persistence
- Have security built in

Modern applications require more performance, scale, and availability

Modern applications are pushing boundaries. Users are connected all the time and expect microsecond latency and access from mobile and internet of things (IoT) devices. They also demand that they have the ability to connect from anywhere they happen to be.

e-commerce | Media streaming | Social media | Online gaming | Shared economy

Users	1M+
Data volume	Terabytes–petabytes
Locality	Global
Performance	Microsecond latency
Request rate	Millions per second
Access	Mobile, IoT, devices
Scale	Virtually unlimited
Economics	Pay-as-you-go
Developer access	Instance API access
Development	Apps and storage are decoupled

What is cloud-native? Why does it matter?

Cloud-native is an evolving term. The vast amount of software that's being built today needs a place to run and all the components and processes required to build an application need to fit together and work cohesively as a system.

The Cloud Native Computing Foundation (CNCF) definition states:

Cloud-native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds.

This definition has to broadly apply to everyone, but not everyone has the same capabilities. This is known as the lowest common denominator problem. It is where you try and appeal to a broader group and their capabilities, but in doing so you also need to limit the capabilities that can be leveraged.

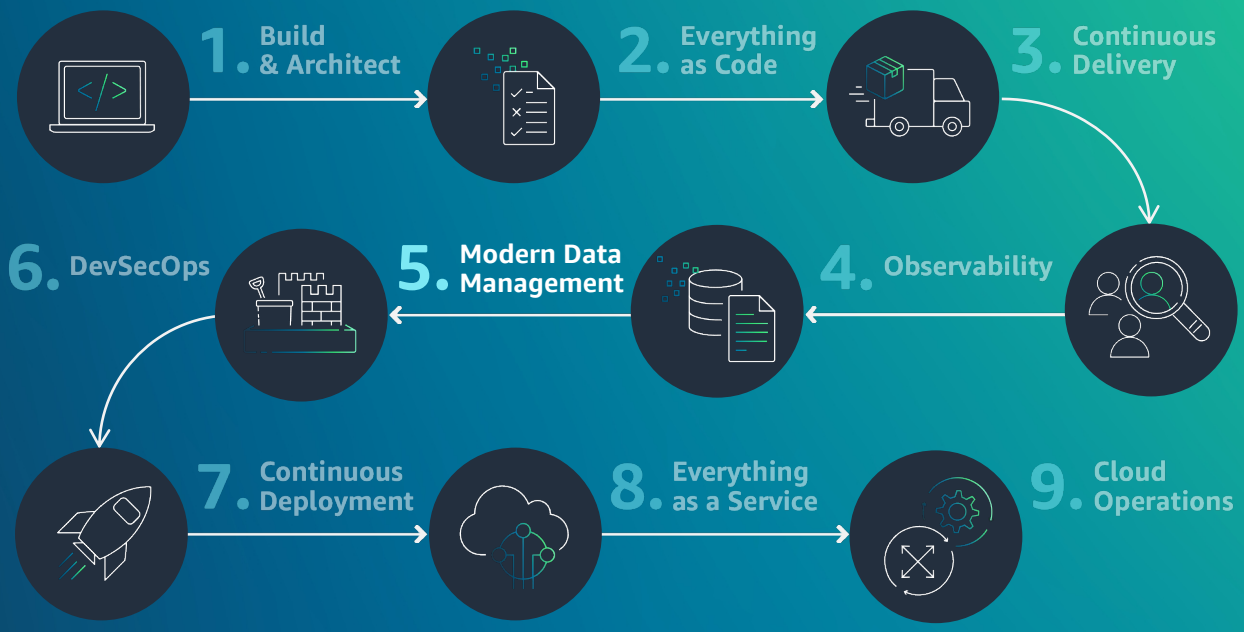
Amazon Web Services (AWS) goes many steps further by providing a broad set of capabilities that belong to a family called serverless. Serverless technologies are more than just AWS Lambda—these services remove the heavy lifting associated with running, managing, and maintaining servers. This lets you focus on core business logic and quickly adding value.

aws marketplace

Modern data management for cloud-native development

As we cover the different capabilities your organization needs to acquire to go fully cloud-native, it's useful to view each one as a step in a journey.

The map below is a model for how organizations typically evolve their cloud-native understanding. As your organization or team moves from stage to stage, capabilities are gained that make releasing new features and functionality faster, better, and cheaper. In the following sections, we'll be focusing on the capability of Modern Data Management.



A look back—traditional three-tier application architecture

Let's first take a look at the traditional architecture model that preceded cloud-native data management.

Historically, applications were built with monolithic, three-tiered web architectures that looked something like this graphic, in which there is:

1. A presentation layer hosted by web servers

Web servers
Presentation layers

2. An application layer in which business logic runs

Application servers
Business logic

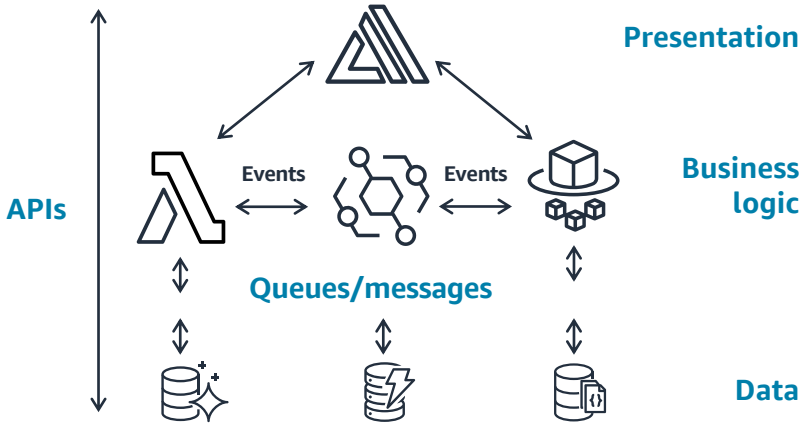
3. A data layer that has database servers

Database servers
Data layer

Note: Some implementations have the application layer horizontally scaled but still bottlenecked at the data layer.

Introducing modern application architectures

Here is an example of a modern application architecture. Yes, it does indeed resemble a three-tier web architecture, but there are some very important differences.



The first major difference in this architecture is that it doesn't reflect the entire application—this is not a monolith, it's a single microservice. The other key differences with this architecture are related to:

Data. How and where you store your data is different in a modern application. In this diagram, we can see multiple data store options. Each database is serving a specific need, meaning it's purpose-built for the task at hand. Additionally, data stores are decoupled.

Application integration and communication. Communication both within the application as well as between each service is different. In modern data architectures, there are fundamentally different approaches to using messaging, events, and APIs within the business.

Compute. The logic you write is important in differentiating your organization. New compute technologies make focusing on business logic easier than ever

Pro tip:

The best tool for a job usually differs by use case so you should build new applications with purpose-built databases. How and where you store your data is different in a modern application. Each database is serving a specific need. This will be covered in depth later.

How AWS customers use microservices

AWS customers run hundreds or even thousands of microservices, an approach that greatly improves their scalability and fault tolerance. Usually, microservices communicate via well-defined APIs, and many customers start the process of refactoring by wrapping their applications with an API.

Decoupling data along with business logic

When it comes to the data requirements of modular services, one size does not fit all. Does the service need massive data volume? High-speed rendering? Data warehousing? AWS customers are considering what they are doing with their data and choosing the datastore that best fits that purpose.

Because the only database choice was a relational database—no matter the shape or function of the data in the application—the data was modeled as relational for decades. Instead of the use case driving the requirements for the database, it was the other way around. The database was driving the data model for the application use case.

Is a relational database purpose-built for a normalized schema and to enforce referential integrity in the database? Absolutely, but the key point here is that not all application data models or use cases match the relational model. Developers are building highly distributed and decoupled applications, and AWS enables them to build these cloud-native applications by using multiple AWS services for **scale**, **performance**, and **availability**.

Why consider purpose-built databases?

You'll likely remember that one of the core concepts related to microservices is that each service needs to own its own data. But why is this important? There are two really big innovation benefits to following this approach.

1. **First**, it makes it easier to **change your schemas**, which is a common challenge associated with monoliths and shared services. When you can independently own and change your schema, that encapsulation lets you evolve your service without breaking service contracts.

2. The **second** really important point is that **each data store can scale independently**. This means that one service may have a small database and another a very large one, but each service can optimize its scale for performance and cost.

aws marketplace

Purpose-built databases: data models and use cases

Here are some examples of purpose-built databases and common use cases. With AWS, you get access to all these data stores, which can all be spun up in minutes. This allows you to focus on your application and business logic rather than the heavy lifting associated with running and managing databases.

	Common data models	Common use cases	Solution components
 Relational	Referential integrity, ACID transactions, schema-on-write	Lift and shift, ERP, CRM, finance	Amazon Aurora Amazon Relational Database Service (RDS)
 Key value	High throughput, low-latency reads and writes, endless scale	Real-time bidding, shopping cart, social, product catalog, customer preferences	Amazon DynamoDB/ Amazon Managed Apache Cassandra
 Document	Store documents and quickly access querying on any attribute	Content management, personalization, mobile	MongoDB Amazon DocumentDB
 In-memory	Query by key with microsecond latency	Leaderboards, real-time analytics, caching	Amazon ElastiCache
 Graph	Quickly and easily create and navigate relationships between data	Fraud detection, social networking, recommendation engine	Amazon Neptune
 Time series	Collect, store, and process data sequenced by time	IoT applications, event tracking	Amazon Timestream
 Ledger	Complete, immutable, and verifiable history of all changes to application data	Systems of record, supply chain, health care, registrations, financial	Amazon Quantum Ledger Database (QLDB)
 Wide column	Scalable, highly-available, and managed Apache Cassandra–compatible service	Build low-latency applications, leverage open source, migrate Cassandra to the cloud	Amazon Keyspaces Managed Cassandra

Duolingo: A customer success story

Duolingo, the maker of a language learning app, uses purpose-built AWS databases to serve up over 31 billion items for 80 language courses with high performance and scalability.

