e-Commerce Saga Orchestration Implementation

Cloud Computing Technologies

Docenti: Anisetti Marco, Ardagna Claudio

Lorenzo Marcolli
mat. 08302A

Alberto Rizzi
mat. 08303A

A.A 2022/23

1 Introduzione

Nel contesto dell'e-commerce, la Saga Orchestration si riferisce a un pattern di progettazione architetturale utilizzato per gestire le transazioni complesse che coinvolgono più servizi o componenti all'interno di un sistema. Invece di utilizzare un approccio tradizionale basato su transazioni atomiche, in cui ogni operazione viene eseguita in modo indipendente, la Saga Orchestration adotta un approccio più flessibile.

In un sistema di Saga Orchestration, le transazioni vengono suddivise in una serie di passi o fasi, ognuno dei quali rappresenta una singola operazione all'interno di un servizio. Ad esempio, nell'ambito di un sistema di ordini di e-commerce, potrebbero esserci passaggi per la verifica dell'inventario, l'elaborazione del pagamento, la generazione dell'etichetta di spedizione, l'invio della notifica di conferma dell'ordine, ecc.

L'orchestratore della Saga si occupa di *coordinare* l'esecuzione di questi passi, monitorando il loro stato e prendendo decisioni in base ai risultati ottenuti. Se una delle operazioni fallisce, l'orchestratore può eseguire azioni di compensazione per annullare o correggere le operazioni precedenti. In questo modo, viene gestita la *coerenza* del sistema, anche in caso di errori o situazioni impreviste.

L'utilizzo di un'orchestrazione di Saga nel contesto dell'e-commerce consente di gestire in modo efficiente e coerente le transazioni complesse, garantendo che tutte le operazioni necessarie vengano eseguite correttamente e che il sistema mantenga uno stato coerente.

1.1 Saga Orchestration nel Cloud Computing

L'adozione della Saga Orchestration nell'ambito del cloud computing può portare numerosi vantaggi, tra cui la *scalabilità*, l'*affidabilità* e la *flessibilità* nella gestione delle transazioni distribuite e dei flussi di lavoro complessi.

Inoltre, i servizi cloud forniscono spesso funzionalità di monitoraggio, registrazione e gestione degli errori che facilitano la gestione e il debugging delle orchestrazioni Saga.

2 Gestione delle identità e degli accessi con IAM

IAM, acronimo di Identity and Access Management, è un framework o una serie di procedure e tecnologie utilizzate per la gestione delle identità e degli accessi su AWS.

