AWS Prescriptive Guidance

Enabling data persistence in microservices
AWS Prescriptive Guidance: Enabling data persistence in microservices

Copyright © Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon’s trademarks and trade dress may not be used in connection with any product or service that is not Amazon’s, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.
# Table of Contents

Introduction ...................................................................................................................................... 1  
Targeted business outcomes ........................................................................................................ 2 
Patterns for enabling data persistence .......................................................................................... 3  
  Database-per-service pattern ....................................................................................................... 3  
  API composition pattern ............................................................................................................. 5  
  CQRS pattern ............................................................................................................................ 6  
  Event sourcing pattern ............................................................................................................... 7  
    Amazon Kinesis Data Streams implementation ....................................................................... 8  
    Amazon EventBridge implementation ................................................................................... 8  
  Saga pattern ............................................................................................................................ 10  
  Shared-database-per-service pattern .......................................................................................... 11  
FAQ ................................................................................................................................................ 13  
  When can I modernize my monolithic database as part of my modernization journey? .......... 13  
  Can I keep a legacy monolithic database for multiple microservices? ................................. 13  
  What should I consider when designing databases for a microservices architecture? .......... 13  
  What is a common pattern for maintaining data consistency across different microservices? 13  
  How do I maintain transaction automation? ............................................................................. 13  
  Do I have to use a separate database for each microservice? .................................................. 14  
  How can I keep a microservice's persistent data private if they all share a single database? .... 14  
Resources ........................................................................................................................................ 15  
  Related guides and patterns ........................................................................................................ 15  
  Other resources ........................................................................................................................ 15  
AWS Prescriptive Guidance glossary ........................................................................................... 16  
Document history ............................................................................................................................. 23
Enabling data persistence in microservices

Tabby Ward, Senior Modernization Architect, Amazon Web Services
Hari Ohm Prasath Rajagopal, Senior Modernization Architect, Amazon Web Services
Apoorva Kulkarni, Modernization Architect, Amazon Web Services

January 2021 (last update (p. 23): February 2021)

Organizations constantly seek new processes to create growth opportunities and reduce time to market. You can increase your organization's agility and efficiency by modernizing your applications, software, and IT systems. Modernization also helps you deliver faster and better services to your customers.

Application modernization is a gateway to continuous improvement for your organization, and it begins by refactoring a monolithic application into a set of independently developed, deployed, and managed microservices. This process has the following steps:

- **Decompose monoliths into microservices** – Use patterns to break down monolithic applications into microservices.
- **Integrate microservices** – Integrate the newly created microservices into a microservices architecture by using Amazon Web Services (AWS) serverless services.
- **Enable data persistence for microservices architecture** – Promote polyglot persistence among your microservices by decentralizing their data stores.

Although you can use a monolithic application architecture for some use cases, modern application features often don't work in a monolithic architecture. For example, the entire application can't remain available while you upgrade individual components, and you can't scale individual components to resolve bottlenecks or hotspots (relatively dense regions in your application's data). Monoliths can become large, unmanageable applications, and significant effort and coordination is required among multiple teams to introduce small changes.

Legacy applications typically use a centralized monolithic database, which makes schema changes difficult, creates a technology lock-in with vertical scaling as the only way to respond to growth, and imposes a single point of failure. A monolithic database also prevents you from building the decentralized and independent components required for implementing a microservices architecture.

Previously, a typical architectural approach was to model all user requirements in one relational database that was used by the monolithic application. This approach was supported by popular relational database architecture, and application architects usually designed the relational schema at the earliest stages of the development process, built a highly normalized schema, and then sent it to the developer team. However, this meant that the database drove the data model for the application use case, instead of the other way round.

By choosing to decentralize your data stores, you promote polyglot persistence among your microservices, and identify your data storage technology based on the data access patterns and other requirements of your microservices. Each microservice has its own data store and can be independently scaled with low-impact schema changes, and data is gated through the microservice's API. Breaking down a monolithic database is not easy, and one of the biggest challenges is structuring your data to achieve the best possible performance. Decentralized polyglot persistence also typically results in
eventual data consistency, and other potential challenges that require a thorough evaluation include
data synchronization during transactions, transactional integrity, data duplication, and joins and latency.

