A Deep Dive into Decoupled Cloud Architectures with Event Buses

2 February 2024 - 10 min. read

Decoupled Architectures | *Event Buses* | *Microservices*

For companies trying to stay ahead in the constantly evolving cloud computing world, scalability, resilience, and flexibility are essential requirements for their systems. Organizations are transitioning from monolithic architectures to distributed systems to benefit from their power.

In this process of breaking down into pieces your monolith application to shift to a distributed system, decoupling strategies, and patterns become crucial to achieving this goal in the right way.

Decoupling is a strategic change that divides components within a system, making them operate independently while promoting interoperability.

This article was inspired by the last AWS re:Invent session: "Advanced Integration Patterns & trade-offs for loosely coupled systems.

In this article, we will explore decoupled cloud architectures, focusing on the function of event buses in this process. Event buses act as a foundation for coordination and communication of applications throughout a distributed system.

This article will explain the complexities of modern cloud architecture to veteran technicians and business leaders who wish to grasp the foundations. It will also highlight the concrete benefits of adopting a decoupled approach.

From a monolithic application to a plethora of microservices: Decoupling in the Cloud Context

Fragmenting your monolith application into several microservices may seem very challenging initially, but it comes with several benefits that can be summarized with "components independence". To name just a few of those benefits:

- **Scalability**: components can scale in/out depending on their usage independently. This usually results in better resource utilization and lower costs. This also is a key for the high availability of the entire system.
- **Resilience**: a single microservice failure will less likely spread to the entire system, creating a major failure.
- **Maintainability**: microservices, as their name suggests, are usually smaller components. In terms of the code base, the maintenance and upgrade become easier, resulting in faster and more consistent updates. Moreover, microservices are also independent from the programming language/technology point of view, therefore, their development can be parallelized across multiple teams. This can tremendously increase the overall product and feature development speed.
- **Testability**: another aspect of microservices' small code basis is that testing the code becomes easier. Developers need to write unit tests just for their microservice, usually resulting in more tests written and better code coverage that helps with software robustness.

This independence is crucial in the dynamic and elastic cloud environment where agility, coupled with the scalability provided by AWS services, really helps to rapidly develop and improve the application, thus all the aforementioned benefits.

Loosely coupled components

A common design pattern is "**Loosely coupled components**": components should be designed to operate independently from each other, interacting with minimal or no dependency. In this sense, components have minimal knowledge of each other, they just know how to exchange information between them via well-defined interfaces or contracts, which standardize the communication: message structures, data formats, and so on...

As hinted before, coupling can be found in many different aspects, to name just a few: from technology coupling, where applications share the same programming language or technology, to communication and conversation coupling: is the communication synchronous or asynchronous? How do we communicate: full results, pagination, caching? What about retries?

Given the scope of this article, which focuses on the communication between systems, we will concentrate on related decoupling strategies.

Event Buses: the backbone of cloud distributed architectures

Speaking about communication between systems, event buses are the perfect solution to achieve decoupling.

In cloud architectures, systems communicate and integrate with each other using events. Event buses can act as an interconnection system between event producers and consumers, orchestrating event flow between the parties.

The key feature here is that, as the bus lies in the middle between producers and consumers, you can connect multiple systems in a many-to-many relationship almost effortlessly: producers will push their events to a bus, and all the consumers interested in a specific event, will receive and process it.

As an example, think about an e-commerce site that needs to process an order under many different aspects: payment processing, inventory management, and order shipment.

Same fashion for consumers that can be interested in the same event type, produced by multiple systems. An example of this can be a common notification system for the whole architecture.

In this sense, you can see the usage of event buses similar to a Pub/Sub system.

Managing Event Buses on AWS: Amazon EventBridge

The AWS service that lets you create and use event buses is **Amazon EventBridge**. Your AWS account starts with an event bus already created called "**default**". That bus is used by all the AWS services inside the account to publish all events related to their operations. This bus is very important because it is the backbone of all event-driven architecture inside AWS. A quick example of this: Lambda function triggered on uploaded objects on S3.

Other than the default event bus, you can create application-specific event buses to decouple multiple applications inside the same AWS account.

Another important level of decoupling that you can achieve using EventBridge event buses is AWS Organizations accounts decoupling. In organizations contexts, where the organization has multiple AWS accounts clustered for specific applications/workloads, usually there is the need to centralize specific resources so that the specific team that needs to work on those can easily do that.

Using an earlier example, think about the need to centralize logs and monitor metrics. EventBridge event buses can be read/write cross-account so, specific applications can

easily send their metrics to the given AWS account event bus where event triggers are in place to store and consolidate that kind of information.

There are additional features in the Amazon EventBridge service like **Rules and Pipes** to define rules for specific events, also from built-in AWS services integrations, filter and transform them, finally delivering them to the consumers.

In addition to EventBridge event buses, there is another AWS service that can be used as an event bus: SNS. SNS can be seen as a highly scalable and flexible event bus, allowing components to publish and subscribe to events, ensuring that changes in one part of the system trigger appropriate responses elsewhere. Using SNS topics, we can decouple events, simplifying their management.

Queues: decoupling, control flow, and flow control

Along with event buses, another tool that we can use to achieve or improve our architecture decoupling level are queues. Queues act as intermediaries between components, helping them in several ways: from asynchronous communication, removing the need to have both systems online at the same time, to timing decoupling, letting producers and consumers create and process messages at their own pace.

The queue service in AWS is SQS, which can also be integrated with EventBridge, complementing the service by acting also as a translator from event-driven to message-driven architectures.

