Architetture Event-Driven a KM0: dal producer al consumer – Parte 2

11 Febbraio 2026 - 13 min. read

Nell'articolo precedente: "Architetture Event-Driven a KM0: dal producer al consumer" abbiamo esplorato le basi concettuali delle architetture Event-Driven, analizzando gli attori coinvolti, le tipologie di messaggi, i canali di messaging e i pattern di coupling che regolano l'interazione tra producer e consumer.

In questo secondo articolo ci concentreremo su come realizzare un'Event-Driven Architecture su AWS, con Amazon EventBridge come servizio centrale. EventBridge va oltre il semplice trasferimento dei dati: funge da event router, orchestrando il flusso tra producer e consumer e gestendo filtraggio, trasformazione e instradamento degli eventi. Il servizio fornisce inoltre l'infrastruttura necessaria per instradare eventi sia tecnici sia di dominio, provenienti da servizi AWS nativi, applicazioni SaaS o soluzioni custom.

L'event bus

In base al tipo di eventi che gestiamo, dobbiamo scegliere la tipologia di event bus più adatta. Ogni account AWS dispone di un default event bus per regione, dove vengono automaticamente instradati gli eventi emessi dai servizi AWS integrati con EventBridge. Quando si tratta di eventi di dominio, potremmo tecnicamente usare il default event bus come destinazione, ma suggerisco di utilizzare un nuovo custom event bus dedicato alla nostra applicazione, per garantire isolamento, governance più chiara e separazione tra eventi infrastrutturali e logica di business.

L'event envelope

Ora che abbiamo un'idea più chiara di dove, concentriamoci su cosa fluisce nell'event bus.

Per inviare un evento a un custom event bus, dobbiamo invocare l'API PutEvents. Quello che segue è un esempio di come possiamo invocarla usando AWS CLI.

```
aws events put-events --entries '[{
    "Source": "it.besharp.service",
    "DetailType": "customEvent",
    "Detail": "{
        \\\\"metadata\\\\": {
            \\\\"event-name\\\\": \\\\"custom-event-name\\\\",
            …
        },
        \\\\"data\\\\": {
            \\\\"key\\\\": \\\\"value\\\\",
            …
        }
    }",
    "EventBusName": "custom-event-bus"
}]'
```

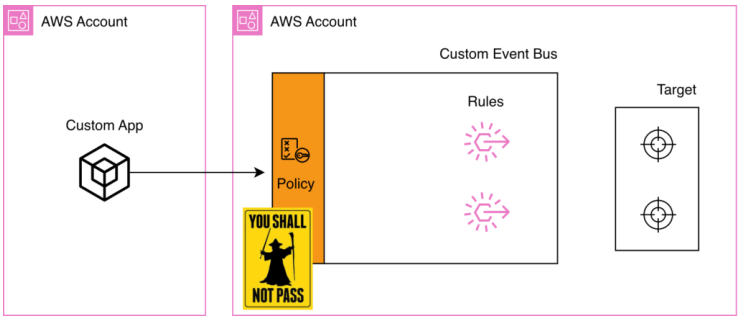
DetailType si riferisce al "tipo di evento"; suggerisce i campi che faranno parte dell'envelope.

L'event envelope è rappresentato dal campo Detail. Il valore associato a tale campo deve corrispondere ad un JSON valido. È pratica comune utilizzare una struttura standard, come "Metadata / Data". I campi Metadata sono pensati per essere applicati a tutti gli eventi, mentre Data contiene campi specifici per tipo di evento (DetailType).

"Metadata / Data" non è l'unico approccio standard esistente: questa è una valida alternativa. È comunque possibile definire la propria struttura, purché fornisca informazioni consistenti e significative ai consumer.

La resource policy

Una volta preparato l'evento, possiamo inviarlo al custom event bus. La prima barriera che l'evento incontra è la Resource Policy — ma diventa obbligatoria solo se stai inviando eventi a un bus in un account AWS diverso.



Se producer ed event bus sono all'interno dello stesso account, i permessi IAM sono sufficienti per l'autorizzazione: l'access policy legata al producer necessita solo del permesso **events:PutEvents** per inviare l'evento all'event bus.
È possibile aggiungere una resource policy per un controllo aggiuntivo (come limitare quali principal possono pubblicare in base a condizioni specifiche), ma non è obbligatorio, finché producer ed event bus sono all'interno dello stesso account. Lo stesso vale per il default bus che riceve eventi dai servizi AWS; nessuna resource policy è necessaria.