This guide is for application owners, business owners, architects, technical leads, and project managers. The guide provides the following six patterns to enable data persistence among your microservices:

- Database-per-service pattern (p. 3)
- API composition pattern (p. 5)
- CQRS pattern (p. 6)
- Event sourcing pattern (p. 7)
- Saga pattern (p. 10)
  - For steps to implement the saga pattern by using AWS Step Functions, see the pattern Implement
    the serverless saga pattern by using AWS Step Functions on the AWS Prescriptive Guidance website.
- Shared-database-per-service pattern (p. 11)

The guide is part of a content series that covers the application modernization approach recommended
by AWS. The series also includes:

- Strategy for modernizing applications in the AWS Cloud
- Phased approach to modernizing applications in the AWS Cloud
- Evaluating modernization readiness for applications in the AWS Cloud
- Decomposing monoliths into microservices
- Integrating microservices by using AWS serverless services

**Targeted business outcomes**

Many organizations find that innovating and improving the user experience is negatively impacted
by monolithic applications, databases, and technologies. Legacy applications and databases reduce
your options for adopting modern technology frameworks, and constrain your competitiveness and
innovation. However, when you modernize applications and their data stores, they become easier to scale
and faster to develop. A decoupled data strategy improves fault tolerance and resiliency, which helps
accelerate the time to market for your new application features.

You should expect the following six outcomes from promoting data persistence among your
microservices:

- Remove legacy monolithic databases from your application portfolio.
- Improve fault tolerance, resiliency, and availability for your applications.
- Shorten your time to market for new application features.
- Reduce your overall licensing expenses and operational costs.
- Take advantage of open-source solutions (for example, MySQL or PostgreSQL).
- Build highly scalable and distributed applications by choosing from more than 15 purpose-built
database engines on the AWS Cloud.
Patterns for enabling data persistence

The following patterns are used to enable data persistence in your microservices.

Topics
- Database-per-service pattern (p. 3)
- API composition pattern (p. 5)
- CQRS pattern (p. 6)
- Event sourcing pattern (p. 7)
- Saga pattern (p. 10)
- Shared-database-per-service pattern (p. 11)

Database-per-service pattern

Loose coupling is the core characteristic of a microservices architecture, because each individual microservice can independently store and retrieve information from its own data store. By deploying the database-per-service pattern, you choose the most appropriate data stores (for example, relational or non-relational databases) for your application and business requirements. This means that microservices don't share a data layer, changes to a microservice's individual database do not impact other microservices, individual data stores cannot be directly accessed by other microservices, and persistent data is accessed only by APIs. Decoupling data stores also improves the resiliency of your overall application, and ensures that a single database can't be a single point of failure.

In the following illustration, different AWS databases are used by the "Sales," "Customer," and "Compliance" microservices. These microservices are deployed as AWS Lambda functions and accessed through an Amazon API Gateway API. AWS Identity and Access Management (IAM) policies ensure that data is kept private and not shared among the microservices. Each microservice uses a database type that meets its individual requirements; for example, "Sales" uses Amazon Aurora, "Customer" uses Amazon DynamoDB, and "Compliance" uses Amazon Relational Database Service (Amazon RDS) for SQL Server.
You should consider using this pattern if:

- Loose coupling is required between your microservices.
- Microservices have different compliance or security requirements for their databases.
- More granular control of scaling is required.

There are the following disadvantages to using the database-per-service pattern:

- It might be challenging to implement complex transactions and queries that span multiple microservices or data stores.
- You have to manage multiple relational and non-relational databases.
- Your data stores must meet two of the CAP theorem requirements: consistency, availability, or partition tolerance.
**Note**
If you use the database-per-service pattern, you must deploy the API composition pattern (p. 5) or the CQRS pattern (p. 6) to implement queries that span multiple microservices.

### API composition pattern

This pattern uses an API composer, or aggregator, to implement a query by invoking individual microservices that own the data. It then combines the results by performing an in-memory join.

The following diagram illustrates how this pattern is implemented.

