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, Tabby Ward, and Dmitry Gulin, Amazon Web Services (AWS)

January 2021 (document history (p. 10))

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.

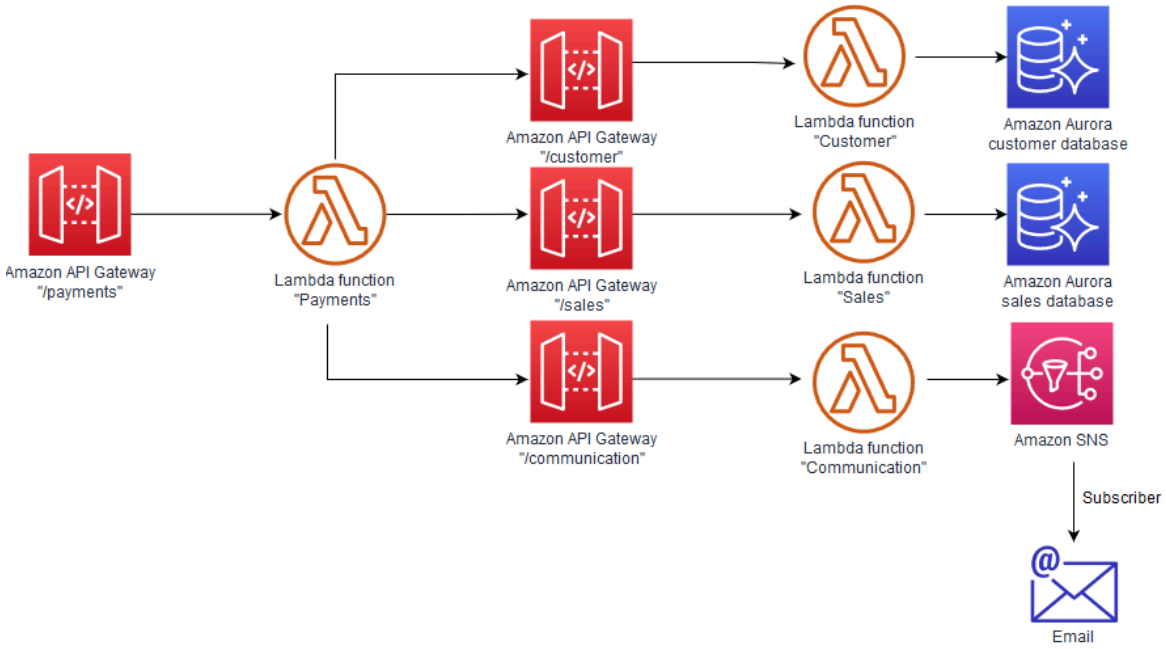
There are three challenges to using this pattern for this use case:

- 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.

