AWS Prescriptive Guidance

Integrating microservices by using serverless services

AWS Prescriptive Guidance: Integrating microservices by using serverless services

Table of Contents

Integrating microservices by using AWS serverless services

Hari Ohm Prasath Rajagopal, Senior Modernization Architect, Amazon Web Services

Tabby Ward, Senior Modernization Architect, Amazon Web Services

Dmitry Gulin, Modernization Architect, Amazon Web Services

January 2021

An important part of modernizing your organization's software is to refactor your monolithic applications into microservices. After you decompose a monolith, several microservices are called to fetch data for one business transaction. If these microservices are incorrectly integrated into your architecture, the benefits of adopting a microservices architecture are undermined. This can cause data loss, or latency and integrity issues. These problems are often hard to resolve, and users are immediately impacted. However, if microservices are correctly integrated, they provide the benefits of distributed systems, help scaling at the service level, improve efficiency, and reduce your infrastructure costs.

This guide is for application owners, business owners, architects, technical leads, and project managers. The guide provides the following three patterns to help integrate new microservices into your architecture:

- API gateway pattern (p. 2)
- Decouple messaging pattern (p. 4)
- Pub/sub pattern (p. 5)

These patterns offer autonomy and scalability, and use serverless services from Amazon Web Services (AWS), such as AWS Lambda and Amazon API Gateway, to help integrate your microservices. 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
- Enabling data persistence for microservices

Targeted business outcomes

By using this guide to integrate your new microservices, you can efficiently transform your organization's architecture into a microservices architecture. This helps provide rapid adjustment to fluctuating business needs without interrupting core activities such as high scalability, improved resiliency, continuous delivery, and failure isolation. A microservices architecture also helps improve fault tolerance and resiliency, and speeds up innovation because each microservice can be individually deployed and tested.

A microservices architecture can also help provide a shorter time to market for your products or services, because each microservice has an independent code base that makes it easier and faster to add new features and iterate on them.

Patterns for integrating microservices

The following patterns are used to integrate microservices into your architecture.

Topics

API gateway pattern

The API gateway pattern is recommended if you want to design and build complex or large microservices-based applications with multiple client applications. The pattern is similar to the facade pattern from object-oriented design, but it is part of a distributed system reverse proxy or gateway routing, and uses a synchronous communication model.

The pattern provides a reverse proxy to redirect or route requests to your internal microservices endpoints. An API gateway provides a single endpoint or URL for the client applications, and it internally maps the requests to internal microservices. A layer of abstraction is provided by hiding certain implementation details (for example, the Lambda function name and version), and additional functionality can also be added on top of the backend service, such as response and request transformations, endpoint access authorization, or tracing.

You should consider using the API gateway pattern if:

- The number of dependencies for a microservice is manageable and does not grow over time.
- The calling system requires a synchronous response from the microservice.
- You have low latency requirements.
- You need to expose one API to collect data from multiple microservices.

Use case

In this use case, a customer makes regular monthly payments in an insurance system that consists of four microservices deployed as Lambda functions ("Customer," "Communication," "Payments," and "Sales"). The “Customer” microservice updates the customer database with the monthly payment details. The “Sales” microservice updates the sales database with relevant information that helps the sales team follow up with the customer for cross-selling opportunities. The “Communication” microservice sends a confirmation email to the customer after the payment is successfully processed. Finally, the “Payments” microservice is the overall system that the customer uses to make their monthly payment. The pattern uses web services to integrate the "Customer," "Sales," and "Communication" subsystems with the "Payments" microservice.