2.1 Policy

Le policy AIM determinano quali azioni specifiche sono consentite o negate per una risorsa AWS. In questo modo è possibile definire con granularità quali servizi e risorse possono interagire tra loro.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "LambdaPermissions",
      "Effect": "Allow",
      "Action": [
        "lambda:InvokeFunction"
      ],
      "Resource": [
        "arn:aws:lambda:region:account-id:function:function-name"
      ]
    },
    {
      "Sid": "S3Permissions",
      "Effect": "Allow",
      "Action": [
        "s3:GetObject",
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::bucket-name/*"
      ]
    }
  ]
}
```

Listing 1: Esempio policy

– **“Version”: “2012-10-17”** specifica la versione della policy IAM.

– **“Statement”** è un array che contiene le singole dichiarazioni di autorizzazione.

– **“Sid”** è un identificatore univoco per ogni dichiarazione che facilita la gestione delle policy.

– **“Effect”** può essere “Allow” o “Deny”.

– **“Action”** specifica l’elenco delle azioni consentite o negate. In questo caso, `lambda:InvokeFunction` permette di invocare la funzione Lambda, e `s3:GetObject` e `s3:PutObject` permettono di leggere e scrivere oggetti in Amazon S3.

– **“Resource”** specifica le risorse su cui si applicano le autorizzazioni. Gli ARN (Amazon Resource Name) vengono utilizzati per identificare le risorse. In questo esempio, vengono specificati l’ARN della funzione Lambda e l’ARN del bucket S3.

3 Implementazione Saga Orchestration

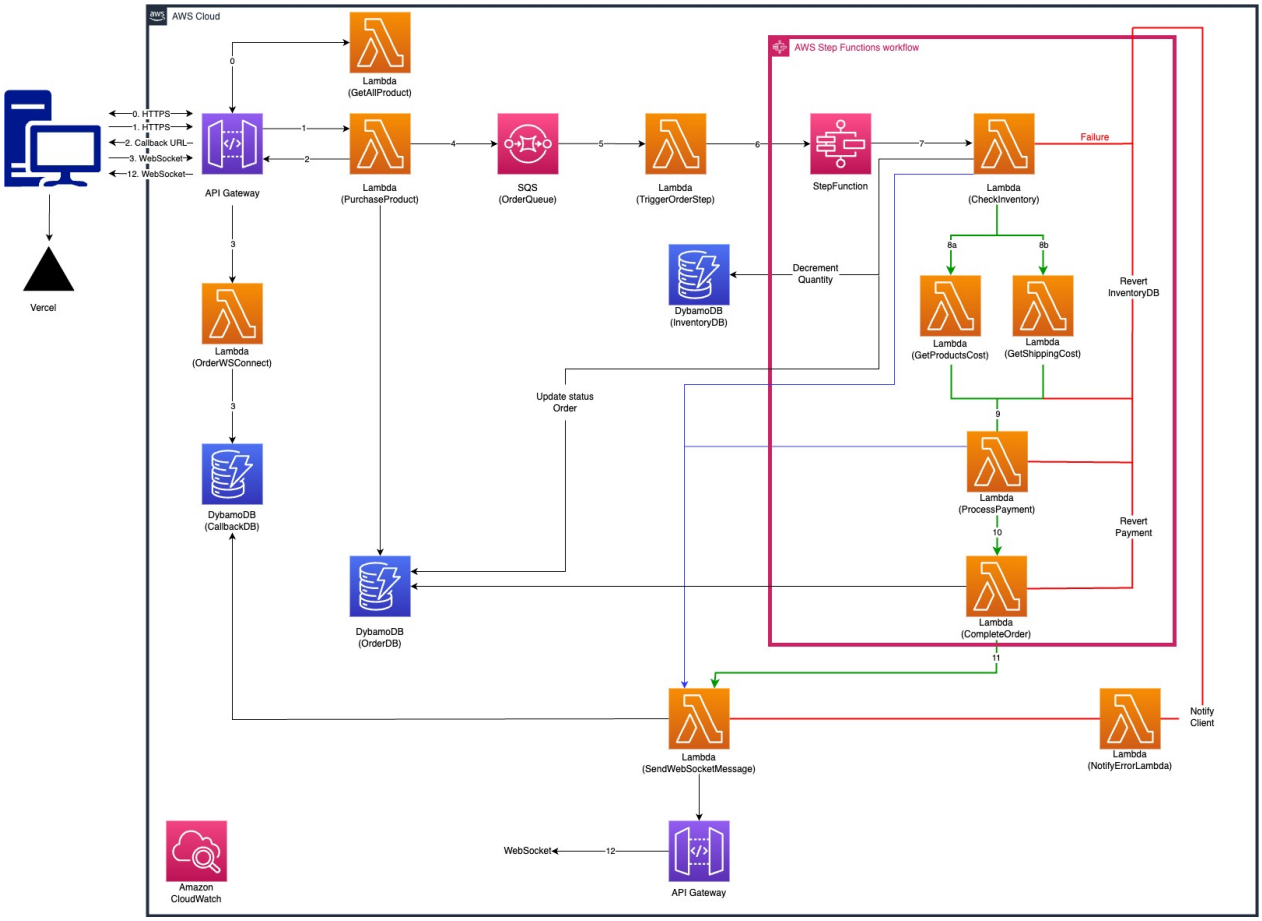

Figura 1: Workflow

Il processo eseguito per compiere la transazione di un ordine eseguirà le seguenti attività:

0. Il client effettua una chiamata `GET` REST API per recuperare tutti i prodotti in vendita, salvati nella tabella `Inventory` di DynamoDB

1. Il client effettua una chiamata `POST` REST API avente come payload tutti i dati inerenti all'ordine che un'utente vuole effettuare: il carrello dei prodotti, luogo di spedizione e i dati della carta

```
{
    "basket": [
        {
            "id": 1,
            "quantity": 1
        }
    ],
    "shippingAddress": {
        "address": "Via Roma 1",
        "city": "italia",
        "postCode": "21020"
    },
    "paymentDetails": {
        "cardNumber": "424242424242",
        "expiry": "26/07",
        "ccv": 123,
        "cardholderName": "Mario Rossi"
    }
}
```

Listing 2: Esempio oggetto da inviare alla lambda `PurchaseProduct`

2. La lambda `PurchaseProduct` salva l'ordine nella tabella `Order` aggiungendo l'attributo `status` con valore `pending`. Risponde al client con:

    - ID dell'ordine appena creato
    - URL per connettersi alla WebSocket e ricevere aggiornamenti sullo stato dell'ordine.