PRIMARY DATABASE:
Amazon DynamoDB

- 24,000 reads and 3,000 writes per second
- Personalize lessons for users taking six billion exercises per month

IN-MEMORY CACHING:
Amazon ElastiCache

- Instance access to common words and phrases

TRANSACTIONAL DATA:
Amazon Aurora

- Maintain user data

aws marketplace

Purpose-built databases: NoSQL

So far, we've talked about modern applications and how these modern applications achieve scale and high availability by using purpose-built databases. Let's now look at a specific family of databases called NoSQL.

NoSQL stands for Not only SQL. NoSQL databases are non-tabular and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide flexible schemas and scale easily with large amounts of data and high user loads.

Examples of NoSQL databases include Amazon DynamoDB (a key-value, fully managed data store) and MongoDB (a document database).

Deeper into traditional three-tier application architectures

As we touched on earlier, developers were stuck with the single three-tier application architecture —before the introduction of cloud technology. Understanding this previous limitation is helpful in grasping the value of the cloud-native tools available today.

Web servers
Presentation layers

Application servers
Business logic

Database servers
Data layer

- **Basic transactions (OLTP)**
- **Complex queries**
- **Key-value access**
- **Analytics (OLAP)**

All forced into the same relational database

With the three-tier application architectures, developers were limited to a relational database management system. And this database was responsible for everything!

Let's look at some common operations that the single database was responsible for:

1. As companies started out, the database was responsible for transactional *create, read, update, and delete* (CRUD)-based queries like getUser or updateOrderStatus. These type of operations are called online transaction processing (OLTP).

2. As the companies' systems grew, queries grew to match the complexity of the systems and teams would join multiple tables and write queries with nested inner joins.

3. Once the company needed to write reports, the database was responsible for long-running analytical style queries that would impact available resources on the database while the database was servicing customer requests.

4. If the team needed something as simple as a basic key-value datastore with hundreds of thousands of entries, the database had to also handle this load.

This ever-growing load often led to downtime, loss of availability and inefficient resource use—meaning it was not a scalable approach.

SQL vs. NoSQL

NoSQL was introduced to address the limitations we learned about in the previous section.

SQL - Optimized for storage	NoSQL - Optimized for compute
Normalized/relational	Denormalized/hierarchical
Ad hoc queries	Instantiated views
Scale vertically	Scale horizontally
Good for OLAP	Built for OLTP at scale

Relational database management systems (RDBMS) were designed in a time when storage was more expensive as compared to compute. So, SQL and RDBMS evolved around normalization, where storage is minimized. Replicating and duplicating data was considered a bad practice.

At that time, developer teams didn't need to think about the access patterns ahead of time. They normalized, reduced redundancy and storage, and queried data on the fly—but at the cost of scale.

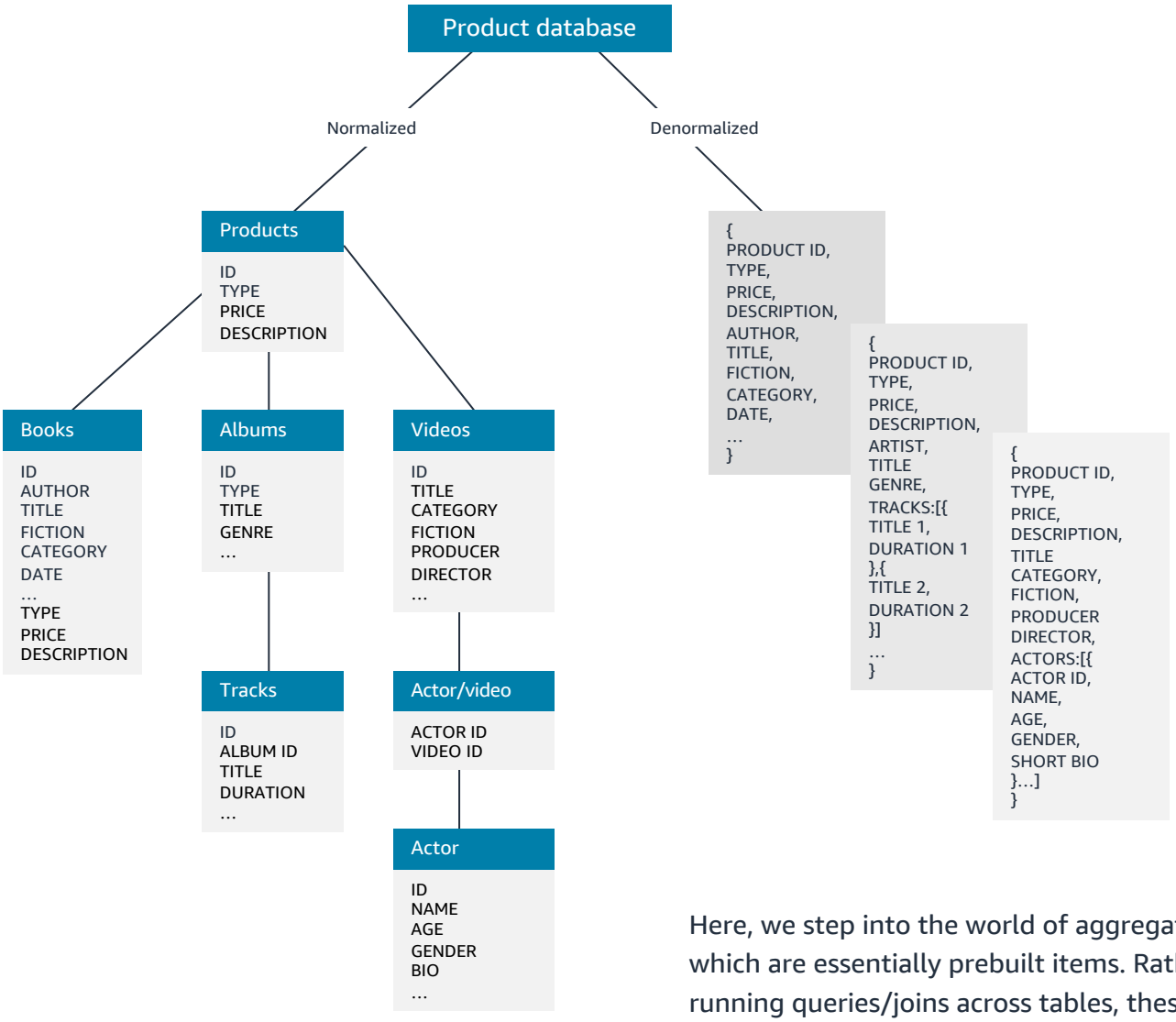
NoSQL was designed at the time when developers focused on OLTP and storage was becoming cheaper. Because storage was cheap, duplicating data was suddenly not a bad thing after all.

If a developer was designing for NoSQL, they had to know their access patterns ahead of time. Access patterns or query patterns define how the users and the system access the data to satisfy business needs. Using SQL means you don't need to know this access pattern ahead of time.

Of the applications developers write today, 90 percent are written to support common business processes that represent OLTP applications. So, NoSQL is actually one of the most relevant technologies you can learn as a developer today.

When we talk about NoSQL, it's important to understand that it's not good for everything—it's good for a certain class of applications. Those applications have repeatable access patterns.

In case of a relational database, the ad-hoc engine gives developers some flexibility. If a developer doesn't yet understand how they are going to access data, then it could be very beneficial to have an ad-hoc query engine, and that's really suitable for an online analytics processing (OLAP) type of workload.



SQL vs. NoSQL design pattern

The typical relational model is nicely organized in “human-readable” constructs—tables that are related to categories. The relational database is modeled to reduce storage.

Here, we step into the world of aggregated items, which are essentially prebuilt items. Rather than running queries/joins across tables, these items are written as they will be retrieved. This requires understanding access patterns and writing the data appropriately, the benefit being nearly limitless scale and consistent performance.

On the right-hand side of this chart—which is denormalized for NoSQL—a developer would simply retrieve a single product record to receive all the information on a specific product.

On the left-hand side—modeled after SQL tables—they have to query for product, execute joins, inner joins, and potentially make additional queries to retrieve all the information for a specific product.

Pro tip:

Forget what you know about relational databases!

An excellent piece of advice is to forget about trying to model your data the way you would in a relational database. Designing for NoSQL is going to feel a bit foreign at first. It is going to feel harder than SQL—as you probably already know how to design schemas and tables in SQL. But RDBMS do not scale like NoSQL and the benefits you gain can be well worth the additional effort to design and think about your access patterns up front.

Some key concepts:

- Forget normalization—replicating and duplicating data is OK.
- There are no joins—forget the idea of one entity type per table. What does this mean? You might have a table that has different types of entities in your entity relationship diagram in the same table.
- An example for NoSQL is that you may have a single table where all your data for a user profile is in a single record. You can return that record and all associated information with a single query or retrieval.

Designing for the cloud means thinking differently—your data access patterns are important

Often developers work with customers who are already using NoSQL but treating it like a relational database—one entity per table, creating joins, and general relational concepts into the application.

While this is one way to do it, it's not how NoSQL was designed and won't provide the benefits it was designed for. Watch out for this type of anti-pattern, where developers fall back into familiar patterns and design concepts.

To take advantage of NoSQL, developers should design around access patterns to ensure that a single query is returning the data they need without having to join and use relational concepts.