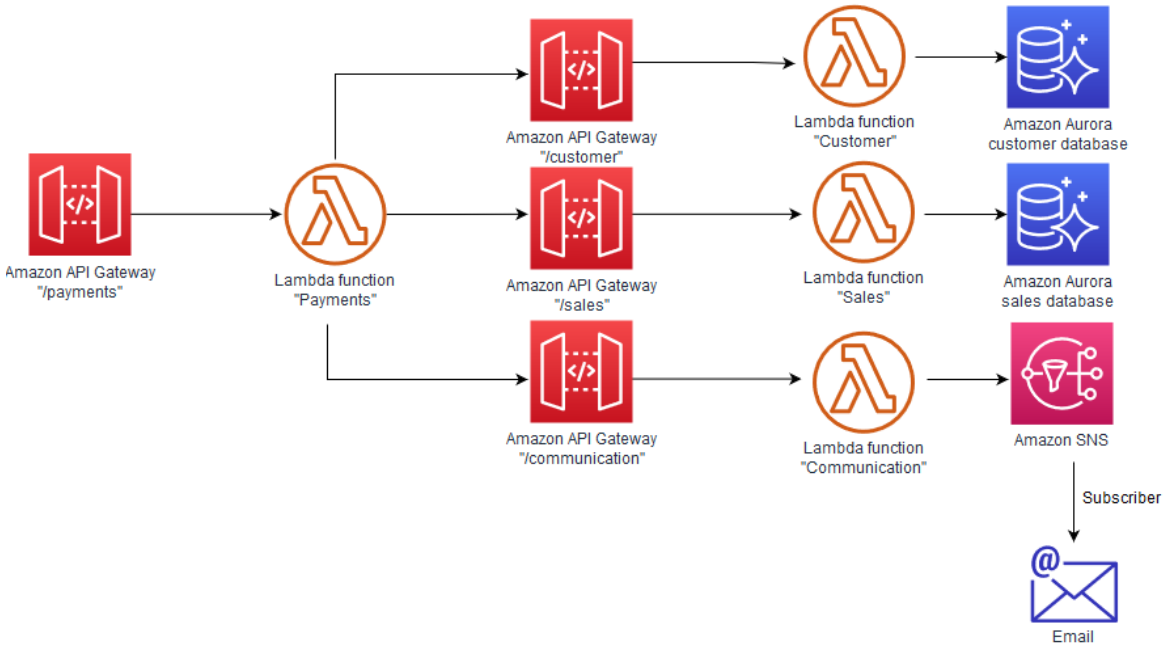
There are three challenges to using this pattern for this use case:

- Synchronous calls are made to downstream systems, which means that any latency caused by these subsystems affects the overall response time.
- Running costs are higher because the “Payments” system is waiting for responses from the other microservices before responding to the calling system. The total running time is therefore relatively higher compared with an asynchronous system.
- Error handling and retry are handled separately for each microservice inside the “Payments” system, not by the individual microservices.

The following two examples outline how to use the API gateway pattern to integrate microservices by using multiple API gateways or one API gateway.