```
    {
        "message": "Message Send to SQS- Here is MessageId:
↪   c9f553cd-db9b-4648-981a-c43144b0cfbf",
        "orderId": "OyEFjKGB1uzBb97efICy",
        "callbackUrl":
↪   "wss://wkr7p95088.execute-api.eu-central-1.amazonaws.com/
        production/"
    }
```

Listing 3: Risposta della lambda `PurchaseProduct` al client

3. Il client riceve l'URL della WebSocket e si connette. La lambda `OrderWSConnect`, appena viene creata una nuova connessione, verrà invocata e salva nella tabella `CallbackDB` l'associazione tra `orderId` e la `connectionId` della nuova connessione.

4. La lambda `PurchaseProduct` invia il nuovo ordine alla coda SQS FIFO `OrdersQueue`.

```
const params = {
    DelaySeconds: 0,
    MessageAttributes: {
        Author: {
            DataType: "String",
            StringValue: "Preeti",
        }
    },
    MessageBody: JSON.stringify(dataSQS),
    MessageGroupId: randomString,
    FifoQueue: true,
    MessageDeduplicationId: randomString,
    MessageRetentionPeriod: 345600, // 4 days
    QueueUrl:
    ↪  "https://sqs.eu-central-1.amazonaws.com/495456954059/
    OrdersQueueFIFO.fifo"
};

const data = await sqsClient.send(new
↪  SendMessageCommand(params));
```

Listing 4: Invio messaggio alla coda SQS

5. L'inserimento di un nuovo ordine nella coda SQS funziona da trigger per la lambda `TriggerOrderStep`

6. La lambda `TriggerOrderStep` inizia l'esecuzione della StepFunction `OrderOrchestration`

```
stepFunctions.start_execution(
        stateMachineArn=state_machine_arn,
        input = sqs_message
    )
```

Listing 5: Esecuzione della StepFunction

7. La StepFunction funge da orchestrator, ovvero gestisce il flow di diversi microservizi e richiama come primo step la lambda `Check Inventory`

8. La lambda `CheckInventory` controlla che effettivamente ci sia disponibilità a magazzino, aggiorna la quantità nel database e in caso di successo viene avvisato il client e la *StepFunction* procede con il flow, in caso di fallimento verrà avvisato il client.

```
const updateItemParams = {
    TableName: 'Order',
    Key: {
        orderId: { S: orderId }
    },
    UpdateExpression: 'SET orderStatus = :statusString',
    ExpressionAttributeValues: {
        ':statusString': { S: 'cancelled' }
    }
};

const command = new UpdateItemCommand(updateItemParams);
const data = await dynamoDBClient.send(command);
```

Listing 6: Aggiornamento stato ordine su DynamoDB

9. Le lambda `GetProductCost` e `GetShippingCost` in parallelo calcolano reciprocamente il costo totale dell'ordine e i costi di spedizione. In caso di successo si procede allo step successivo; in caso di fallimento viene effettuato `rollback` andando a reinserire la quantità a magazzino e viene avvisato il client.

10. La lambda `ProcessPayment` esegue il pagamento e in caso positivo procede con lo step successivo, in caso di fallimento verrà effettuato `rollback` per reinserire la quantità dei prodotti a magazzino.

11. La lambda `CompleteOrder` aggiorna lo stato dell'ordine a DB e in caso di successo viene avvisato il client, in caso di fallimento viene effettuato `rollback` per rimborsarsa l'utente, reinserire la quantità dei prodotti a magazzino e viene avvisato il client.

12. Le lambda `SendWebSocketMessage` si occuperà di gestire l'invio di messaggi sulla connessione instaurata con il client tramite WebSocket.

```
const orderId = event.orderId;
const messageObj = event.messageObj;

const params = {
    TableName: 'CallbackDB',
    Key: {
        'orderId': { S: orderId }
    }
};
const command = new GetItemCommand(params);
const data = await dynamoDBClient.send(command);

// Retrieve the connection ID from the DynamoDB response
const connectionId = data.Item.connectionId.S;

// API Gateway Management API client initialization
const apigatewaymanagementapi = new
↪  ApiGatewayManagementApiClient({
    apiVersion: '2018-11-29',
    endpoint: 'https://wkr7p95088.execute-api.eu-central-1.
    amazonaws.com/production/'
});