Designing for NoSQL and scalability

When designing for NoSQL, developers need to have the following information up front:

- **A defined entity relationship diagram (ERD)—** data is relational, of course, but it doesn't need to be stored that way.
- **Access patterns that predetermine the way data will be accessed.** If this can't be done up front, NoSQL may be the wrong approach for the project.
- **Indices that are designed around access patterns.**

NoSQL use cases

Built for the cloud, NoSQL databases can scale nearly infinitely without impacting performance. Additionally, data models for NoSQL in JSON are natural for developers as the data is similar to code. This list of use cases is not exhaustive, these are just a few examples that take advantage of NoSQL's scale and flexibility as a data store.

Here we see some typical use cases for NoSQL. Most of them take advantage of the key benefits of using NoSQL: Having a flexible schema that allows for changes and experimentation of the application.

Fraud detection

Inventory and catalog management

Personalization

Remote asset monitoring

Financial services and payments

Instant Messaging

Geolocation operations

Logistics and asset management

Content management systems

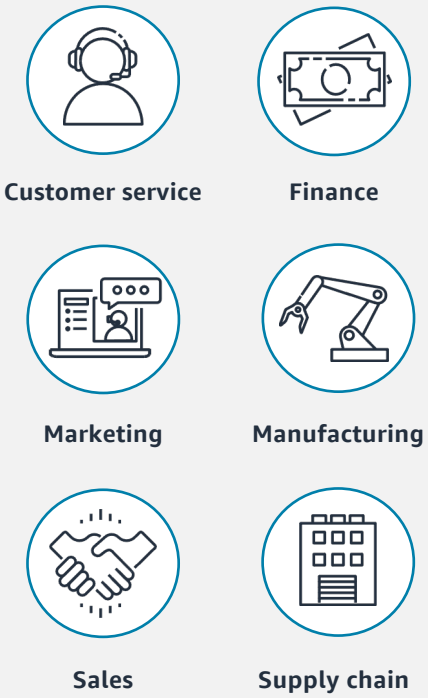
Digital and media management

Data democracy and silos

We just covered NoSQL and the scalability and flexibility it offers. Now let's talk about data democracy.

Data democracy does not mean every employee needs to have access to every bit of data that an organization has. Instead, it's an ongoing process of enabling everybody in an organization—irrespective of their technical know-how—to work with data comfortably, to feel confident talking about it, and, as a result, make data-informed decisions and build customer experiences powered by data.

One of the key issues that data democracy addresses is data silos. A data silo is a repository of data that's controlled by a single department or team, isolated from the rest of the organization. Data silos tend to arise naturally in large companies because departments often have their own goals, priorities, and IT budgets.



But any size organization can end up with challenges related to data silos if they're not intentional about preventing them. Here are some ways to break down data silos and connect data assets:

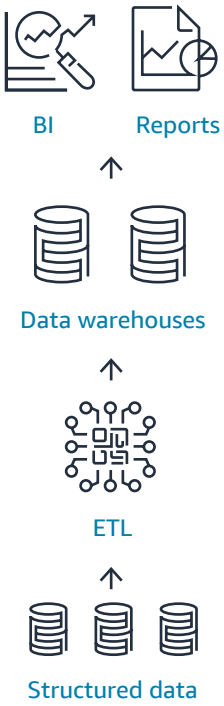
- **Data integration** usually involves extracting siloed data and loading it into a target system or application
- **Data warehouses and data lakes** can store large amounts of structured or unstructured data in a repository
- **Enterprise data management and governance** focuses on preventing new silos from forming
- **Culture changes** that might consist of instituting a data governance initiative or change management program

The cost of data silos depends on the organization but can impact businesses in terms of finances, productivity, effectiveness, missed opportunities, and a lack of trust around data.

Data warehouse

A data warehouse is a central repository of information that can be analyzed to make more informed decisions. In this model, data flows into a data warehouse from transactional systems, relational databases, and other sources—typically, in a regular cadence.

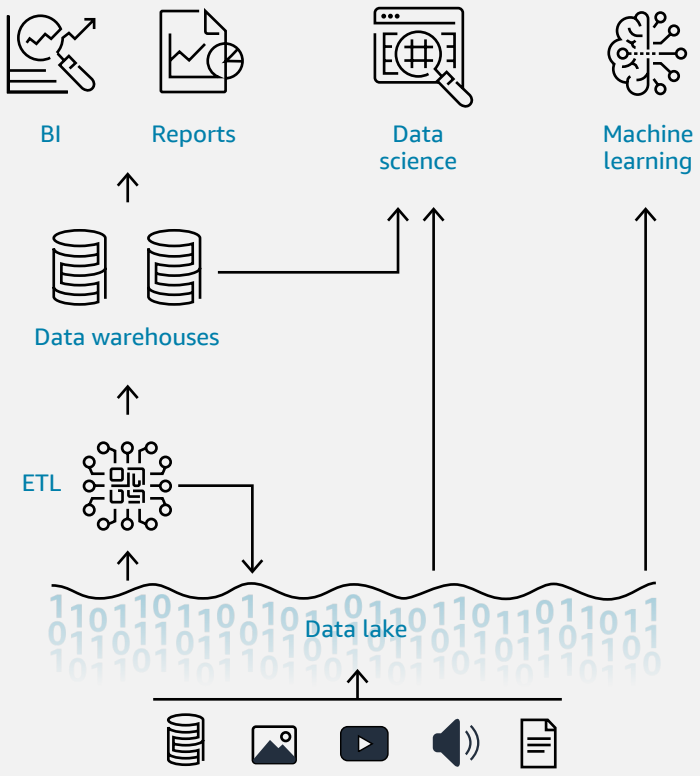
A key feature of data warehouses is that structured data goes through an extract, transform, and load (ETL) process and afterwards that data is stored in a data warehouse database. The primary persona that uses data warehouses is **business professional**.



Data lake

A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale. You can store your data as-is—without having to first structure it—and run different types of analytics, from dashboards and visualizations to big data processing, real-time analytics, and machine learning to guide better decisions.

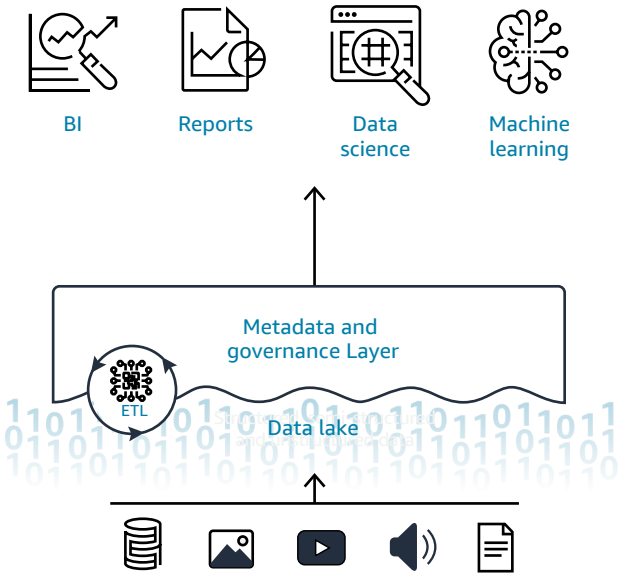
Some key differences compared to a data warehouse: A data lake has semi-structured and unstructured data in addition to a data warehouse. Furthermore, a data lake pulls from multiple data sources and is not limited to a single data source and structure. The primary personas that use data lakes are **data scientists** and **data engineers**.



Data lakehouse

A data lakehouse brings in the best of both the data warehouse and the data lake, combining the reliability, structure, and atomicity, consistency, isolation, and durability (ACID) transactions of data warehouses with the scalability and agility of data lakes.

A lakehouse enables business intelligence and machine learning for all data. Lakehouses bring different organizational personas together onto one platform: **Data scientists**, **data engineers**, **data teams**, **business professionals**, and **software developers**.



Why do developers choose lakehouses?

Lack of data agility and model reproducibility makes it challenging to meet business-specific requirements. A lakehouse simplifies the complexity of reporting, risk management, and compliance by securely streamlining the acquisition, processing, and transmission of data.

Data governance and management

Data silos prevent having a complete view of customer behaviors, opportunities, and the insights needed for personalization at scale. Lakehouses unify a variety of data, enabling personalized experiences that drive opportunities and customer satisfaction.

Deeper customer insights

Vendor lock-in and disjointed tools hinder the ability to perform real-time analytics that drive and democratize smarter business decisions. Lakehouses allow for rapid ingestion of all your data sources at scale to make better investment decisions, quickly detect new fraud patterns, and bring real-time capabilities to risk management practices.

Real-time decisions

Legacy technologies can't harness customer insights from fast-growing unstructured and alternative data sets, and don't offer open data sharing capabilities. Data lakehouses bring together vast amounts of internal and third-party data to share innovative business solutions, monetize new data products, and deliver advanced analytics capabilities.