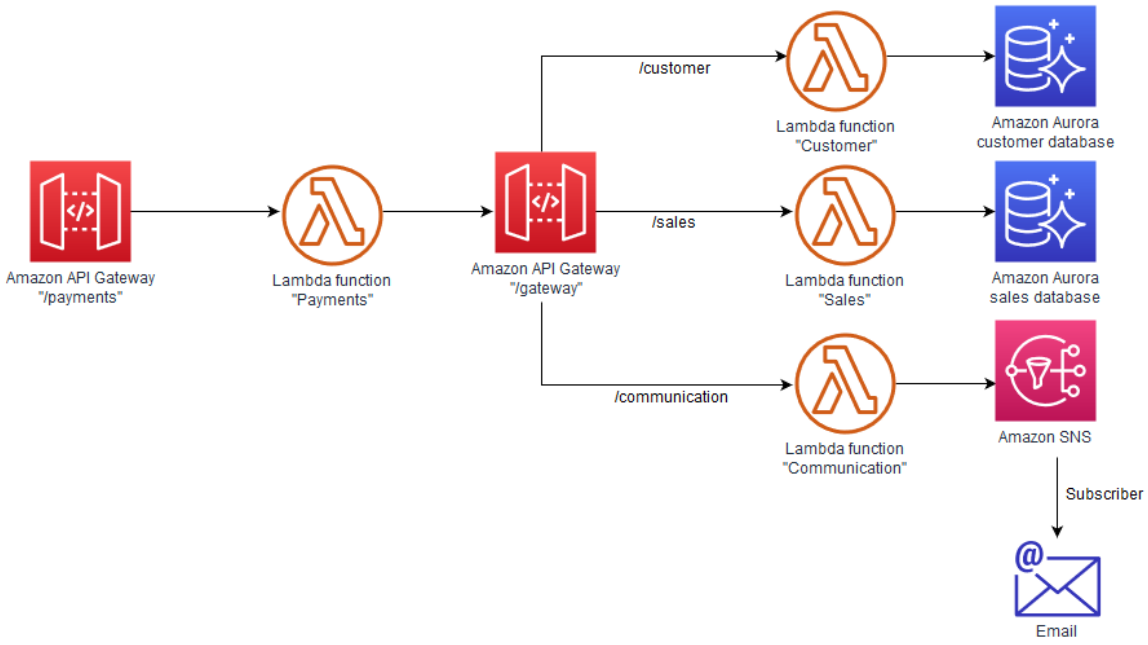
Multiple API gateways

In the following illustration, each microservice has its own API gateway. The "Payments" microservice calls out individual systems and implements the API gateway pattern.



Single API gateway

In the following illustration, each microservice is deployed as a Lambda function but all microservices are connected by the same API gateway.



Decouple messaging pattern

This pattern provides asynchronous communication between microservices by using an asynchronous poll model. When the backend system receives a call, it immediately responds with a request identifier and then asynchronously processes the request. A loosely coupled architecture can be built, which avoids bottlenecks caused by synchronous communication, latency, and input/output operations (IO). In the pattern's use case, Amazon Simple Queue Service (Amazon SQS) and Lambda are used to implement asynchronous communication between different microservices.

You should consider using this pattern if:

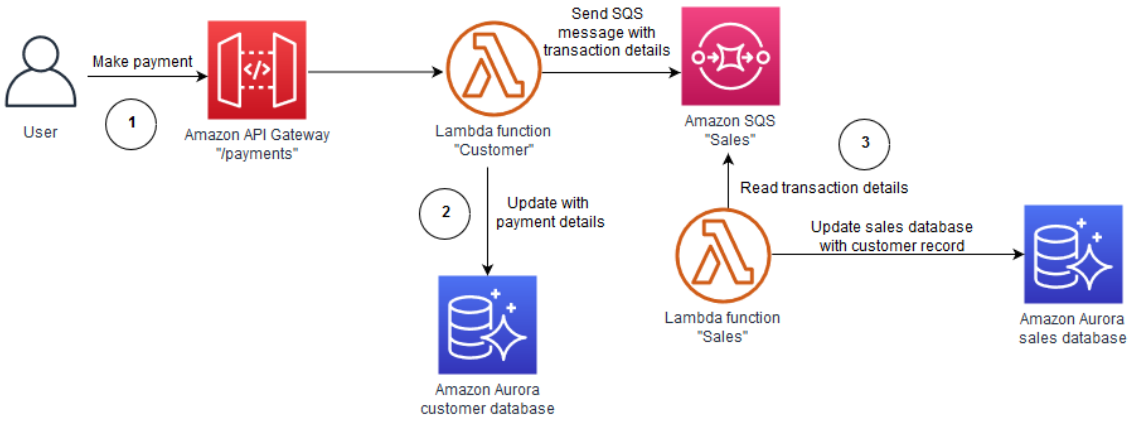
- You want to create loosely coupled architecture.
- All operations don't need to be completed in a single transaction, and some operations can be asynchronous.
- The downstream system cannot handle the incoming transactions per second (TPS) rate. The messages can be written to the queue and processed based on the availability of resources.

A disadvantage of this pattern is that business transaction actions are synchronous. Even though the calling system receives a response, some part of the transaction might still continue to be processed by downstream systems.

> **Important**
> Because this pattern is more suitable for a fire-and-forget model, the client calling this service should poll the actual service by using a request ID to get the transaction status.

Use case

In this use case, the insurance system has a sales database that is automatically updated with the customer transaction details after a monthly payment is made. The following illustration shows how to build this system by using the decouple messaging pattern.



The workflow consists of the following steps:

1. The frontend application calls the API Gateway with the payment information after a user makes their monthly payment.
2. The API Gateway runs the “Customer” Lambda function that saves the payment information in an Amazon Aurora database, writes the transaction details in a message to the “Sales" Amazon SQS, and responds to the calling system with a success message.
3. A “Sales” Lambda function pulls the transaction details from the SQS message and updates the sales data. Failure and retry logic to update the sales database is incorporated as part of the “Sales” Lambda function.