To dive deeper into the potential of the EventBridge service and how we can leverage it for architectural decoupling, we need to introduce two additional concepts: **control flow and flow control.**

Control flow defines the order of operations to process messages or tasks. This topic is crucial to better understand every event bus integration in general. We will see that applied to use cases with EventBridge.

Basically, we need to define how systems integrate with each other. We can categorize each component of a given integration into 4 different classes depending on the communication with the other components of the integration:

- Puller: actively gets messages; its outputs are passively taken from the next step
- Pusher: its inputs are pushed from the previous step, actively pushes output messages
- Queue: passively gets input messages and gives output messages

- Driver: actively gets input messages and gives output messages

Let's see this in practice with two simple examples using a very common integration:

SNS → EventBridge → event destination.

Since SNS is a pusher source (actively push inputs), EventBridge event filtering and transformations are also pusher steps (they receive the input and process it, forwarding to the next step), different destinations types can lead to different types of integrations:

1. SQS destination: SQS is a queue (passively receives inputs). The integration happens seamlessly.
2. API destination: APIs can be seen as a driver step (get inputs, push outputs), and it can be a problem with only push steps in front.

In order to achieve the second integration, control flow strategies must be applied. An additional component must be placed in between to invert the control flow, getting the actively pushed inputs from the source and preparing the outputs for an active poller destination.

This is the definition of a queue step. One of the control flow properties of queues is to invert the flow, thus enabling integrations between pusher-puller systems.

Moreover, APIs can have rate limits, therefore settings like invocation rate and queue messages TTL can be used to not overload the destination system.

Here is where **flow control** comes into play. Queues are great for their temporal decoupling property; the produced message rate can vary a lot from the consumed one. However, if those rates are really different, this can become an issue as the queue continuously fills up until it's completely full.

To solve this problem, we have two main flow control patterns: TTL (time to live), which discards too old messages, and back pressure, which slows down the arrival rate. The first one is already implemented in SQS meanwhile, the second method must be implemented on the producer side.

Order and Delivery semantics

Writing about distributed systems and how their components exchange messages, it's worth spending a few lines discussing message order and delivery semantics.

Maintaining the order in which messages are processed in distributed systems can often be essential to guarantee the consistency and integrity of business processes. Message ordering is about ensuring that events are consumed and processed in the same order as they were produced.

The order becomes a tough challenge when we start having multiple consumers. We can achieve message ordering using the First-In-First-Out (FIFO) pattern. EventBridge's event buses events are delivered, guaranteeing FIFO ordering. SNS has FIFO Topics, while SQS has FIFO queues along with message groups feature. Groups achieve local ordering for messages. In fact, every message inside a given group will be sent for processing to the same consumer.

An additional note on FIFO Topics and queues: message groups will be forwarded from the topic to the queue.

Speaking about **delivery semantics**, EventBridge supports two delivery semantics: "At Least Once" and "Exactly Once." The former choice hints at the possibility of duplicate events, therefore, deduplication strategies should be applied and consumers should be designed to be idempotent. We can achieve deduplication with SQS queues using message deduplication ID and visibility timeout features.

Failure Handling, Archives, and Replays

Failures can occur also in distributed systems because of their complex nature. Effective error handling is essential for preserving system reliability, regardless of the cause of the error – network problems, unavailable services, or unforeseen data anomalies. Errors can be categorized into two main categories: temporary errors, which resolve with time, like a system is not available or the load is too high at that moment, and systematic errors, which are errors due to bugs that will happen over and over again.

We can use MaxRetryAttempt parameter and Dead Letter Queues (DLQs) to distinguish between the two kinds of errors and, for the latter one, have a team inspect it and update the code logic to solve it.

Messages can be archived for auditing, compliance, and historical analysis. You can archive using S3 for long-term event storage and use DLQs for immediate errors. Once events are archived, you can replay the needed messages.

The ability to replay events provides a continuous improvement loop, enabling developers to learn from errors, refine system logic, and enhance overall system reliability.

Quick tip on this: beware of using AWS message IDs because they will change during reprocessing. Moreover, consumers, along with downstream systems, will reprocess the same event. Therefore, they must be designed to be idempotent.

Conclusions

As we conclude this exploration of the complexities of decoupled cloud architectures on AWS, we reach a point where resilience and innovation converge. AWS services like EventBridge and SQS help you orchestrate and decouple your infrastructure, paving the way for a new era of cloud computing where reliability, scalability, and flexibility operate harmonically together.

Throughout our investigation, we saw how decoupling frees up components so they can operate together harmoniously yet separately. We analyzed flow control and control flow strategies, and we realized how important they are to balance messaging and communication between components in distributed systems. We also quickly explored topics like message ordering and delivery semantics, concluding by speaking about the potential given by error handling strategies, using DLQs as safety nets for the infrastructure, and event replays as historical memory for continuous improvement.

That said, it becomes clear that decoupling is not just a technical strategy, it serves as a tool for organizations to embrace innovation, quickly adjust to shifting conditions, and create long-lasting, resilient, and efficient systems. Let event buses help you as catalysts of this innovation process.

As we finish this examination, let's continue to innovate, adapt, and unlock the boundless potential of decoupled cloud architectures on AWS.

Resources

- AWS re:Invent 2023 - Advanced integration patterns & trade-offs for loosely coupled systems (API309)

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Matteo Goretti

DevOps Engineer @ beSharp. Passionate about Artificial Intelligence, in particular, Machine Learning and Deep Learning, and interested in Cloud Computing. I love trekking and nature in general. I relax with my guitar, play video games, and watch TV series.