Immaginiamo di avere una Custom App nell'Account A che deve inviare eventi a un Custom Event Bus nell'Account B.

Configuriamo il nostro producer - una Lambda Function - con un'access policy che punta al bus dell'Account B; effettuiamo il deploy; proviamo a pubblicare un evento ma... EventBridge blocca la richiesta perché l'accesso cross-account richiede che entrambe le parti siano d'accordo.

In particolare:

- la Custom App necessita del permesso events:PutEvents per inviare l'evento sul Custom Event Bus nell'Account B;
- il Custom Event Bus deve esplicitamente permettere all'Account A (o allo IAM Role associato alla nostra Lambda Function) di inviare eventi.

Per far passare gli eventi dall'Account A all'Account B, questa resource policy deve essere agganciata all'event bus nell'Account B:

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowAccountAToPublishEvents",
```

```
      "Effect": "Allow",
      "Principal": {
        "AWS": "arn:aws:iam::<ACCOUNT-A-ID>:root"
      },
      "Action": "events:PutEvents",
      "Resource": "arn:aws:events:us-east-1:<ACCOUNT-B-ID>:event-bus/
custom-event-bus-name"
    }
  ]
}
```

Questa policy dice: "Mi fido dell'Account A. Lascia che i suoi principal pubblichino eventi." Ora entrambi i controlli di autorizzazione passano e gli eventi possono essere trasmessi all'event bus senza intoppi autorizzativi.

Event routing

Una volta che l'evento raggiunge l'event bus, EventBridge deve applicare eventuali logiche di trasformazione e capire a chi è destinato l'evento. È qui che entrano in gioco le EventBridge Rule.

Ad ogni regola di routing, collegata a un event bus, corrisponde un filtro chiamato event pattern. Tale filtro consente di intercettare solo gli eventi che corrispondono all'event pattern. Se il pattern corrisponde, l'evento viene inoltrato ai target associati alla regola di routing, previa eventuale trasformazione dell'input. Un singolo evento può attivare diverse regole di routing.

Prendiamo in considerazione il seguente evento:

```
{
  "version": "0",
  "id": "unique-event-id",
  "detail-type": "order.created",
  "source": "com.mycompany.orders",
  "account": "123456789012",
  "time": "2026-01-21T10:30:00Z",
  "region": "us-east-1",
  "resources": [],
  "detail": {
```

```
    "metadata": {
      "correlationId": "corr-123",
      "userId": "user-789"
    },
    "data": {
      "orderId": "ORD-12345",
      "customerId": "CUST-789",
      "amount": 99.99,
      "status": "pending"
    }
  }
}
```

All'interno dell'event pattern è possibile specificare anche i campi nidificati, ad esempio **detail.data**.

Prerequisiti per il matching

- **Il matching è case-sensitive:** **"order.created"** non è **"Order.Created"**.
- **Può raggiungere campi nidificati all'interno di altri oggetti:** **detail.data.orderId** o **detail.metadata.userId**
- **Tutti i criteri devono corrispondere:** l'evento deve soddisfare ogni condizione che definisci.
- **I valori sono contenuti in array:** questa condizione è vera anche nel caso di un singolo valore associato ad un campo. **"source": ["com.mycompany.orders"]**

Operatori di matching

I seguenti sono esempi di operatori comunemente utilizzati nella definizione di un event pattern.

Exact match (uguaglianza)

Match su valori specifici:

```
{
  "source": ["com.mycompany.orders"],
```

```
  "detail-type": ["order.created"]
}
```

Logica OR

Usa array per fare match su uno qualsiasi di diversi valori:

```
{
  "source": ["com.mycompany.orders"],
  "detail-type": ["order.created", "order.updated", "order.cancelle
d"]
}
```

Questo pattern consente di intercettare eventi con source **com.mycompany.orders** e (detail-type **order.created** , **order.updated** o **order.cancelled**).

Logica AND

L'evento corrisponde all'event pattern solo se tutte le condizioni definite sono soddisfatte.