The diagram shows the following workflow:

1. An API gateway serves the "/customer" API, which has an "Orders" microservice that tracks customer orders in an Aurora database.
2. The "Support" microservice tracks customer support issues and stores them in an Amazon OpenSearch Service (successor to Amazon Elasticsearch Service) database.
3. The "CustomerDetails" microservice maintains customer attributes (for example, address, phone number, or payment details) in a DynamoDB table.

4. The “GetCustomer” Lambda function runs the APIs for these microservices, and performs an in-memory join on the data before returning it to the requester. This helps easily retrieve customer information in one network call to the user-facing API, and keeps the interface very simple.

The API composition pattern offers the simplest way to gather data from multiple microservices. However, there are the following disadvantages to using the API composition pattern:

- It might not be suitable for complex queries and large datasets that require in-memory joins.
- Your overall system becomes less available if you increase the number of microservices connected to the API composer.
- Increased database requests create more network traffic, which increases your operational costs.

**CQRS pattern**

The command query responsibility segregation (CQRS) pattern separates the data mutation, or the command part of a system, from the query part. You can use the CQRS pattern to separate updates and queries if they have different requirements for throughput, latency, or consistency. The CQRS pattern splits the application into two parts: the command side and the query side. The command side handles create, update, and delete requests. The query side runs the query part by using the read replicas.

In the following illustration, a NoSQL data store, such as DynamoDB, is used to optimize the write throughput and provide flexible query capabilities. This achieves high write scalability on workloads that have well-defined access patterns when you add data. A relational database, such as Aurora, provides complex query functionality. A DynamoDB stream sends data to a Lambda function that updates the Aurora table.
You should consider using this pattern if:

- You implemented the database-per-service pattern and want to join data from multiple microservices.
- Your read and write workloads have separate requirements for scaling, latency, and consistency.
- Eventual consistency is acceptable for the read queries.

**Important**
The CQRS pattern typically results in eventual consistency between the data stores.

**Event sourcing pattern**

The event sourcing pattern is typically used with the CQRS pattern (p. 6) to decouple read from write workloads, and optimize for performance, scalability, and security. Data is stored as a series of events, instead of direct updates to data stores. Microservices replay events from an event store to compute the appropriate state of their own data stores. The pattern provides visibility for the current state of the application and additional context for how the application arrived at that state. The event
sourcing pattern works effectively with the CQRS pattern because data can be reproduced for a specific event, even if the command and query data stores have different schemas.

By choosing this pattern, you can identify and reconstruct the application's state for any point in time. This produces a persistent audit trail and makes debugging easier. However, data eventually becomes consistent and this might not be appropriate for some use cases.

This pattern can be implemented by using either Amazon Kinesis Data Streams or Amazon EventBridge.

**Amazon Kinesis Data Streams implementation**

In the following illustration, Kinesis Data Streams is the main component of a centralized event store. The event store captures application changes as events and persists them on Amazon Simple Storage Service (Amazon S3).

1. When the "/withdraw" or "/credit" microservices experience an event state change, they publish an event by writing a message into Kinesis Data Streams.
2. Other microservices, such as "/balance" or "/creditLimit," read a copy of the message, filter it for relevance, and forward it for further processing.

**Amazon EventBridge implementation**

In the following illustration, EventBridge is used as an event store. EventBridge provides a default event bus for events that are published by AWS services, and you can also create a custom event bus for domain-specific buses.
The workflow consists of the following steps:

1. "OrderPlaced" events are published by the "Orders" microservice to the custom event bus.
2. Microservices that need to take action after an order is placed, such as the "/route" microservice, are initiated by rules and targets.
3. These microservices generate a route to ship the order to the customer and emit a "RouteCreated" event.
4. Microservices that need to take further action are also initiated by the "RouteCreated" event.
5. Events are sent to an event archive (for example, Amazon S3 Glacier) so that they can be replayed for reprocessing, if required.
6. If targets are not initiated, the affected events are placed in a dead letter queue (DLQ) for further analysis and reprocessing.