Access to third-party data

aws marketplace

DevOps for databases

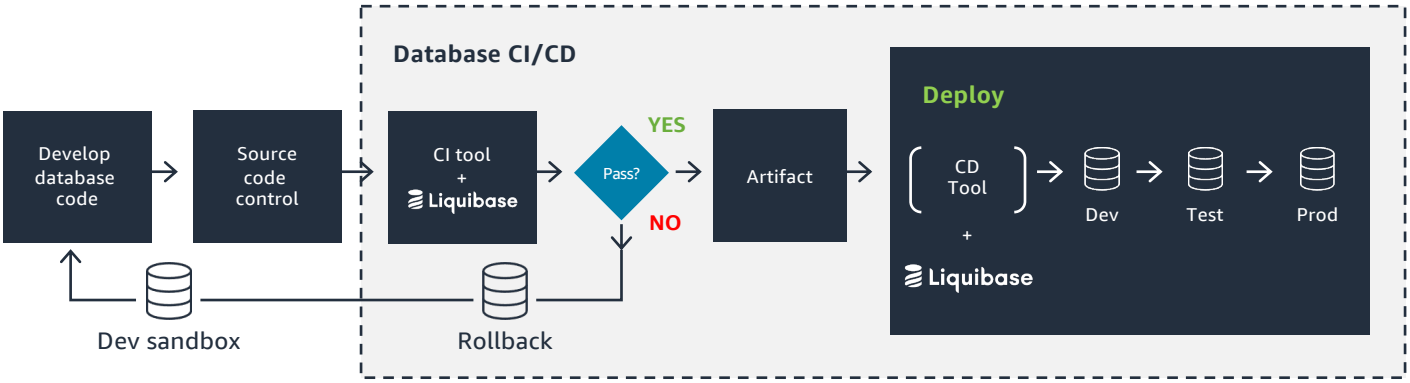
We just talked about data lakehouses and the capabilities that can give your business an edge. Now we are going to shift focus and talk about DevOps for databases.

DevOps is defined as the combination of cultural philosophies, practices, and tools that increases an organization's ability to deliver applications and services at high velocity. In most DevOps implementations, data is an afterthought. So, let's talk about a few DevOps capabilities for databases that can help bring it back into the spotlight.

Continuous Integration/Continuous Delivery for databases

CI/CD for databases enables the rapid integration of database schema and logic changes into application development efforts and provide immediate feedback to developers on any issues that arise. Database CD produces releasable database code in short, incremental cycles.

CI/CD for databases leverages DevOps tools such as Jenkins, CircleCI, AWS CodePipeline, and layers in database-specific tooling that allows for database changes to follow the same development workflows that software engineering teams are already familiar with.

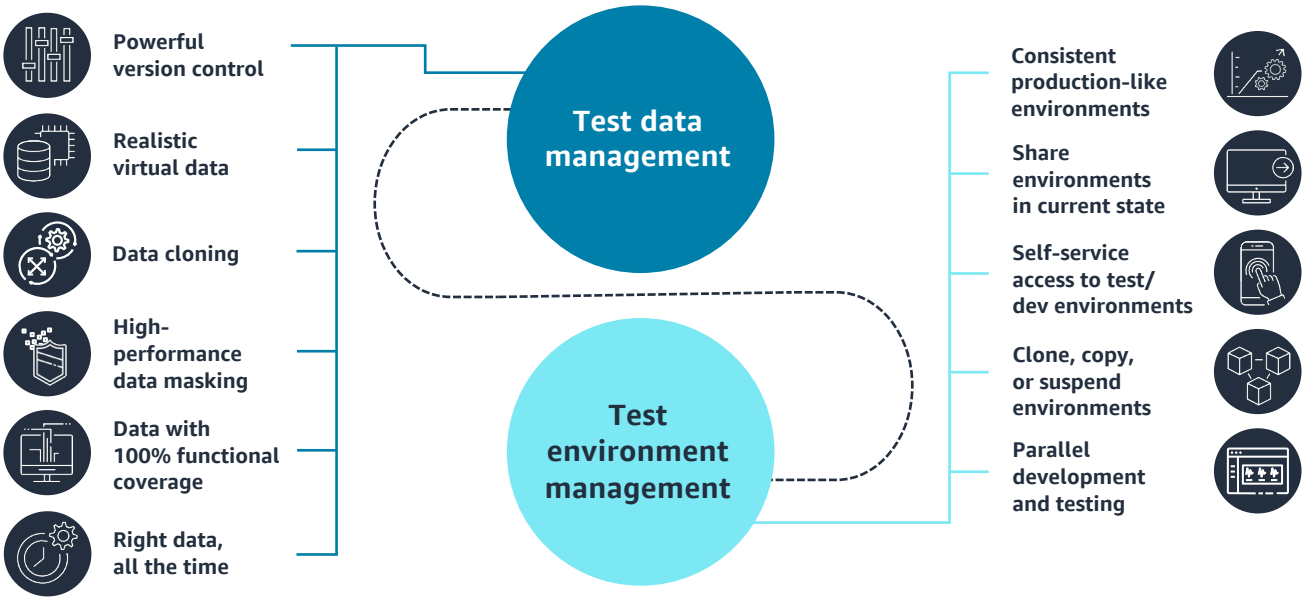


In this example, Liquibase—an open-source database-independent library for tracking, managing, and applying database schema changes—is layered in to provide lifecycle capabilities for databases. Some of the benefits of using Liquibase include version-controlled database schema changes, branching and merging of database changes, which allows teams of developers to work simultaneously on database changes, and easy rollback of changes as well as fix forward.

Backup and disaster recovery

The next capability we'll look at is **backup and disaster recovery (DR)**.

The database is a very critical system for most applications and needs to be extremely resilient. Organizations need to have a clear process and practice for database backup and disaster recovery. Mature DevOps implementations practice gamedays and chaos engineering to make sure their backup and DR strategies work as expected, without causing any downtime.



Test data management

The third capability is **test data management**: Continuous Integration, Continuous Delivery and Continuous Deployment all depend on strong test automation. And for the test automation to be reliable and complete, teams need access to complete, high-quality data sets.

Happy path testing—a type of software testing that uses known input and produces an expected output—is straightforward and often does not require extensive amounts of data.

But for Continuous Integration, Continuous Delivery and Continuous Deployment to work effectively, the tests need to be complete—verifying all scenarios and edge cases. This is not possible unless you have access to production-scale data (of course keeping regulatory compliance and any other organizational policies in mind).

Observability into applications

The last capability we'll look at is **Observability**. Observability into metrics, events, logs, and traces (MELT) data across all systems is a critical capability. With modern applications being hybrid and distributed, it is important to get visibility into third-party services like email, payments, location services, delivery services, and even data changes to get the complete picture as you are debugging your application.

A good example would be debugging a failed order: Was it due to a third-party payment gateway being down or data schema change? Without complete visibility into a transaction, it is impossible to debug and identify the root cause.

Key takeaways for Database DevOps

Let's review some of the important concepts we've covered so far.

Use cloud-native purpose-built databases

One size doesn't fit all, and developers may be trying to solve a problem that is already solved for them in a purpose-built database.

NoSQL does not mean non-relational

Relationships are accessed and modeled differently than SQL. Developers need to model data differently in NoSQL, so forget what you know about RDBMS. Think about access patterns when modeling. NoSQL can be a great fit for OLTP, where databases are read, written, and updated frequently.

Data democracy

Data lakehouses provide the flexibility, cost-efficiency, and scale of data lakes with the data management and ACID transactions of data warehouses. Plus, lakehouses are designed for most organizational personas, which further democratizes data.

DevOps for databases

This is where we highlighted four key capabilities application development teams need to build:

1. CI/CD for database changes using tools like Liquibase
2. Backup and DR to ensure resiliency and that applications are built to handle failures
3. Test data management to achieving continuous deployment where there is no manual intervention to deploy to production
4. Observability into MELT data across all systems, including databases for complete root cause analysis while debugging issues

aws marketplace

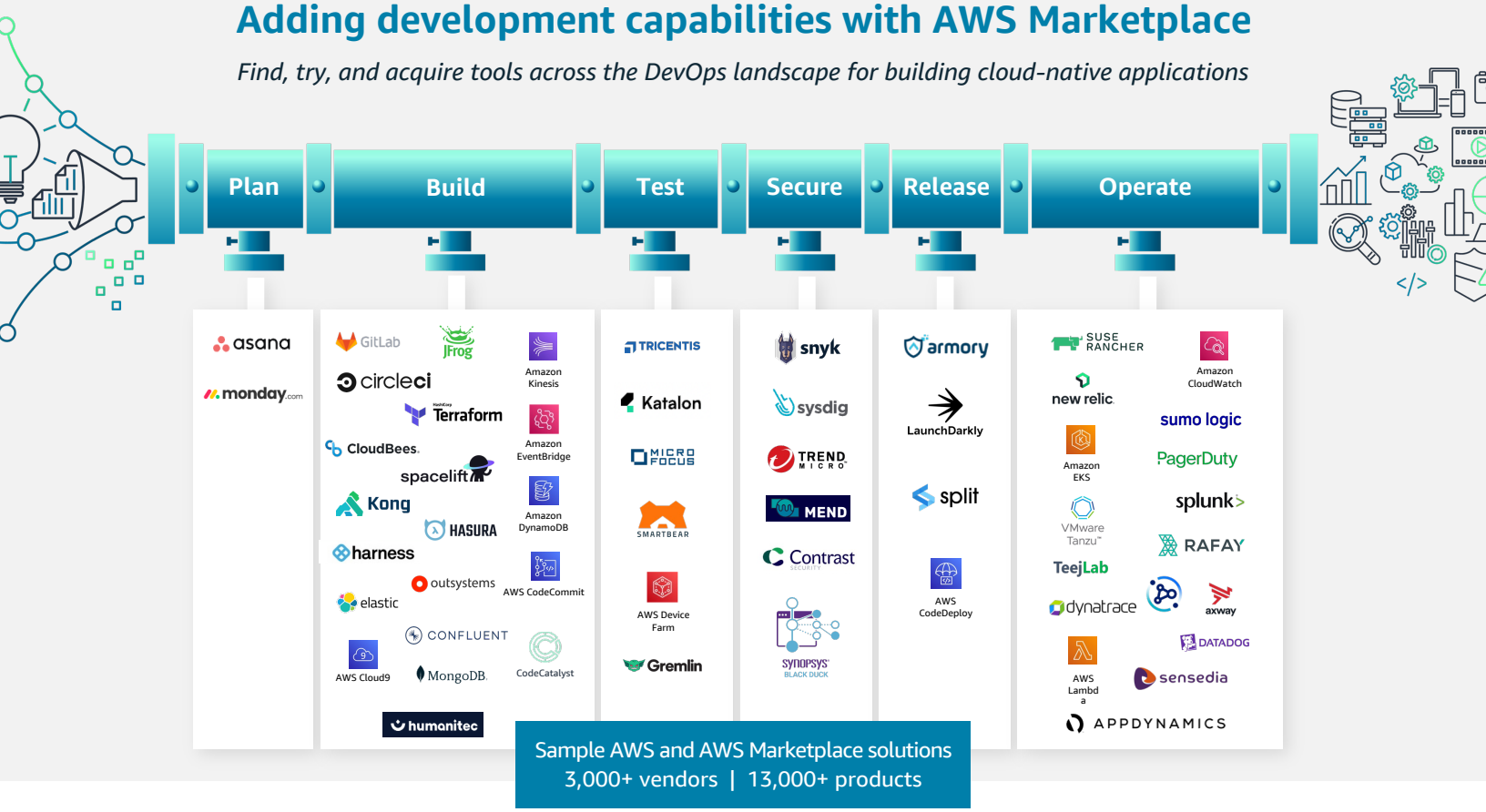
Implementing modern data management processes and tools