Pub/sub pattern

When a platform grows, it can be difficult for different microservices to interact without creating interdependency. The publish/subscribe (pub/sub) pattern provides asynchronous communication among multiple AWS services, such as Amazon SQS, Lambda or Amazon Simple Storage Service (Amazon S3), without creating interdependency. In this pattern, microservices publish events as messages in a channel that subscribers can listen to. For example, a factory can use a pub/sub pattern to enable equipment to publish problems or failures to a channel, a subscriber can then display and log these equipment issues.

You should consider using this pattern if:

- You have an event-driven architecture.
- You can enable loosely coupled architecture.
- You don't need to complete all operational parts of a transaction before the response back to the calling system (certain operations can be asynchronous).
- You need to scale to volumes that are beyond the capability of a traditional data center. This level of scalability is primarily due to parallel operations, message caching, tree-based routing, and other features built into the pub/sub model.

There are several disadvantages to using this pattern. For example, the pub/sub pattern typically cannot guarantee delivery of messages to all subscriber types, although certain services such as Amazon Simple Notification Service (Amazon SNS) can provide exactly-once delivery to some

subscriber subsets. Another disadvantage is that a publisher could assume that a subscriber is listening to a channel when, in fact, they are not.

Use case

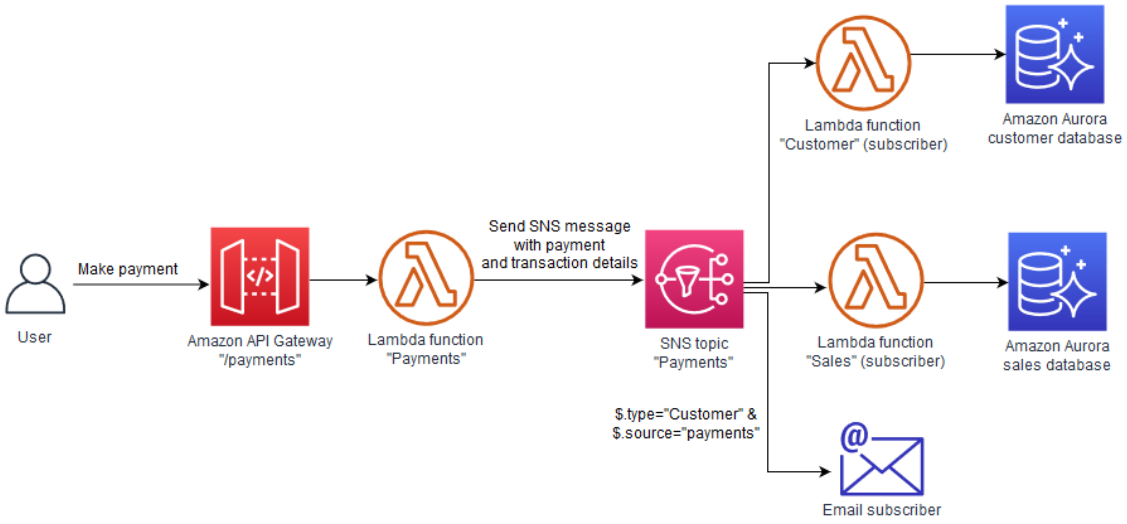
In this use case, an SNS topic is used to publish events to several dependent microservices in an insurance system. After a customer makes their monthly payment, the information must be updated in subsystems such as “Customer” or “Sales,” and an email must be sent to the customer with the payment confirmation. This pattern can be implemented by using either Amazon SNS or Amazon EventBridge.

EventBridge filters events between multiple subscribers. The EventBridge implementation provides the following two options:

- Send three events with different event types. The distant target picks them up based on the event rule.
- Send one message with three event rules listening to the same event type. Unnecessary data is filtered out before specific targets are invoked.