```
{
  "source": ["com.mycompany.orders"],
  "detail-type": ["order.created"],
  "detail": {
    "data": {
      "status": ["pending"]
    }
  }
}
```

Prefix matching

Match su campi che iniziano con uno specifico prefisso:

```
{
  "source": ["com.mycompany.orders"],
  "detail": {
    "data": {
```

```
      "orderId": [{"prefix": "ORD-"}]
    }
  }
}
```

Suffix matching

Match su campi che terminano con uno specifico suffisso:

```
{
  "source": ["com.mycompany.users"],
  "detail": {
    "data": {
      "email": [{"suffix": "@mycompany.com"}]
    }
  }
}
```

Wildcard matching

È possibile utilizzare questa strategia per intercettare dei pattern più complessi:

```
{
  "detail": {
    "data": {
      "transactionId": [{"wildcard": "TXN-*-US-*"}]
    }
  }
}
```

"TXN-12345-US-VISA", **"TXN-67890-US-AMEX"** soddisfano il criterio di matching; lo stesso non vale per **"TXN-12345-EU-VISA"**.

Esclusione con anything-but

Puoi escludere valori specifici usando **anything-but**:

```
{
  "detail": {
```

```
    "data": {
      "status": [{ "anything-but": ["cancelled", "refunded"] }]
    }
  }
}
```

Questo operatore seleziona gli eventi in cui **status** non è né **cancelled** né **refunded**.

Combinare operatori di matching

In fase di definizione dell'event pattern, è possibile combinare diversi operatori per creare filtri complessi. Ecco un esempio che intercetta ordini enterprise ad alta priorità:

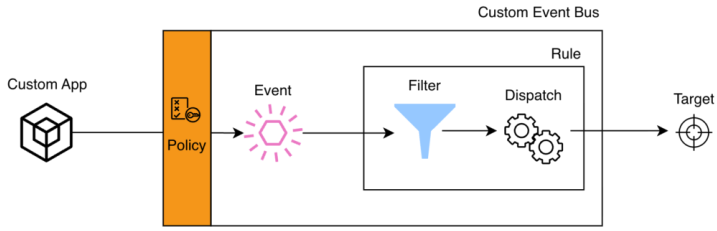
```
{
  "source": ["com.mycompany.orders"],
  "detail-type": ["order.created", "order.updated"],
  "detail": {
    "metadata": {
      "priority": ["high", "urgent"]
    },
    "data": {
      "customer": {
        "country": ["US", "CA", "GB"],
        "email": [{"suffix": "@enterprise.com"}]
      },
      "orderId": [{"prefix": "ORD-2026-"}]
    }
  }
}
```

Questo pattern fa match su eventi che soddisfano tutti i seguenti criteri:

- source è **com.mycompany.orders**;
- event type è **order.created** o **order.updated**;
- priority in metadata è **high** o **urgent**;
- customer country è **US**, **CA**, o **GB**;
- customer email termina con **@enterprise.com**;

- order ID inizia con **ORD-2026-**.

Event dispatching

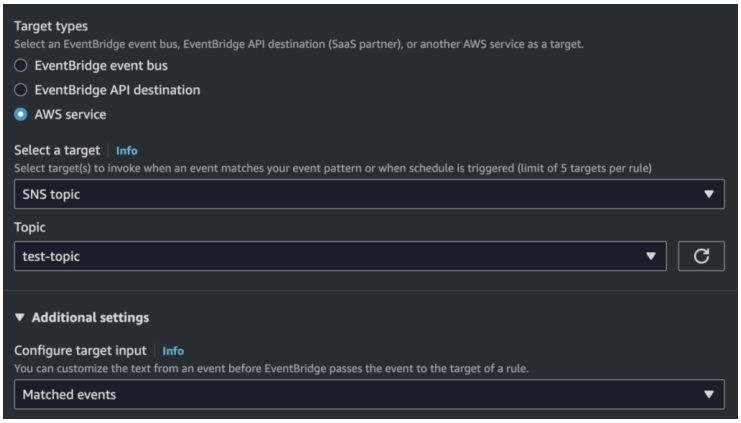


Oltre ad intercettare gli eventi sulla base di un event pattern (filtering), le regole di routing possono modificarne il contenuto (dispatching).

EventBridge offre quattro strategie di dispatching, pensate per scenari diversi.

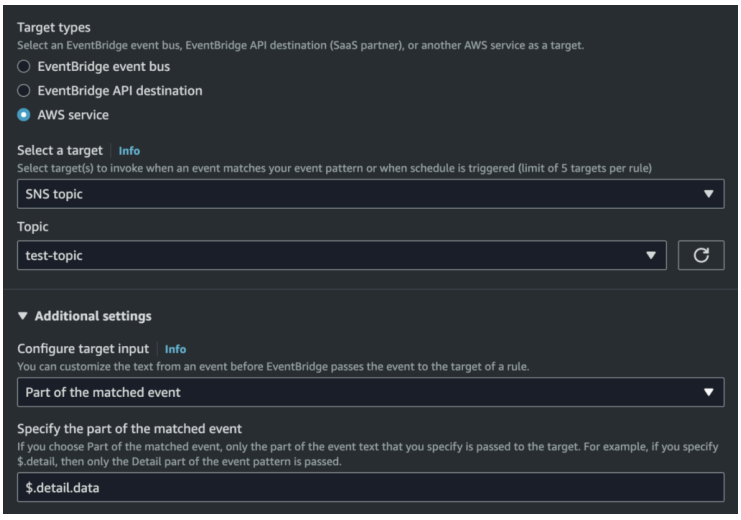
Event dispatching - Pass through

Questo è l'approccio più semplice: l'evento viene inoltrato senza essere modificato.



Event dispatching - Filtered: estrarre solo ciò che conta

A volte il target ha bisogno solo di una sezione dell'evento intercettato.



Usando Input Path, è possibile indirizzare uno specifico campo, eventualmente annidato. Per esempio, è possibile inoltrare solo il campo **detail.data**.

Input Path: **$.detail.data**

Cosa riceve il target