In this section, we'll look at some of the best-fit tools you could use to achieve the tenets discussed in the previous section. At AWS, we've long been believers in enabling builders to use the right tool for the job—and when you build with AWS, you're provided with choice. You can build using the native services AWS provides or use AWS Marketplace to acquire third-party software offered by AWS Partners to take away the heavy lifting and allow your development teams to focus on delivering value to customers.

Let's take a deeper look at three key components at this stage of your cloud-native journey: a way to get data out of silos and democratize data for all personas in the company, a purpose-built database for OLTP applications that can scale to meet the demands of customers, and end-to-end Observability of your application and data workloads.

Adding development capabilities with AWS Marketplace

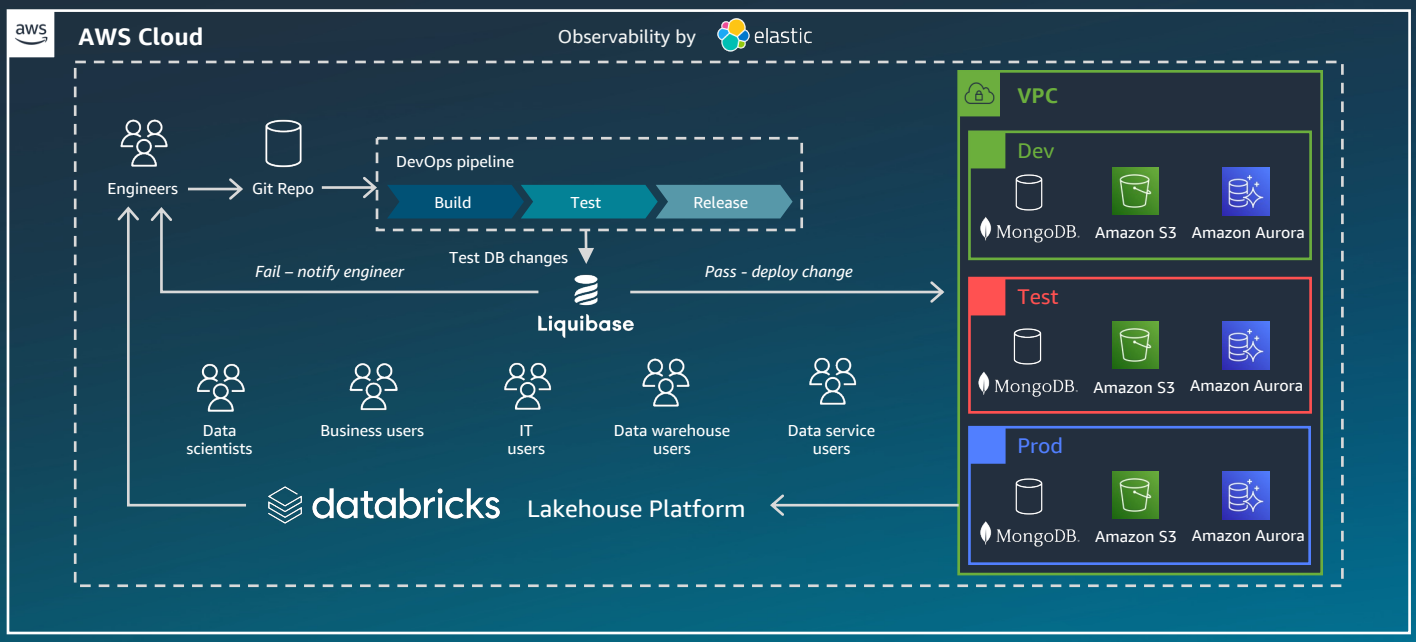
Find, try, and acquire tools across the DevOps landscape for building cloud-native applications



Sample AWS and AWS Marketplace solutions
3,000+ vendors | 13,000+ products

AWS Marketplace is a cloud marketplace that makes it easy to find, try, and acquire the tools you need to build cloud-native. More than 13,000 products from over 3,000 Independent Software Vendors are listed in AWS Marketplace—many of which you can try for free and, if you decide to use, will be billed through your AWS account.

For our example architecture, we'll use **MongoDB** for purpose-built databases, **Databricks** to democratize data, and **Elastic** for end-to-end visibility. Here is an architectural diagram that illustrates how these three components can be implemented.



MongoDB Atlas

MongoDB Atlas

[Try it in AWS Marketplace ›]

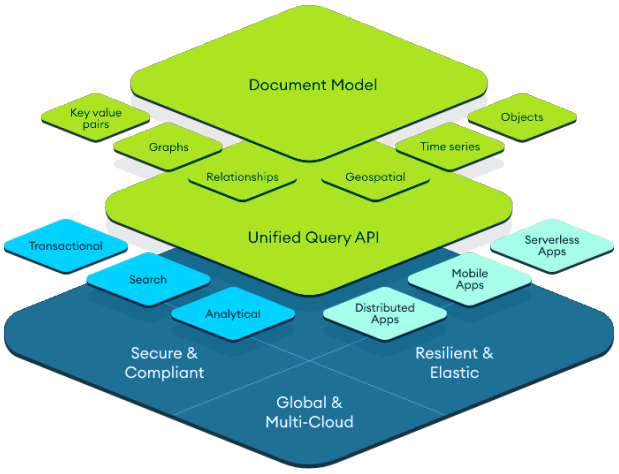
[Watch a demo and learn more ›]

MongoDB Atlas provides the purpose-built database for OLTP, along with Amazon Simple Storage Service (Amazon S3) for object storage, and Amazon Aurora for online analytic processing.

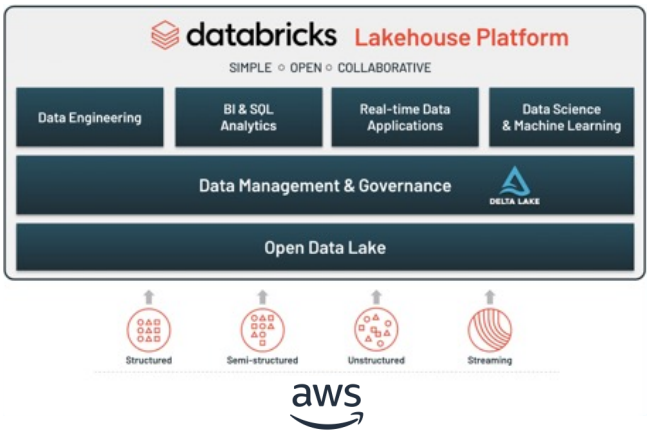
To democratize data, the solution uses the **Databricks Lakehouse Platform**, which serves the needs of business professionals who need to run reports and business analytics, as well as data scientists, data engineers, and software engineers who need to run machine learning (ML) workloads to surface business insights.

Observability is provided by **Elastic** to enable visibility into cloud-native applications that span multiple Availability Zones, multiple AWS Regions, multiple accounts, and hybrid clouds.

More about how each component plays a part in the architecture



MongoDB is a NoSQL document database. MongoDB Atlas is an integrated suite of cloud-native database services that allow you to address a wide variety of use cases—from transactional to analytical, from search to data visualizations.



Databricks enables massive-scale data engineering, collaborative data science, full-lifecycle machine learning and business analytics. Combining the best of data warehouses and data lakes, the Databricks Lakehouse Platform is built on an open and reliable data foundation that efficiently handles all data types and applies one common security and governance approach across all data and cloud platforms.