You should consider using this pattern if:

- Events are used to completely rebuild the application's state.
- You require events to be replayed in the system, and that an application's state can be determined at any point in time.
- You want to be able to reverse specific events without having to start with a blank application state.
- Your system requires a stream of events that can easily be serialized to create an automated log.
- Your system requires heavy read operations but is light on write operations; heavy read operations can be directed to an in-memory database, which is kept updated with the events stream.

**Important**
If you use the event sourcing pattern, you must deploy the Saga pattern (p. 10) to maintain data consistency across microservices.
The saga pattern is a failure management pattern that helps establish consistency in distributed applications, and coordinates transactions between multiple microservices to maintain data consistency. A microservice publishes an event for every transaction, and the next transaction is initiated based on the event's outcome. It can take two different paths, depending on the success or failure of the transactions.

The following illustration shows how the saga pattern implements an order processing system by using AWS Step Functions. Each step (for example, "ProcessPayment") also has separate steps to handle the success (for example, "UpdateCustomerAccount") or failure (for example, "SetOrderFailure") of the process.

You should consider using this pattern if:

- The application needs to maintain data consistency across multiple microservices without tight coupling.
- There are long-lived transactions and you don’t want other microservices to be blocked if one microservice runs for a long time.
- You need to be able to roll back if an operation fails in the sequence.

Important

The saga pattern is difficult to debug and its complexity increases with the number of microservices. The pattern requires a complex programming model that develops and designs compensating transactions for rolling back and undoing changes.
Shared-database-per-service pattern

In the shared-database-per-service pattern, the same database is shared by several microservices. You need to carefully assess the application architecture before adopting this pattern, and make sure that you avoid hot tables (single tables that are shared among multiple microservices). All your database changes must also be backward-compatible; for example, developers can drop columns or tables only if objects are not referenced by the current and previous versions of all microservices.

In the following illustration, an insurance database is shared by all the microservices and an IAM policy provides access to the database. This creates development time coupling; for example, a change in the "Sales" microservice needs to coordinate schema changes with the "Customer" microservice. This pattern does not reduce dependencies between development teams, and introduces runtime coupling because all microservices share the same database. For example, long-running "Sales" transactions can lock the "Customer" table and this blocks the "Customer" transactions.
You should consider using this pattern if:

- You don't want too much refactoring of your existing code base.
- You enforce data consistency by using transactions that provide atomicity, consistency, isolation, and durability (ACID).
- You want to maintain and operate only one database.
- Implementing the database-per-service pattern is difficult because of interdependencies among your existing microservices.
- You don't want to completely redesign your existing data layer.
AWS Prescriptive Guidance Enabling data persistence in microservices
When can I modernize my monolithic database as part of my modernization journey?

FAQ

This section provides answers to commonly raised questions about enabling data persistence in microservices.

When can I modernize my monolithic database as part of my modernization journey?

You should focus on modernizing your monolithic database when you begin to decompose monolithic applications into microservices. Make sure that you create a strategy to split your database into multiple small databases that are aligned with your applications.

Can I keep a legacy monolithic database for multiple microservices?

Keeping a shared monolithic database for multiple microservices creates tight coupling, which means you can’t independently deploy changes to your microservices, and that all schema changes must be coordinated among your microservices. Although you can use a relational data store as your monolithic database, NoSQL databases might be a better choice for some of your microservices.

What should I consider when designing databases for a microservices architecture?

You should design your application based on domains that align with your application’s functionality. Make sure that you evaluate the application’s functionality and decide if it requires a relational database schema. You should also consider using a NoSQL database, if it fits your requirements.

What is a common pattern for maintaining data consistency across different microservices?

The most common pattern is using an event-driven architecture.

How do I maintain transaction automation?

In a microservices architecture, a transaction consists of multiple local transactions handled by different microservices. If a local transaction fails, you need to roll back the successful transactions that were previously completed. You can use the Saga pattern (p. 10) to avoid this.
**Do I have to use a separate database for each microservice?**

The main advantage of a microservices architecture is loose coupling. Each microservice’s persistent data must be kept private and accessible only through a microservice’s API. Changes to the data schema must be carefully evaluated if your microservices share the same database.