```
{
  "orderId": "ORD-12345",
  "customerId": "CUST-789",
  "amount": 99.99,
  "status": "pending"
}
```

Event dispatching - Transformed

Nel caso in cui il target dovesse aspettarsi una struttura completamente diversa o un contesto aggiuntivo che non è contenuto nell'evento, è possibile rimodellare l'evento utilizzando un Input Trasformer.

L'Input Transformer si aspetta un input path e un template.

Input path: definizione di variabili i cui valori vengono estratti dall'evento tramite un'espressione JSONPath.

Template: stringa o JSON contenente placeholder per le variabili definite all'interno dell'input path.

Esempio: trasforma un evento di ordine in un payload di notifica.

Input path

```
{
  "orderId": "$.detail.data.orderId",
  "customerEmail": "$.detail.data.customer.email",
  "amount": "$.detail.data.amount",
  "eventTime": "$.time"
}
```

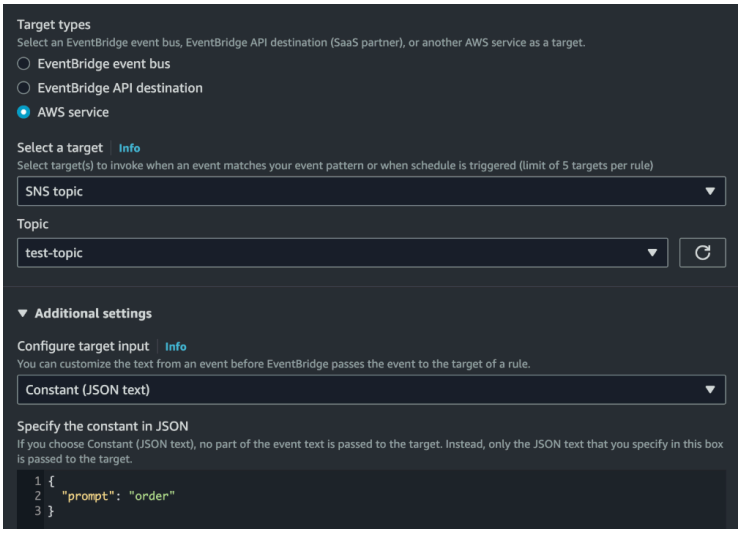
Input template