Elastic is the solution historically acknowledged as providing the de facto search platform: Elasticsearch, Logstash, and Kibana—commonly known as the ELK stack. Today, Elastic offers three solutions:

Elastic Security

Elastic Enterprise Search

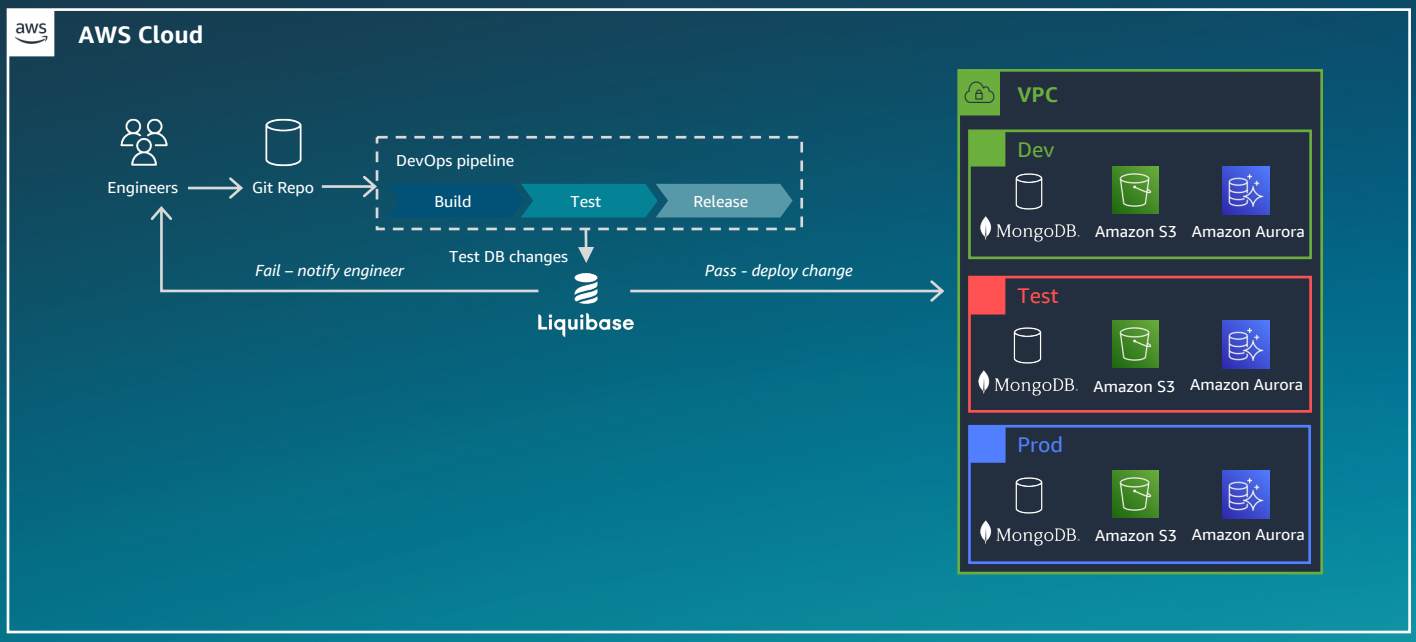
Elastic Observability

Elastic offers a SaaS model, a self-managed solution, and hybrid deployments relying on federation concepts to offer hybrid architectures.

How the solution works

In this architecture, an engineer would check in a database change—also known as a migration—into a Git repository. The recommended format for Liquibase is XML, as this format is database agnostic.

The change triggers a DevOps pipeline that reads in the XML, creates the change set, executes the change, and tests the database changes. If the change passes the tests, the change is promoted, and the environment database is updated. If the change fails, the database change is rolled back and a notification is sent to the engineer.

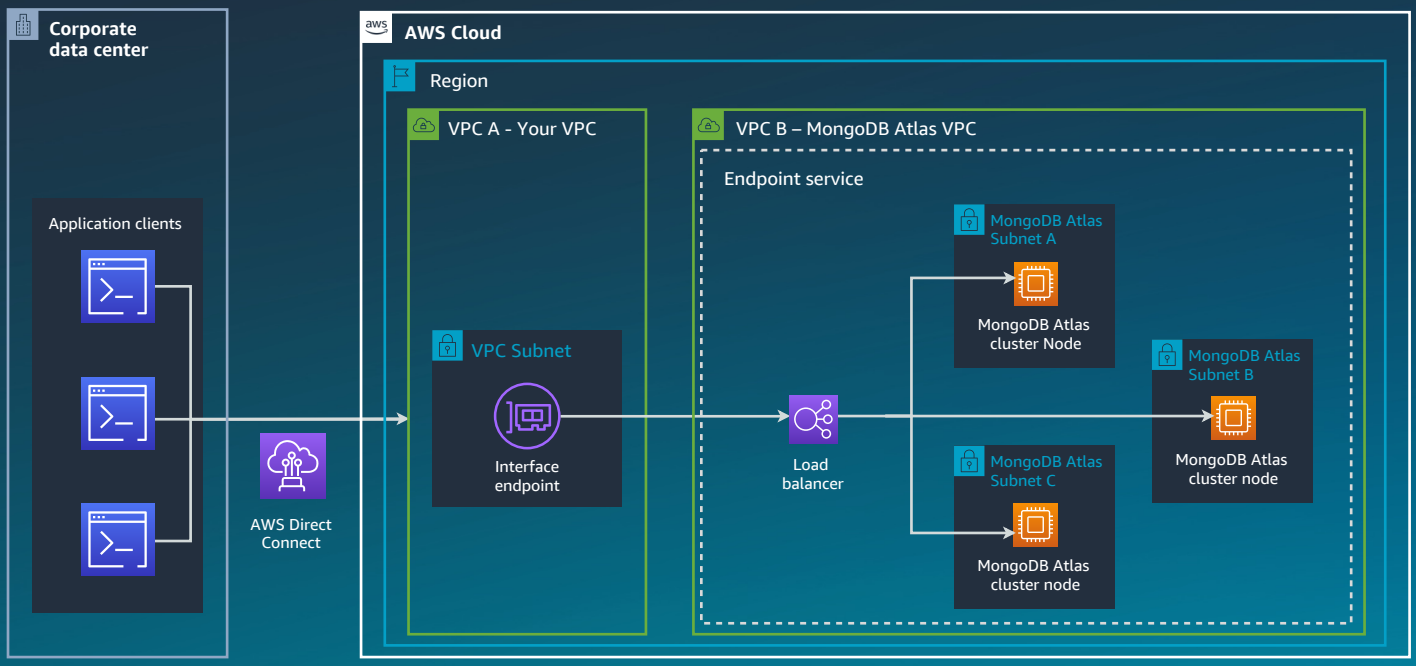


Because it maintains a history and sequence of changes—besides production databases—Liquibase is a good use case for standing up databases for sandbox and temporary environments.

aws marketplace

Cloud-native NoSQL with MongoDB Atlas

The solution for a NoSQL purpose-built document database is provided by MongoDB Atlas. MongoDB Atlas is a SaaS solution, and connecting this to your AWS account is very straightforward.



In this example, an AWS Direct Connect is used to connect an on-premises environment to a MongoDB Atlas cluster. The cluster is distributed to multiple locations to provide scalability and availability. MongoDB takes care of managing and maintaining the infrastructure so developers can focus on building the business logic and application.

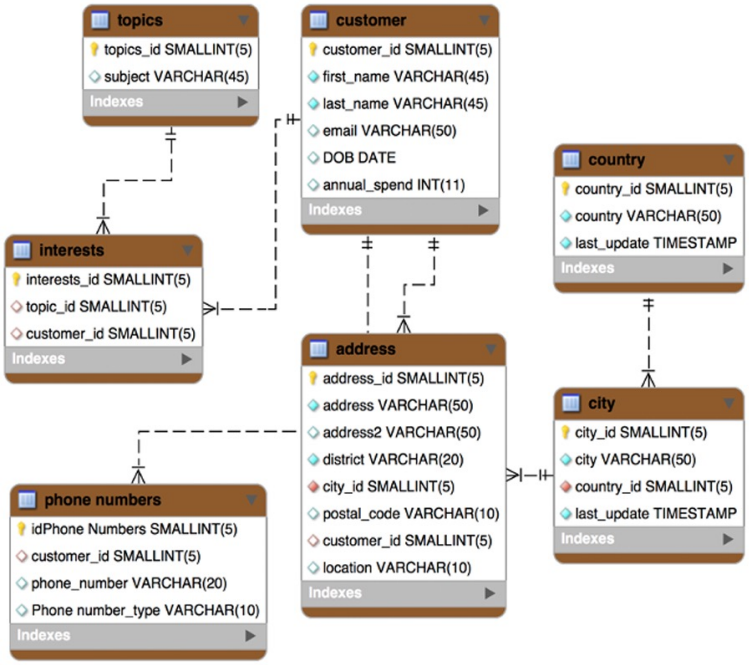
The document model is fundamentally different

MongoDB's document model naturally maps to objects in code and eliminates requirements to use object-relational mapping (ORMs). It breaks down complex interdependencies between developer and database administration (DBA) teams.

With MongoDB, developers can represent the data of any structure, and each document can contain different fields. The schema can be modified at any time. Additionally, MongoDB is strongly typed for ease of processing with over 20 binary encoded JSON data types.