**How can I keep a microservice’s persistent data private if they all share a single database?**

If your microservices share a relational database, make sure that you have private tables for each microservice. You can also create individual schemas that are private to the individual microservices.
Resources

Related guides and patterns

- Strategy for modernizing applications in the AWS Cloud
- Phased approach to modernizing applications in the AWS Cloud
- Evaluating modernization readiness for applications in the AWS Cloud
- Decomposing monoliths into microservices
- Integrating microservices by using AWS serverless services
- Implement the serverless saga pattern by using AWS Step Functions

Other resources

- Application modernization with AWS
- Build highly available microservices to power applications of any size and scale
- Cloud-native application modernization with AWS
- Cost optimization and innovation: An introduction to application modernization
- Developer guide: Scale with microservices
- Distributed data management - Saga Pattern
- Implementing microservice architectures using AWS services: Command query responsibility segregation pattern
- Implementing microservice architectures using AWS services: Event sourcing pattern
- Modern applications: Creating value through application design
- Modernize your applications, drive growth and reduce TCO
The following are commonly used terms in artificial intelligence (AI) and machine learning (ML)-related strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the Provide feedback link at the end of the glossary.

**binary classification**
A process that predicts a binary outcome (one of two possible classes). For example, your ML model might need to predict problems such as “Is this email spam or not spam?” or “Is this product a book or a car?”

**classification**
A categorization process that helps generate predictions. ML models for classification problems predict a discrete value. Discrete values are always distinct from one another. For example, a model might need to evaluate whether or not there is a car in an image.

**data preprocessing**
To transform raw data into a format that is easily parsed by your ML model. Preprocessing data can mean removing certain columns or rows and addressing missing, inconsistent, or duplicate values.

**deep ensemble**
To combine multiple deep learning models for prediction. You can use deep ensembles to obtain a more accurate prediction or for estimating uncertainty in predictions.

**deep learning**
An ML subfield that uses multiple layers of artificial neural networks to identify mapping between input data and target variables of interest.

**exploratory data analysis (EDA)**
The process of analyzing a dataset to understand its main characteristics. You collect or aggregate data and then perform initial investigations to find patterns, detect anomalies, and check assumptions. EDA is performed by calculating summary statistics and creating data visualizations.

**features**
The input data that you use to make a prediction. For example, in a manufacturing context, features could be images that are periodically captured from the manufacturing line.

**feature transformation**
To optimize data for the ML process, including enriching data with additional sources, scaling values, or extracting multiple sets of information from a single data field. This enables the ML model to benefit from the data. For example, if you break down the "2021-05-27 00:15:37" date into “2021”, “May”, “Thu”, and “15”, you can help the learning algorithm learn nuanced patterns associated with different data components.
multiclass classification: A process that helps generate predictions for multiple classes (predicting one of more than two outcomes). For example, an ML model might ask "Is this product a book, car, or phone?" or "Which product category is most interesting to this customer?"

regression: An ML technique that predicts a numeric value. For example, to solve the problem of "What price will this house sell for?" an ML model could use a linear regression model to predict a house's sale price based on known facts about the house (for example, the square footage).

training: To provide data for your ML model to learn from. The training data must contain the correct answer. The learning algorithm finds patterns in the training data that map the input data attributes to the target (the answer that you want to predict). It outputs an ML model that captures these patterns. You can then use the ML model to make predictions on new data for which you don't know the target.

target variable: The value that you are trying to predict in supervised ML. This is also referred to as an outcome variable. For example, in a manufacturing setting the target variable could be a product defect.

tuning: To change aspects of your training process to improve the ML model's accuracy. For example, you can train the ML model by generating a labeling set, adding labels, and then repeating these steps several times under different settings to optimize the model.

uncertainty: A concept that refers to imprecise, incomplete, or unknown information that can undermine the reliability of predictive ML models. There are two types of uncertainty: Epistemic uncertainty is caused by limited, incomplete data, whereas aleatoric uncertainty is caused by the noise and randomness inherent in the data. For more information, see the Quantifying uncertainty in deep learning systems guide.

Migration terms:

The following are commonly used terms in migration-related strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the Provide feedback link at the end of the glossary.