```
{
  "notificationType": "ORDER_CONFIRMATION",
  "recipient": "<customerEmail>",
  "message": "Your order <orderId> for $<amount> has been confirme
d.",
  "timestamp": "<eventTime>"
}
```

Cosa riceve il target

```
{
  "notificationType": "ORDER_CONFIRMATION",
  "recipient": "customer@example.com",
  "message": "Your order ORD-12345 for $99.99 has been confirmed.",
  "timestamp": "2026-01-21T10:30:00Z"
}
```

Event dispatching - Static



In questo caso, ai target viene inoltrato un payload costante e predefinito. Il contenuto dell'evento in input viene ignorato.

I target

Una volta intercettato e modellato dalla regola di routing, l'evento va consegnato al target.

EventBridge supporta diversi tipi di target: funzioni Lambda, Step Function, code SQS, topic SNS e tanti altri. Inoltre, è possibile destinare l'evento ad altri event bus. Per ogni regola di routing, è possibile definire fino a 5 target che possono risiedere in un account diverso da quello in cui si trova l'event bus sorgente.

Event Bus come target: costruire reti di eventi

È possibile configurare una regola di routing per inoltrare eventi a un altro event bus, sia nello stesso account che in un account diverso da quello dell'event bus sorgente.

È possibile centralizzare eventi provenienti da diversi account in un singolo account (di monitoring, ad esempio), favorendo la separazione delle responsabilità e una corretta ownership dei team che operano sui vari workload.

Target Event Bus nello stesso account

Se il target di una regola di routing corrisponde a un altro event bus nello stesso account, EventBridge può invocare direttamente il target event bus, senza dover specificare un execution role.

```
aws events put-targets \\\\\\\\
  --rule MyForwardingRule \\\\\\\\
  --targets "Id"="SameAccountTarget","Arn"="arn:aws:events:us-east-1:
111111111111:event-bus/target-bus"
```

Target Event Bus cross-account

Se il target della regola di routing è un event bus che si trova in un account diverso da quello del custom event bus sorgente, è necessario configurare:

- un execution role che permetta a EventBridge di inoltrare gli eventi verso l'event bus target;
- una resource policy - associata all'event bus di destinazione - che consenta alla regola di routing di inoltrare gli eventi.

Esempio: Forwarding dall'Account A (111111111111) all'Account B (222222222222)

Step 1: creazione di un ruolo IAM nell'Account A che EventBridge può assumere.

Trust policy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "events.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Permissions policy

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": "events:PutEvents",
      "Resource": "arn:aws:events:us-east-1:222222222222:event-bus/target-bus"
    }
  ]
}
```

Step 2: creazione di una resource policy da associare all'event bus target nell'Account B.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowAccountAToForwardEvents",
      "Effect": "Allow",
      "Principal": {
```

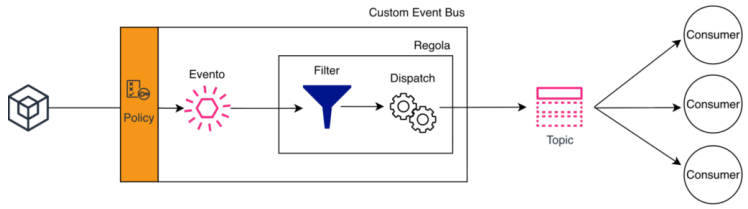
```
        "AWS": "arn:aws:iam::111111111111:root"
      },
      "Action": "events:PutEvents",
      "Resource": "arn:aws:events:us-east-1:222222222222:event-bus/target-bus"
    }
  ]
}
```

Step 3: configurazione del target della regola di routing, specificando l'ARN del ruolo creato nello step 1.