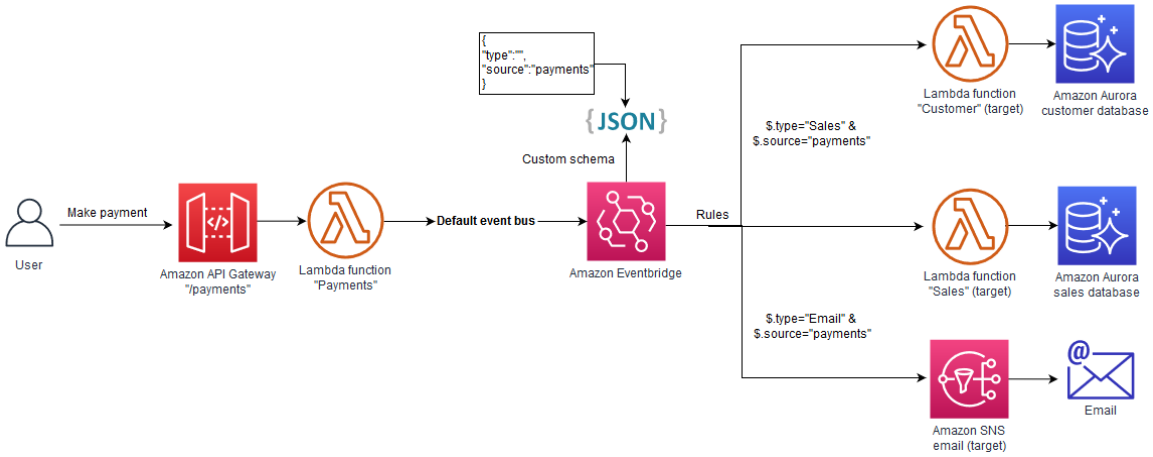
Amazon SNS implementation

The following illustration shows how Amazon SNS is used to implement the pub/sub pattern. After a user makes a payment, an SNS message is sent by the “Payments” Lambda function to the "Payments" SNS topic. This SNS topic has three subscribers that receive a copy of the message and process it.



Amazon EventBridge implementation

In the following illustration, EventBridge is used to implement the pub/sub pattern. After a user makes a payment, the “Payments” Lambda function sends a message to EventBridge by using the default event bus based on a custom schema that has three different rules pointing to different targets. Each microservice processes the messages and performs the required actions.


{
"type":"",
"source":"payments"
}
{JSON}
Custom schema
User
Make payment
Amazon API Gateway
"/payments"
Lambda function
"Payments"
Default event bus
Amazon Eventbridge
Rules
$.type="Sales" &
$.source="payments"
$.type="Email" &
$.source="payments"
Lambda function
"Customer" (target)
Amazon Aurora
customer database
Lambda function
"Sales" (target)
Amazon Aurora
sales database
Amazon SNS
email (target)
Email

Integrating microservices FAQ

This section provides answers to commonly raised questions about integrating microservices.

Can integration patterns be combined?

Yes, the API gateway pattern (p. 2) and decouple messaging pattern (p. 4) are typically used together when integrating microservices.

Resources

Related guides

- 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
- Enabling data persistence in microservices

Other resources

- Implementing enterprise integration patterns with AWS messaging services: point-to-point channels
- Pub/Sub messaging: Asynchronous event notifications

AWS Prescriptive Guidance glossary

AI and ML terms (p. 10) | Migration terms (p. 11) | Modernization terms (p. 14)

AI and ML terms

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-

	scale transformations. For more information, see the CCoE posts on the AWS Cloud Enterprise Strategy Blog.
cloud stages of adoption	The four phases that organizations typically go through when they migrate to the AWS Cloud: • Project – Running a few cloud-related projects for proof of concept and learning purposes • Foundation – Making foundational investments to scale your cloud adoption (e.g., creating a landing zone, defining a CCoE, establishing an operations model) • Migration – Migrating individual applications • Re-invention – Optimizing products and services, and innovating in the cloud 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

	login) is available free of charge to all AWS consultants and APN Partner consultants.
Migration Readiness Assessment (MRA)	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	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	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)	An agreement that clarifies what functional IT groups promise to deliver to each other, to support a service-level agreement (SLA).
operations integration (OI)	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)	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	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	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	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)	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.

update-history-change	update-history-description	update-history-date
Initial publication (p. 17)	—	January 11, 2021