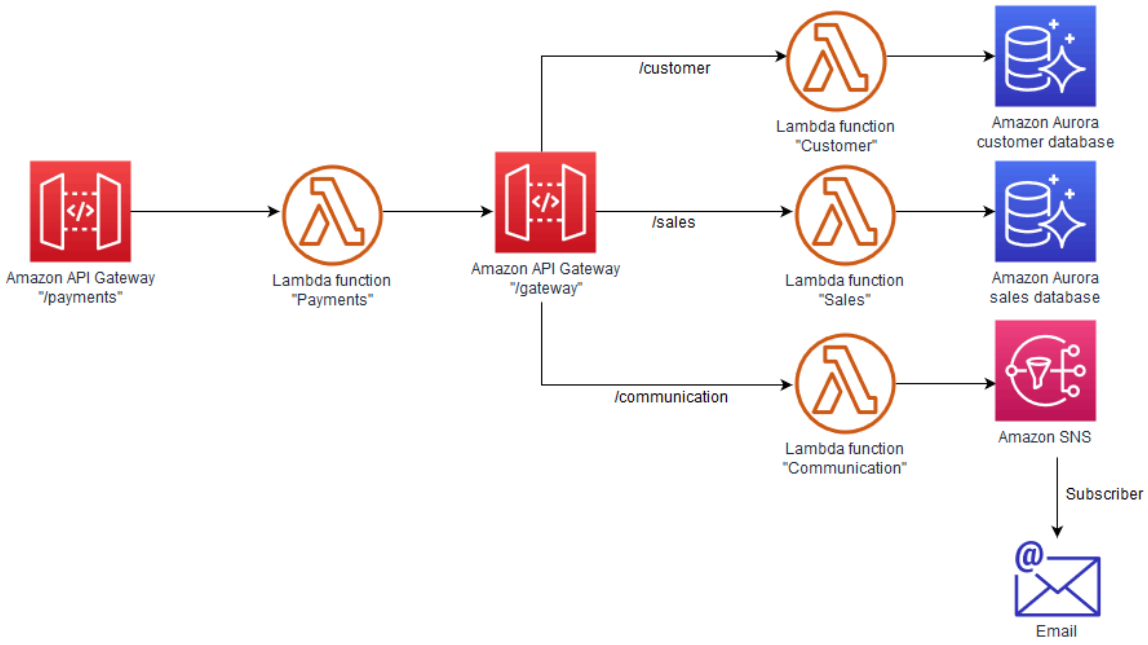
Multiple API gateways

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:

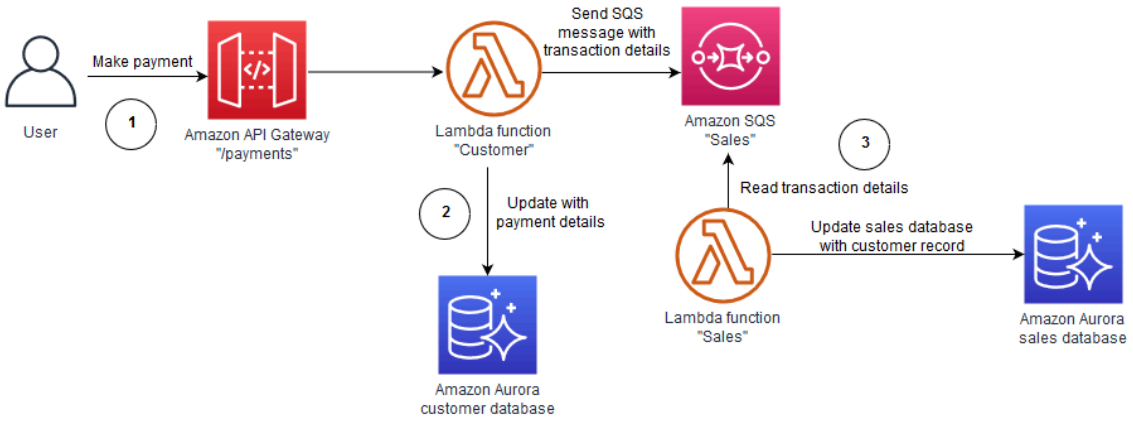
- You want to create loosely coupled architecture.
- All operations don't need to be completed in a single transaction, and some operations can be asynchronous.
- The downstream system cannot handle the incoming transactions per second (TPS) rate. The messages can be written to the queue and processed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

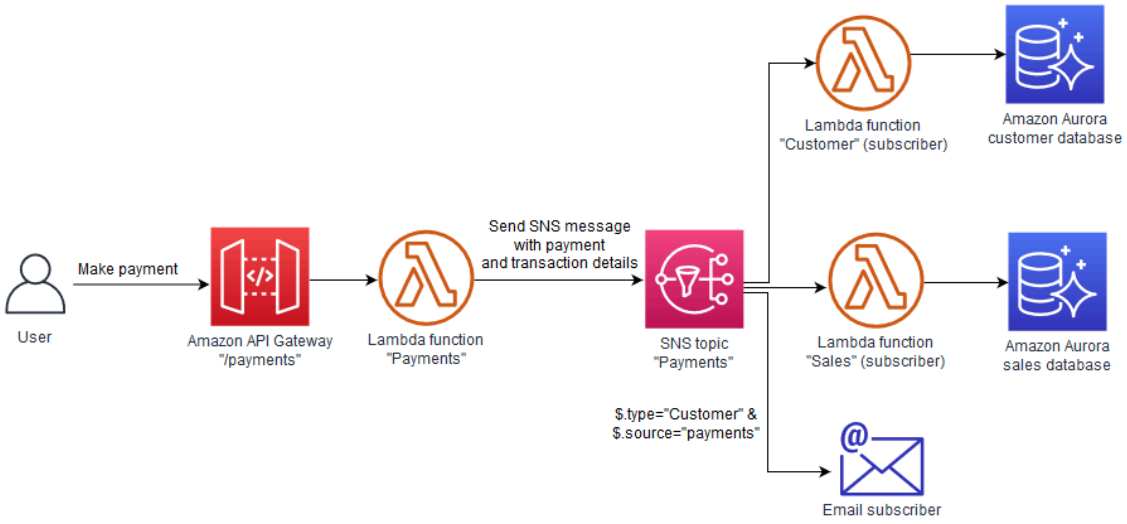
In this use case, an SNS topic is used to publish events to several dependent microservices in an insurance system. After a customer makes their monthly payment, the information must be updated in subsystems such as “Customer” or “Sales,” and an email must be sent to the customer with the payment confirmation.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