```
aws events put-targets \\\\\\\\
  --rule MyForwardingRule \\\\\\\\
  --targets "Id"="CrossAccountTarget","Arn"="arn:aws:events:us-east-1:222222222222:event-bus/target-bus","RoleArn"="arn:aws:iam::111111111111:role/PutEventsToTargetBusRole"
```

Fan-out attraverso SNS

Come dicevamo in precedenza, a una regola di routing è possibile associare fino a un massimo di 5 target. Tuttavia, è possibile superare questo limite utilizzando un topic SNS come target della regola di routing.



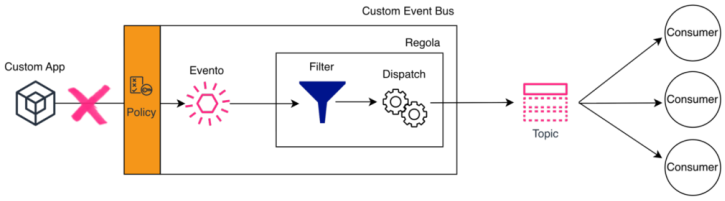
Questa architettura permette di sottoscrivere nuovi consumer al topic SNS, senza modificare la configurazione della regola di routing.

Tieni a mente...

Questa citazione famosa di Werner Vogels (CTO di Amazon) è un invito a progettare le nostre architetture in modo resiliente, pensando a tutti i possibili scenari di malfunzionamento di uno o più componenti della nostra soluzione. Problemi di rete, throttling e sistemi target temporaneamente non disponibili sono solo alcuni esempi di problematiche che — nella fattispecie — possono impedire il corretto inoltro dell'evento ai consumer.

Malfunzionamento lato client

Quando, lato client, utilizziamo l'API PutEvents per pubblicare un evento sul bus, l'operazione potrebbe fallire.

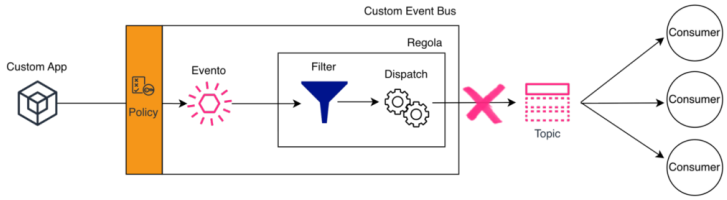


È importante notare che **PutEvents** è un'API batch (fino a 10 eventi per chiamata) e può restituire fallimenti parziali: alcuni eventi possono avere successo mentre altri falliscono nella stessa richiesta. La risposta include **FailedEntryCount** e l'array **Entries** con i dettagli per ciascun evento (ad esempio **ErrorCode** e **ErrorMessage** per quelli falliti). I client dovrebbero ispezionare la risposta e ritentare l'inoltro degli eventi associati ad un messaggio di errore, non l'intero batch.

Per gestire questi errori, il team responsabile dello sviluppo della Custom App può introdurre una logica di retry lato client. Ad esempio, è possibile implementare un exponential backoff con jitter. Invece di riprovare immediatamente una richiesta fallita, il client aspetta un tempo che aumenta esponenzialmente dopo ogni fallimento. Il jitter, inoltre, introduce una componente casuale nel tempo di attesa, sparpagliando le richieste nel tempo e riducendo le collisioni.

Malfunzionamento in fase di consegna al target

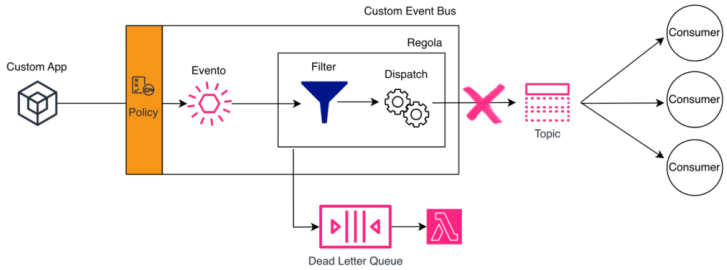
Anche in fase di inoltro dell'evento a uno specifico target di una regola di routing, qualcosa può andare storto.



Quando un evento non viene recapitato al target per via di errori recuperabili, EventBridge attiva una logica di retry dell'invio sulla base di due soglie: **Maximum event age** e **Maximum retry attempts**. I valori predefiniti sono, relativamente, 24 ore e 185 tentativi. EventBridge tenta nuovamente la consegna dell'evento, applicando un backoff esponenziale con jitter. EventBridge ritenta l'inoltro finché una delle due soglie impostate non viene raggiunta o l'evento non viene consegnato con successo. Per evitare che l'evento vada perso nel caso di raggiungimento di una delle due soglie, è sempre possibile appellarsi, previa configurazione, a un ulteriore strumento: la Dead Letter Queue.

Utilizzo di una Dead Letter Queue

È possibile configurare una coda SQS come DLQ per ciascun target all'interno di una regola di routing. Ogni regola può avere fino a 5 target e ciascun target può avere la propria DLQ configurata. Quando EventBridge raggiunge una delle due soglie citate in precedenza, invia l'evento alla DLQ del target invece di scartarlo. In questo modo, l'evento non viene perso e può essere processato da una logica dedicata.