7 Rs: Seven common migration strategies for moving applications to the cloud. These strategies build upon the 5 Rs that Gartner identified in 2011 and consist of the following:

- Refactor/re-architect – Move an application and modify its architecture by taking full advantage of cloud-native features to improve agility, performance, and scalability. This typically involves porting the operating system and database. Example: Migrate your on-premises Oracle database to the Amazon Aurora PostgreSQL-Compatible Edition.
- Replatform (lift and reshape) – Move an application to the cloud, and introduce some level of optimization to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Amazon Relational Database Service (Amazon RDS) for Oracle in the AWS Cloud.
- Repurchase (drop and shop) – Switch to a different product, typically by moving from a traditional license to a SaaS model. Example: Migrate your customer relationship management (CRM) system to Salesforce.com.
- Rehost (lift and shift) – Move an application to the cloud without making any changes to take advantage of cloud capabilities. Example: Migrate your on-premises Oracle database to Oracle on an EC2 instance in the AWS Cloud.
• Relocate (hypervisor-level lift and shift) – Move infrastructure to the cloud without purchasing new hardware, rewriting applications, or modifying your existing operations. This migration scenario is specific to VMware Cloud on AWS, which supports virtual machine (VM) compatibility and workload portability between your on-premises environment and AWS. You can use the VMware Cloud Foundation technologies from your on-premises data centers when you migrate your infrastructure to VMware Cloud on AWS. Example: Relocate the hypervisor hosting your Oracle database to VMware Cloud on AWS.

• Retain (revisit) – Keep applications in your source environment. These might include applications that require major refactoring, and you want to postpone that work until a later time, and legacy applications that you want to retain, because there’s no business justification for migrating them.

• Retire – Decommission or remove applications that are no longer needed in your source environment.

application portfolio
A collection of detailed information about each application used by an organization, including the cost to build and maintain the application, and its business value. This information is key to the portfolio discovery and analysis process and helps identify and prioritize the applications to be migrated, modernized, and optimized.

artificial intelligence operations (AIOps)
The process of using machine learning techniques to solve operational problems, reduce operational incidents and human intervention, and increase service quality. For more information about how AIOps is used in the AWS migration strategy, see the operations integration guide.

AWS Cloud Adoption Framework (AWS CAF)
A framework of guidelines and best practices from AWS to help organizations develop an efficient and effective plan to move successfully to the cloud. AWS CAF organizes guidance into six focus areas called perspectives: business, people, governance, platform, security, and operations. The business, people, and governance perspectives focus on business skills and processes; the platform, security, and operations perspectives focus on technical skills and processes. For example, the people perspective targets stakeholders who handle human resources (HR), staffing functions, and people management. For this perspective, AWS CAF provides guidance for people development, training, and communications to help ready the organization for successful cloud adoption. For more information, see the AWS CAF website and the AWS CAF whitepaper.

AWS landing zone
A landing zone is a well-architected, multi-account AWS environment that is scalable and secure. This is a starting point from which your organizations can quickly launch and deploy workloads and applications with confidence in their security and infrastructure environment. For more information about landing zones, see Setting up a secure and scalable multi-account AWS environment.

AWS Workload Qualification Framework (AWS WQF)
A tool that evaluates database migration workloads, recommends migration strategies, and provides work estimates. AWS WQF is included with AWS Schema Conversion Tool (AWS SCT). It analyzes database schemas and code objects, application code, dependencies, and performance characteristics, and provides assessment reports.

business continuity planning (BCP)
A plan that addresses the potential impact of a disruptive event, such as a large-scale migration, on operations and enables a business to resume operations quickly.