Tabular (relational) data model

Related data split across multiple records and tables



Document data model

Related data contained in a single, rich document

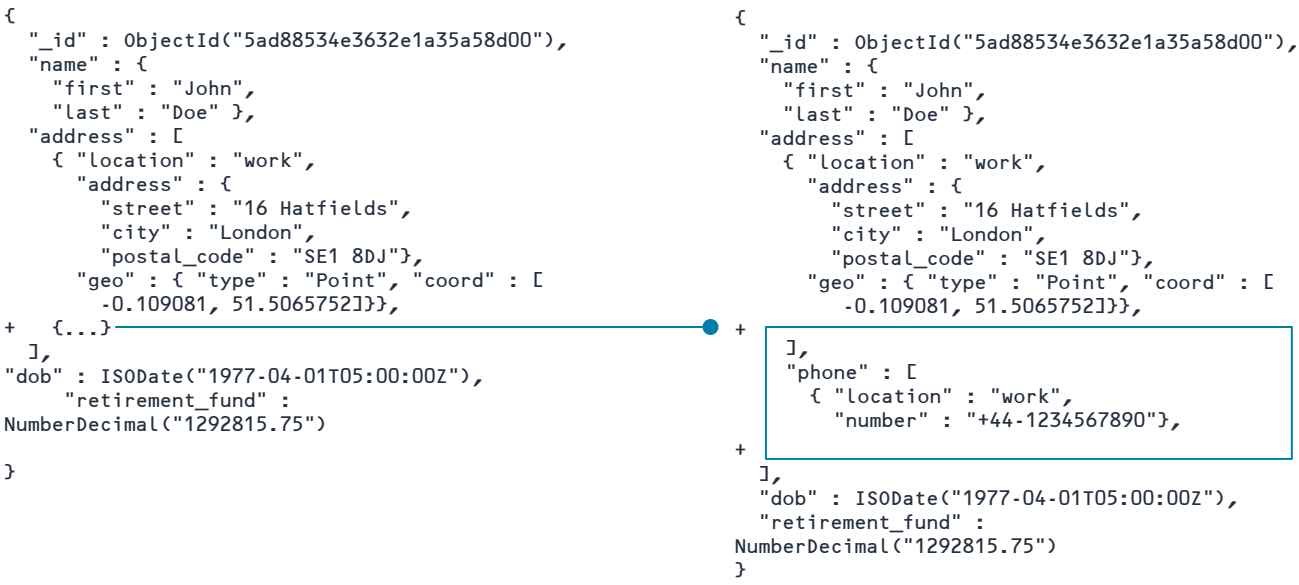
```
{
  "_id" : ObjectId("5ad88534e3632e1a35a58d00"),
  "name" : {
    "first" : "John",
    "last" : "Doe" },
  "address" : [
    { "location" : "work",
      "address" : {
        "street" : "16 Hatfields",
        "city" : "London",
        "postal_code" : "SE1 8DJ"},
      "geo" : { "type" : "Point", "coord" : [
         -0.109081, 51.5065752]}},
+   {...}
  ],
  "dob" : ISODate("1977-04-01T05:00:00Z"),
  "retirement_fund" :
NumberDecimal("1292815.75")
}
```

The example illustrated here shows the relational model on the left and a document model on the right. Notice that all the information contained on the left is also on the right side, which highlights the simplicity of having all information contained in a single JSON object. It's easy to read and a developer doesn't have to understand the complex tabular relationship.

Flexible: Adapt to change

Perhaps the most powerful feature of document databases is the ability to nest objects inside of documents. A good rule of thumb for structuring data in MongoDB is to prefer embedding data inside documents as opposed to breaking it apart into separate collections. There are, of course, some exceptions such as needing to store unbounded lists of items or needing to look up objects directly without retrieving a parent document.

Add new fields dynamically at runtime



Note in this example that the name field is a nested object containing both given and family name components, and that the address field stores an array containing multiple addresses. Each address can have different fields in it, which makes it easy to store different types of data.

MongoDB for Microsoft Visual Studio (Microsoft VS) code

MongoDB also provides an extension for Microsoft VS code, which lets you work with MongoDB and your data directly within your coding environment. You can use MongoDB for Visual Studio Code to:

- **Explore your MongoDB data**
- **Prototype queries and run MongoDB commands**
- **Create a Shared Tier Atlas cluster using a Terraform template.**

Top features of MongoDB Atlas

1. **Ad-hoc queries for optimized real-time analytics.** You might have noticed that earlier we talked about NoSQL for online transaction processing—MongoDB does that as well as online analytics processing.

2. **Indexing for better query executions.** MongoDB indices can be created on demand to accommodate real-time, ever-changing query patterns and application requirements. They can also be declared on any field within any of your documents, including those nested within arrays.

3. **Replication for better data availability and stability.** When your data only resides in a single database, it is exposed to multiple potential points of failure, such as a server crash, service interruptions, or even hardware failure. Any of these events would make accessing your data nearly impossible. Replication allows you to sidestep these vulnerabilities by deploying multiple servers for disaster recovery and backup. Horizontal scaling across multiple servers that house the same data means greatly increased data availability and stability.

4. **Sharding.** This is the process of splitting larger datasets across multiple distributed collections. This helps the database distribute and better execute queries. Sharding in MongoDB allows for much greater horizontal scalability.

5. **Large-scale load balancing.** MongoDB supports this via horizontal scaling features like replication and sharding. The platform can handle multiple concurrent read and write requests for the same data with best-in-class concurrency control and locking protocols that ensure data consistency. There's no need to add an external load balancer —MongoDB ensures that each and every user has a consistent view and quality experience with the data they need to access.

6. **Database provisioning, maintenance, and upgrades.** MongoDB Atlas handles all of the heavy lifting behind these tasks. Select the cluster configuration you want using the UI or API and deploy a new cluster or update an existing cluster in minutes. Security patches and minor version upgrades are automatically applied, and all updates occur in a rolling fashion across deployment to reduce performance impact to applications. And if a node goes down, MongoDB Atlas immediately elects a new primary and restores or replaces the offline node to ensure continuous availability.

aws marketplace

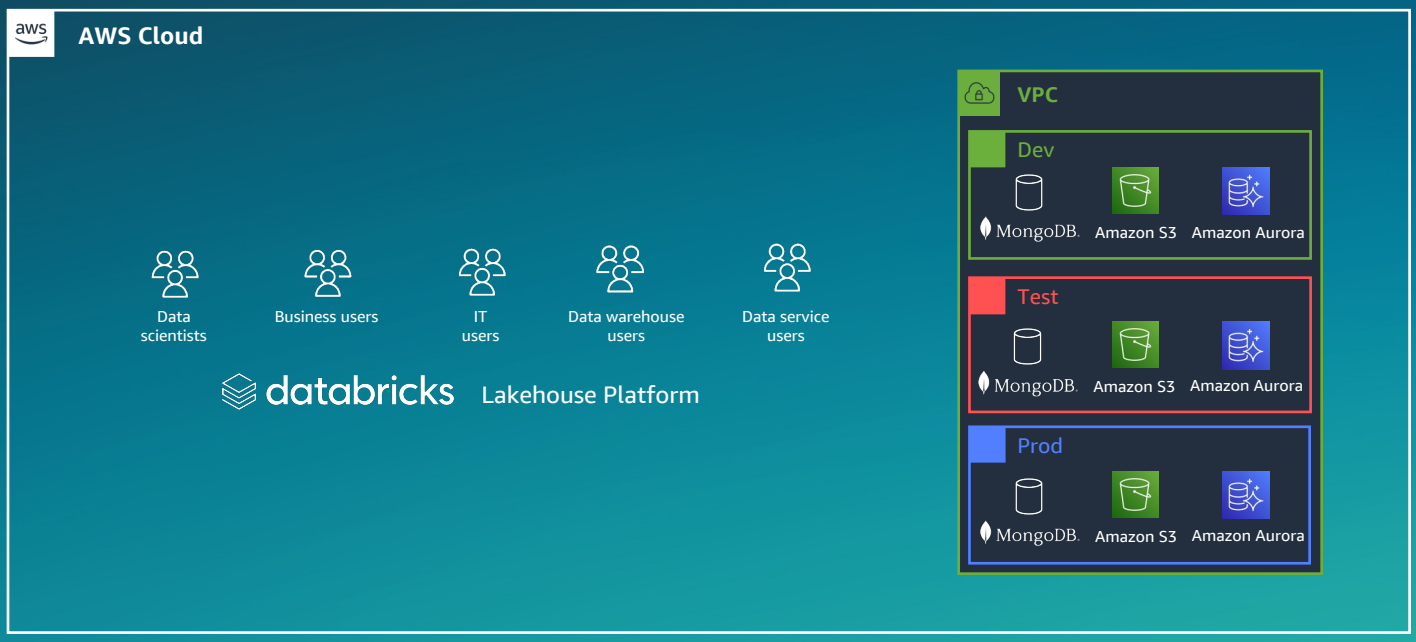
databricks

Databricks Lakehouse Platform

Try it in AWS Marketplace ›

Start Databricks on AWS training ›

The Databricks Lakehouse Platform combines the best elements of data lakes and data warehouses to deliver the reliability, strong governance, and performance of data warehouses with the openness, flexibility, and ML support of data lakes.



This unified approach simplifies the modern data stack by eliminating data silos that traditionally separate and complicate data engineering, analytics, BI, data science, and machine learning. It's built on open source and open standards to maximize flexibility. Additionally, its common approach to data management, security, and governance helps developers operate more efficiently and innovate faster.

In our solution, Databricks integrates with the data plane in a customer's account for access to MongoDB, Amazon S3, and Amazon Aurora database services. The control plane in the Databricks cloud environment allows for uniform access for different personas in the organization via https, JDBC, ODBC, or REST APIs.

Key features of Databricks Lakehouse Platform

The Databricks Lakehouse Platform architecture:

Takes a decentralized approach to data ownership. Organizations can create many different lakehouses to serve the individual needs of specific business groups. Based on their needs, they can store and manage various data—images, video, text, structured tabular data, and related data assets such as ML models and associated code to reproduce transformations and insights.

Helps organizations manage data as a product by providing different data team members in domain-specific teams with complete control over the data lifecycle.

Provides an end-to-end data platform for data management, data engineering, analytics, data science, and machine learning with integrations for a broad ecosystem of tools. Adding data management on top of existing data lakes simplifies data access and sharing—anyone can request access.

Enables discovery of data and other artifacts like code and ML models. Organizations can assign different administrators to different parts of the catalog to decentralize control and management of data assets. This approach —a centralized catalog with federated control—preserves the independence and agility of the local domain-specific teams while ensuring data asset reuse across these teams and enforcing a common security and governance model.