L'architettura illustrata nelle ultime immagini serve a dare un'idea delle tecniche che si possono utilizzare per rendere resiliente il flusso di trasmissione dell'evento. Chiaramente, è possibile evolvere la soluzione in base ai requisiti del proprio workload. A prescindere dalla dimensione e dalla complessità della nostra architettura, le scelte progettuali vanno fatte tenendo sempre in considerazione la possibilità di malfunzionamenti provocati da uno o più componenti.

È opportuno anticipare possibili malfunzionamenti in produzione attraverso simulazioni eseguite in ambienti di sviluppo e collaudo. Durante i test, ad esempio, è possibile far fallire deliberatamente i target, così da verificare che l'evento non vada perduto e che, nel caso peggiore, venga inoltrato alla DLQ. In questo modo, possiamo accertarci che un'architettura resiliente sulla carta lo sia anche in pratica.

Conclusioni: dalla teoria alla pratica

Nel primo articolo, abbiamo esplorato le fondamenta concettuali delle Architetture Event-Driven: comprendere gli attori in gioco, gli eventi, i canali di messaging e i pattern di coupling che influenzano l'interazione tra producer e consumer, non solo nel tempo e nella topologia, ma anche nel formato e nella semantica. Questi principi aiutano a comprendere come strutturare sistemi poco accoppiati, che possano crescere e adattarsi facilmente, e come i diversi pattern di messaggistica contribuiscano a raggiungere questi obiettivi.

In questo articolo, siamo passati dalla teoria all'implementazione con **Amazon EventBridge** come orchestratore del flusso di eventi. Abbiamo ripercorso l'intero ciclo di vita di un evento, partendo dalla pubblicazione di eventi ben strutturati su custom event bus e dalla gestione delle autorizzazioni cross-account tramite resource policy. Abbiamo poi visto come utilizzare event pattern per intercettare e instradare gli eventi, trasformando i payload secondo le esigenze dei target.

Infine, ci siamo concentrati sulla consegna verso target eterogenei — inclusi event bus cross-account — e sulla gestione di eventuali malfunzionamenti attraverso meccanismi di retry integrati e Dead Letter Queue.

That's all, Folks!

Spero che questa serie di articoli ti abbia dato gli strumenti necessari per cominciare a progettare e, successivamente, implementare Architetture Event-Driven utilizzando Amazon EventBridge.

Ti invito ad approfondire i temi che abbiamo trattato e di non limitarti ad essi. I servizi evolvono, si aggiungono integrazioni e, molto spesso, i requisiti ci spingono oltre quello che ci offrono le funzionalità di base del servizio core - in questo caso EventBridge - della nostra soluzione.

Le scelte architetturali, soprattutto quelle fatte nelle fasi iniziali di progettazione, possono influire pesantemente sugli sviluppi futuri! Per questo motivo, è importante esplorare e studiare tecnologie alternative a quella che abbiamo intenzione di utilizzare.

Come sempre, sentiti libero di condividere feedback o contattarci se vuoi discutere casi d'uso specifici.

Resta sintonizzato per nuovi articoli!

About Proud2beCloud

Proud2beCloud è il blog di beSharp, APN Premier Consulting Partner italiano esperto nella progettazione, implementazione e gestione di infrastrutture Cloud complesse e servizi AWS avanzati. Prima di essere scrittori, siamo Solutions Architect che, dal 2007, lavorano quotidianamente con i servizi AWS. Siamo innovatori alla costante ricerca della soluzione più all'avanguardia per noi e per i nostri clienti. Su Proud2beCloud condividiamo regolarmente i nostri migliori spunti con chi come noi, per lavoro o per passione, lavora con il Cloud di AWS. Partecipa alla discussione!

Eric Villa

Solutions Architect @beSharp | AWS Community Builder | AWS Authorized Instructor

---