Cloud Center of Excellence (CCoE)
A multi-disciplinary team that drives cloud adoption efforts across an organization, including developing cloud best practices, mobilizing resources, establishing migration timelines, and leading the organization through large-
<table>
<thead>
<tr>
<th><strong>cloud stages of adoption</strong></th>
<th>The four phases that organizations typically go through when they migrate to the AWS Cloud:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Project – Running a few cloud-related projects for proof of concept and learning purposes</td>
<td></td>
</tr>
<tr>
<td>• Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model)</td>
<td></td>
</tr>
<tr>
<td>• Migration – Migrating individual applications</td>
<td></td>
</tr>
<tr>
<td>• Re-invention – Optimizing products and services, and innovating in the cloud</td>
<td></td>
</tr>
</tbody>
</table>

These stages were defined by Stephen Orban in the blog post *The Journey Toward Cloud-First & the Stages of Adoption* on the AWS Cloud Enterprise Strategy blog. For information about how they relate to the AWS migration strategy, see the [migration readiness guide](#).

| **configuration management database (CMDB)** | A database that contains information about a company’s hardware and software products, configurations, and inter-dependencies. You typically use data from a CMDB in the portfolio discovery and analysis stage of migration. |
| **epic** | In agile methodologies, functional categories that help organize and prioritize your work. Epics provide a high-level description of requirements and implementation tasks. For example, AWS CAF security epics include identity and access management, detective controls, infrastructure security, data protection, and incident response. For more information about epics in the AWS migration strategy, see the [program implementation guide](#). |
| **heterogeneous database migration** | Migrating your source database to a target database that uses a different database engine (for example, Oracle to Amazon Aurora). Heterogeneous migration is typically part of a re-architecting effort, and converting the schema can be a complex task. AWS provides [AWS SCT](#) that helps with schema conversions. |
| **homogeneous database migration** | Migrating your source database to a target database that shares the same database engine (for example, Microsoft SQL Server to Amazon RDS for SQL Server). Homogeneous migration is typically part of a rehosting or replatforming effort. You can use native database utilities to migrate the schema. |
| **IT information library (ITIL)** | A set of best practices for delivering IT services and aligning these services with business requirements. ITIL provides the foundation for ITSM. |
| **IT service management (ITSM)** | Activities associated with designing, implementing, managing, and supporting IT services for an organization. For information about integrating cloud operations with ITSM tools, see the [operations integration guide](#). |
| **Migration Acceleration Program (MAP)** | An AWS program that provides consulting support, training, and services to help organizations build a strong operational foundation for moving to the cloud, and to help offset the initial cost of migrations. MAP includes a migration methodology for executing legacy migrations in a methodical way and a set of tools to automate and accelerate common migration scenarios. |
| **Migration Portfolio Assessment (MPA)** | An online tool that provides information for validating the business case for migrating to the AWS Cloud. MPA provides detailed portfolio assessment (server right-sizing, pricing, TCO comparisons, migration cost analysis) as well as migration planning (application data analysis and data collection, application grouping, migration prioritization, and wave planning). The [MPA tool](#) (requires... |
Migration Readiness Assessment (MRA) is available free of charge to all AWS consultants and APN Partner consultants.

Migration Readiness Assessment (MRA) is the process of gaining insights about an organization's cloud readiness status, identifying strengths and weaknesses, and building an action plan to close identified gaps, using the AWS CAF. For more information, see the migration readiness guide. MRA is the first phase of the AWS migration strategy.

Migration at scale is the process of moving the majority of the application portfolio to the cloud in waves, with more applications moved at a faster rate in each wave. This phase uses the best practices and lessons learned from the earlier phases to implement a migration factory of teams, tools, and processes to streamline the migration of workloads through automation and agile delivery. This is the third phase of the AWS migration strategy.

Migration factory is cross-functional teams that streamline the migration of workloads through automated, agile approaches. Migration factory teams typically include operations, business analysts and owners, migration engineers, developers, and DevOps professionals working in sprints. Between 20 and 50 percent of an enterprise application portfolio consists of repeated patterns that can be optimized by a factory approach. For more information, see the discussion of migration factories and the CloudEndure Migration Factory guide in this content set.

Operational-level agreement (OLA) is an agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).

Operations integration (OI) is the process of modernizing operations in the cloud, which involves readiness planning, automation, and integration. For more information, see the operations integration guide.

Organizational change management (OCM) is a framework for managing major, disruptive business transformations from a people, culture, and leadership perspective. OCM helps organizations prepare for, and transition to, new systems and strategies by accelerating change adoption, addressing transitional issues, and driving cultural and organizational changes. In the AWS migration strategy, this framework is called people acceleration, because of the speed of change required in cloud adoption projects. For more information, see the OCM guide.