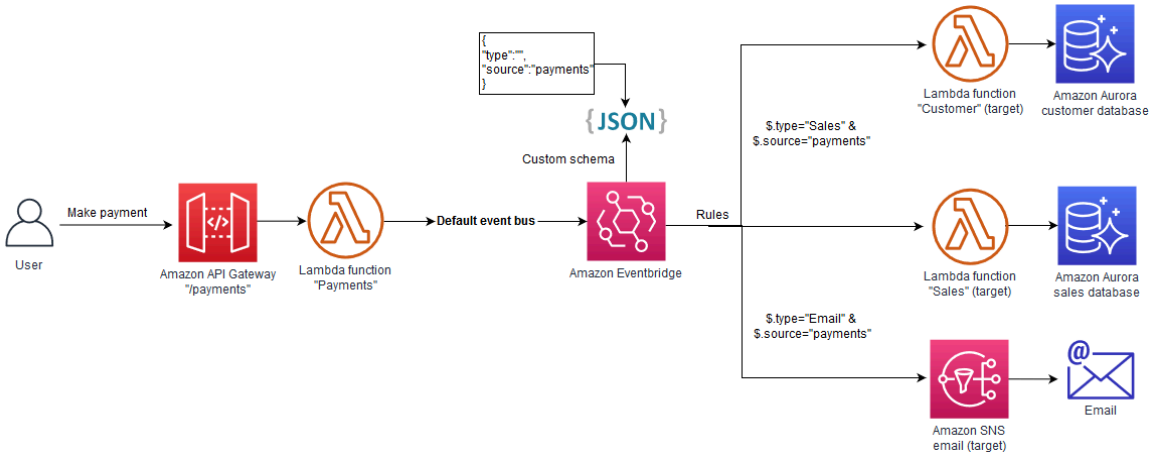
Amazon SNS implementation

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 build a version of the pub/sub pattern in which subscribers are defined by using event rules. 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"
Lambda function
"Customer" (target)
Amazon Aurora
customer database
Lambda function
"Sales" (target)
Amazon Aurora
sales database
$.type="Email" &
$.source="payments"
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

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.

Change	Description	Date
Initial publication (p. 10)	—	January 11, 2021

AWS Prescriptive Guidance glossary

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

Modernization terms

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.

domain-driven design

An approach to developing a complex software system by connecting its components to evolving domains, or core business goals, that each component serves. This concept was introduced by Eric Evans in his book, *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Boston: Addison-Wesley Professional, 2003). For information about how you can use domain-driven design with the strangler fig pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

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.

strangler fig pattern

An approach to modernizing monolithic systems by incrementally rewriting and replacing system functionality until the legacy system can be decommissioned. This pattern uses the analogy of a fig vine that grows into an established tree and eventually overcomes and replaces its host. The pattern was introduced by Martin Fowler as a way to manage risk when rewriting monolithic systems. For an example of how to apply this pattern, see Modernizing legacy Microsoft ASP.NET (ASMX) web services incrementally by using containers and Amazon API Gateway.

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.