Ingesting logs and metrics

No solution is complete without end-to-end visibility of your application and data stack. With the complexity of modern environments, tracing issues or performance problems across microservices that extend into other Availability Zones, Regions, accounts, and even hybrid clouds, is difficult without a tool to aggregate all this information.

elastic

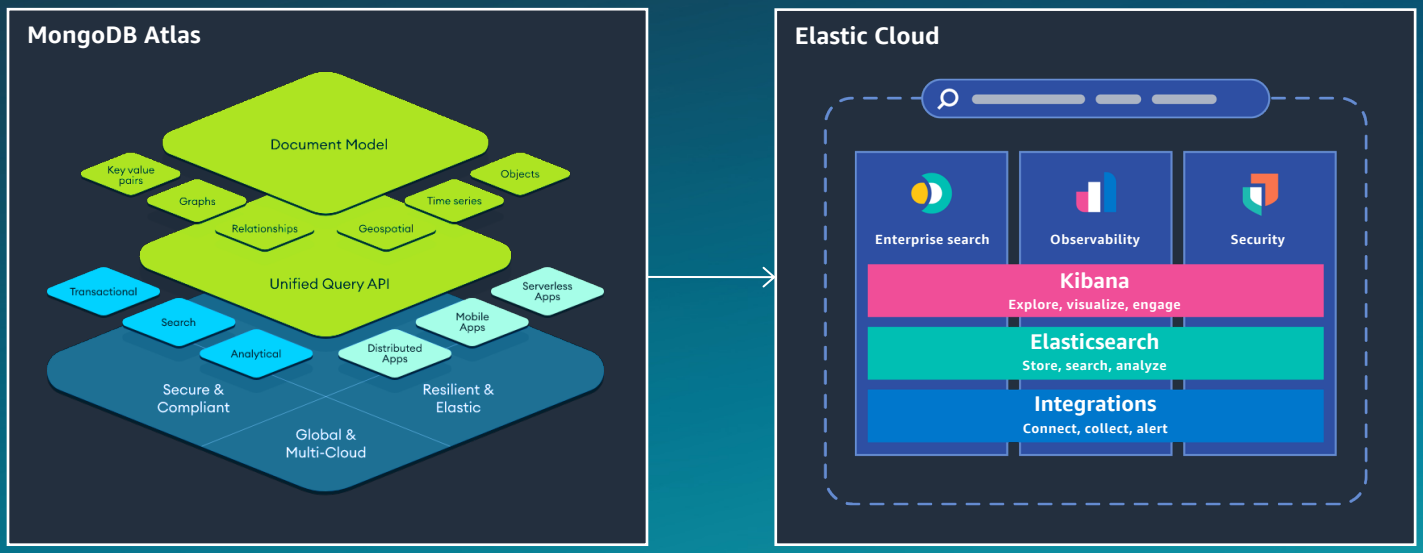
[Try it in AWS Marketplace ›]

[Watch a demo and learn more ›]

[Start a hands-on lab ›]

Elastic Cloud is a SaaS solution that can be deployed in minutes across supported AWS Regions and scales capacity automatically so you can focus on building features and functionality versus building and maintaining monitoring tools.

There are over 23 out-of-the-box integrations for AWS products and services that allow MELT data to be sent into the Elastic Cloud. Elastic supports cloud-native technologies such as Kubernetes, AWS Lambda, DynamoDB, and many others.



Ingesting telemetry from MongoDB Atlas

MongoDB monitoring is a critical component of all database administration, and tight MongoDB cluster monitoring will show the state of your database. However, due to its complex architecture, monitoring can be a challenging task. With the Elastic Agent, logs and metrics are streamed to Elastic Cloud with full visualizations in Kibana.

Elastic Cloud supports integrations with both MongoDB and Databricks for full observability across applications and these data sources.

Key benefits of Elastic Cloud Observability

Elastic Cloud enables you to:

Get up and running in minutes. This not only includes setup of the Elastic Cloud services, but also ingesting data from AWS accounts and hybrid cloud accounts.

Save time on operational overhead and focus on delivering business value by leveraging managed services. Automatically scale up or down and pay only for the resources you use.

Make sure your deployment is secure from the ground up, including but not limited to OS hardening, network security controls, and encryption for data in motion as well as data at rest.

Collect telemetry data seamlessly. Ingest all business and operational data, including support for open instrumentation standards and open-source projects such as OpenTelemetry, Jaeger, and Prometheus. This allows you to consolidate monitoring tools and efficiently store data at affordable costs to visualize and analyze historical trends.

Recap: Modern data architecture

To recap the solution:

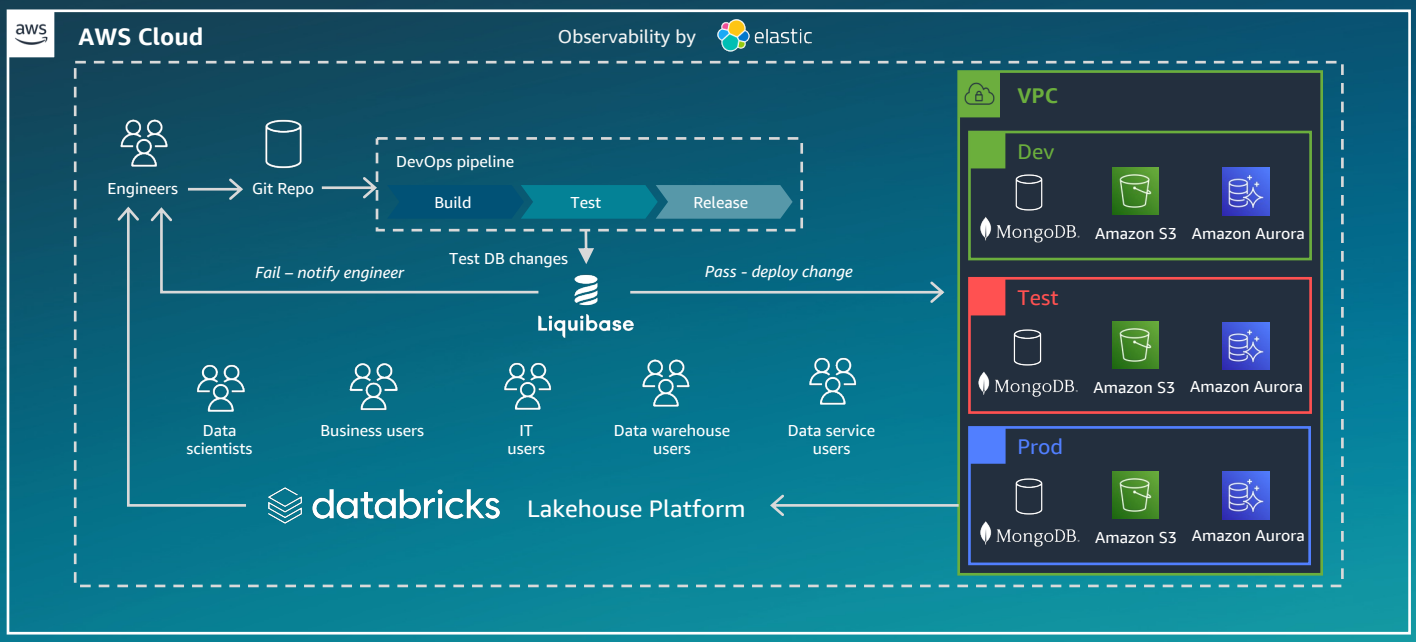
Liquibase is used to manage database migrations and treat data as part of a DevOps lifecycle.

The capabilities of **MongoDB** are leveraged for NoSQL document storage that can scale to handle performance and availability requirements of the most demanding cloud-native applications.

Databricks Lakehouse Platform breaks down data silos and democratizes data.

Elastic Cloud Observability brings together MELT data into one place to visualize.

An important final point is that all of these services are SaaS offerings on AWS that abstract away the heaving lifting of managing and maintaining servers and infrastructure, helping developers focus on creating new features to drive increased customer satisfaction.



aws marketplace

Implement your modern data management strategy today

Databricks Lakehouse Platform, MongoDB Atlas, and Elastic Cloud can be used together along with AWS to build the foundation for establishing a well-engineered approach to modern data architecture. Establishing these capabilities can provide a strong foundation for continuing to advance through your cloud-native journey. You can explore these and other DevOps and modern data management-focused tools in AWS Marketplace.

To get started, visit: https://aws.amazon.com/marketplace/solutions/devops

AWS Marketplace

Third-party research has found that customers using AWS Marketplace are experiencing an average time savings of 49 percent when needing to find, buy, and deploy a third-party solution. And some of the highest-rated benefits of using AWS Marketplace are identified as:

Time to value

Cloud readiness of the solution

Return on Investment

Part of the reason for this is that AWS Marketplace is supported by a team of solution architects, security experts, product specialists, and other experts to help you connect with the software and resources you need to succeed with your applications running on AWS.

Over 13,000 products from 3,000+ vendors:

MongoDB Atlas

databricks

circleci

elastic

HashiCorp

JFrog

Buy through AWS Billing using flexible purchasing options:

- Free trial
- Pay-as-you-go
- Hourly | Monthly | Annual | Multi-Year
- Bring your own license (BYOL)
- Seller private offers
- Channel Partner private offers

Deploy with multiple deployment options:

- AWS Control Tower
- AWS Service Catalog
- AWS CloudFormation (Infrastructure as Code)
- Software as a Service (SaaS)
- Amazon Machine Image (AMI)
- Amazon Elastic Container Service (ECS)
- Amazon Elastic Kubernetes Service (EKS)

aws marketplace

Get started today

Visit http://aws.amazon.com/marketplace to find, try and buy software with flexible pricing and multiple deployment options to support your use case.

https://aws.amazon.com/marketplace/solutions/devops

Authors:

James Bland
Global Tech Lead for DevOps, AWS

Aditya Muppavarapu
Global Segment Leader for DevOps, AWS