Playbook is a set of predefined steps that capture the work associated with migrations, such as delivering core operations functions in the cloud. A playbook can take the form of scripts, automated runbooks, or a summary of processes or steps required to operate your modernized environment.

Responsible, accountable, consulted, informed (RACI) matrix is a matrix that defines and assigns roles and responsibilities in a project. For example, you can create a RACI to define security control ownership or to identify roles and responsibilities for specific tasks in a migration project.

Runbook is a set of manual or automated procedures required to perform a specific task. These are typically built to streamline repetitive operations or procedures with high error rates.

Service-level agreement (SLA) is an agreement that clarifies what an IT team promises to deliver to their customers, such as service uptime and performance.

Modernization terms

The following are commonly used terms in modernization-related strategies, guides, and patterns provided by AWS Prescriptive Guidance. To suggest entries, please use the Provide feedback link at the end of the glossary.
business capability
What a business does to generate value (for example, sales, customer service, or marketing). Microservices architectures and development decisions can be driven by business capabilities. For more information, see the Organized around business capabilities section of the Running containerized microservices on AWS whitepaper.

microservice
A small, independent service that communicates over well-defined APIs and is typically owned by small, self-contained teams. For example, an insurance system might include microservices that map to business capabilities, such as sales or marketing, or subdomains, such as purchasing, claims, or analytics. The benefits of microservices include agility, flexible scaling, easy deployment, reusable code, and resilience. For more information, see Integrating microservices by using AWS serverless services.

microservices architecture
An approach to building an application with independent components that run each application process as a microservice. These microservices communicate through a well-defined interface by using lightweight APIs. Each microservice in this architecture can be updated, deployed, and scaled to meet demand for specific functions of an application. For more information, see Implementing microservices on AWS.

modernization
Transforming an outdated (legacy or monolithic) application and its infrastructure into an agile, elastic, and highly available system in the cloud to reduce costs, gain efficiencies, and take advantage of innovations. For more information, see Strategy for modernizing applications in the AWS Cloud.

modernization readiness assessment
An evaluation that helps determine the modernization readiness of an organization's applications; identifies benefits, risks, and dependencies; and determines how well the organization can support the future state of those applications. The outcome of the assessment is a blueprint of the target architecture, a roadmap that details development phases and milestones for the modernization process, and an action plan for addressing identified gaps. For more information, see Evaluating modernization readiness for applications in the AWS Cloud.

monolithic applications (monoliths)
Applications that run as a single service with tightly coupled processes. Monolithic applications have several drawbacks. If one application feature experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features also becomes more complex when the code base grows. To address these issues, you can use a microservices architecture. For more information, see Decomposing monoliths into microservices.

polyglot persistence
Independently choosing a microservice's data storage technology based on data access patterns and other requirements. If your microservices have the same data storage technology, they can encounter implementation challenges or experience poor performance. Microservices are more easily implemented and achieve better performance and scalability if they use the data store best adapted to their requirements. For more information, see Enabling data persistence in microservices.

split-and-seed model
A pattern for scaling and accelerating modernization projects. As new features and product releases are defined, the core team splits up to create new product teams. This helps scale your organization's capabilities and services, improves developer productivity, and supports rapid innovation. For more information, see Phased approach to modernizing applications in the AWS Cloud.

two-pizza team
A small DevOps team that you can feed with two pizzas. A two-pizza team size ensures the best possible opportunity for collaboration in software development.
For more information, see the Two-pizza team section of the Introduction to DevOps on AWS whitepaper.
Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

<table>
<thead>
<tr>
<th>update-history-change</th>
<th>update-history-description</th>
<th>update-history-date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Added a link for implementing the saga pattern with Step Functions (p. 23)</td>
<td>We updated the Home and Saga pattern sections with the link to the pattern Implement the serverless saga pattern by using AWS Step Functions from the AWS Prescriptive Guidance website.</td>
<td>February 23, 2021</td>
</tr>
<tr>
<td>Initial publication (p. 23)</td>
<td>—</td>
<td>January 27, 2021</td>
</tr>
</tbody>
</table>