// Prepare and send the WebSocket message using the connection
↪  ID
const paramsApiGateway = {
    ConnectionId: connectionId,
    Data: JSON.stringify(messageObj)
};
const postToConnectionCommand = new
↪  PostToConnectionCommand(paramsApiGateway);
await apigatewaymanagementapi.send(postToConnectionCommand);
```

Listing 7: Invio messaggio tramite WebSocket

3.1 Client

Il Client è quell'applicazione con la quale un utente può effettuare un ordine, andando così ad iniziare l'intero flow di acquisto.
Può essere una webapp, un app, o una qualunque applicazione in grado di effettuare delle chiamate di rete.

Per questo progetto è stato scelto di realizzare un prototipo di eCommerce andando a creare una semplice webapp.

3.1.1 Tecnologie utilizzate

Next.js Framework di sviluppo web React-oriented che consente di creare applicazioni web moderne e performanti. Con la sua architettura basata sul server-side rendering (SSR) e sul rendering lato client, Next.js[8] offre un'esperienza di sviluppo efficiente e ottimizza la velocità di caricamento delle pagine, migliorando la SEO e offrendo una navigazione fluida agli utenti. É stato usato con

- Typescript[10]

- Tailwind.css[9]

Vercel Piattaforma di deployment e hosting per applicazioni web e statiche che si integra perfettamente con Next.js e altri framework popolari. Maggiori informazioni al capitolo 4.

Figura 2: Vercel e Next.js

3.2 API Gateway e Websocket

API Gateway è un servizio di gestione delle API completamente gestito e offerto da AWS. Funziona come un proxy che consente di creare, pubblicare, gestire e proteggere le API RESTful e WebSocket.

L'API Gateway può gestire fino a 10.000 richieste al secondo, se abbinata con Lambda anch'essa scalerà in automatico per far fronte al numero di richieste in entrata.

3.2.1 Load balancing

- API Gateway offre il bilanciamento del carico tra più istanze di backend per garantire una distribuzione equa del traffico.
- Supporta il bilanciamento del carico a livello di risorsa, consentendo di distribuire il traffico in base a criteri specifici come il peso o la priorità delle risorse.
- Offre anche l'integrazione con Elastic Load Balancer di AWS per il bilanciamento del carico su diverse regioni geografiche.

3.2.2 Fault tolerance

- API Gateway fornisce una gestione automatica dei fault tolerance grazie alla sua architettura a disponibilità elevata.
- Supporta la distribuzione multi-regione per ridurre il rischio di interruzioni del servizio in caso di guasti in una determinata regione AWS.
- Utilizza la replica dei dati e la ridondanza per garantire che le richieste degli utenti siano elaborate anche in caso di guasti hardware o software.

3.3 AWS Lambda

AWS Lambda è un servizio di calcolo serveless che consente agli sviluppatori di eseguire il proprio codice in modo scalabile ed efficiente, senza la necessità di gestire l'infrastruttura sottostante.

Con Lambda, gli sviluppatori possono scrivere il proprio codice e definire le azioni da eseguire in risposta a specifici eventi. Questi eventi possono essere generati da servizi AWS come Amazon S3, DynamoDB o SQS, o da fonti esterne come notifiche push o aggiornamenti di dati. Quando si verifica un evento, Lambda avvia automaticamente l'esecuzione del codice associato, fornendo una risposta rapida e affidabile.

Uno dei principali vantaggi di Lambda è la sua natura serverless, che permette agli sviluppatori di concentrarsi sulla logica dell'applicazione senza dover gestire l'approvvigionamento o la configurazione delle risorse sottostanti. Lambda scala automaticamente in base alla richiesta di esecuzione, garantendo che le risorse siano allocate in modo efficiente per gestire picchi di carico improvvisi senza costi eccessivi.

Lambda supporta diversi linguaggi di programmazione, tra cui Node.js, Python, Java, C# e Go, consentendo agli sviluppatori di utilizzare le proprie competenze e preferenze. Inoltre, Lambda si integra facilmente con altri servizi AWS, consentendo di creare soluzioni complesse e distribuite sfruttando l'intero ecosistema di servizi cloud di AWS.

Alcune delle caratteristiche interessanti sono:

- Versioning
- Scalabilità
- Alta disponibilità

Lambda richiama la funzione in un ambiente di esecuzione sicuro e isolato. Per gestire una richiesta, Lambda deve prima inizializzare un ambiente di esecuzione (la fase *Init*), prima di utilizzarlo per invocare la funzione.

3.4 Amazon SQS

Amazon Simple Queue Service (SQS) consente di inviare, memorizzare e ricevere qualsiasi volume di messaggi tra componenti software senza perdita di dati e indipendentemente dalla disponibilità di altri servizi.

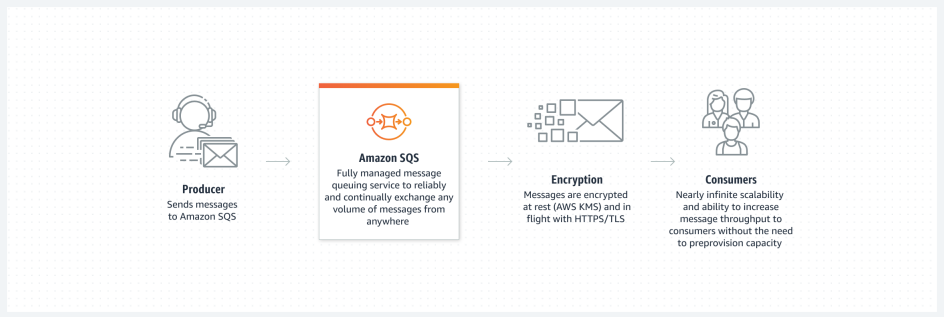

Figura 3: AWS SQS

Le caratteristiche principali sono:

- É un servizio di accodamento messaggi completamente gestito
- Garantisce una comunicazione affidabile tra componenti software distribuiti e microservizi
- Disaccoppiato da altri componenti che possono quindi fallire indipendentemente → permettono quindi di creare applicazioni fault tollerance e facili da scalare
- Permette lo scambio di messaggi tra diversi sistemi, di qualunque entità di volume senza perdita
- Mantieni l'ordine dei messaggi con la deduplicazione[1]

3.4.1 Tiplogie di code

Amazon SQS offre due tipi di coda:

1. Code standard
2. Code FIFO

Code standard La consegna del messaggio avviene con un principio "miglior ordine possibile", per cui di tanto in tanto i messaggi potrebbero essere consegnati in un ordine diverso da quello di ricezione. Inoltre, garantisce che il messaggio venga consegnato "At-Least-Once", quindi un messaggio viene consegnato almeno una volta, ma di tanto in tanto viene consegnata più di una copia di un messaggio.

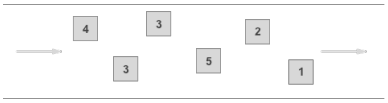

Figura 4: Coda standard

[1]La deduplicazione dei dati è una tecnica che riduce al minimo lo spazio necessario per l'archiviazione. È stata realizzata per consentire alle organizzazioni di risolvere il problema dei dati duplicati.

Code FIFO L'elaborazione del messaggio è "exactly-once": un messaggio viene consegnato una volta e rimane disponibile fino a quando un client non lo elabora e lo cancella. Non vengono introdotti duplicati nella coda. Il principio di consegna è First-In-First-Out: viene quindi mantenuto l'ordine esatto in cui i messaggi vengono inviati e ricevuti.

Figura 5: Coda FIFO

Tra tutte le caratteristiche, le più interessanti sono:

- Dimensione payload: i payload dei messaggi possono contenere fino a 256 KB di testo in qualunque formato.
- Conservazione dei messaggi in coda fino a un massimo di 14 giorni.
- Invio e lettura dei messaggi simultaneamente.

3.4.2 Code nella "Saga Orchestration"

La coda FIFO SQS garantisce la consegna affidabile dei messaggi, mantenendo un ordine di consegna dei messaggi in base all'ordine di inserimento. In questo modo, la coda SQS assicura che gli ordini vengano elaborati in modo coerente e che nessun ordine venga perso o ignorato nel processo di orchestrazione.

Possiamo quindi riassumere le motivazioni dell'uso delle coda SQS nei seguenti punti:

- Ordine garantito: utilizzando SQS FIFO, la coda garantisce che gli ordini vengano elaborati nell'ordine in cui sono stati inviati. Questo è fondamentale per un'applicazione di e-commerce, in quanto gli utenti si aspettano che i loro ordini siano gestiti in modo sequenziale.
- Scalabilità: SQS FIFO è altamente scalabile e può gestire un grande numero di messaggi in coda. Ciò consente all'applicazione di gestire picchi di traffico senza problemi e senza compromettere l'ordine di elaborazione degli ordini.
- Affidabilità: SQS offre una forte affidabilità nella consegna dei messaggi. I messaggi vengono archiviati in modo sicuro e persistente fino a quando non vengono elaborati dai consumatori. In caso di errore o arresto del server backend, i messaggi non elaborati rimarranno nella coda e saranno disponibili per l'elaborazione successiva.
- Integrazione con altri servizi AWS

3.5 AWS Step Functions

AWS Step Functions è un servizio gestito di orchestrazione di flussi di lavoro basato su cloud che semplifica la creazione e l'esecuzione di applicazioni distribuite. Permette di coordinare passaggi di lavoro complessi, gestire errori e implementare logiche di business personalizzate.

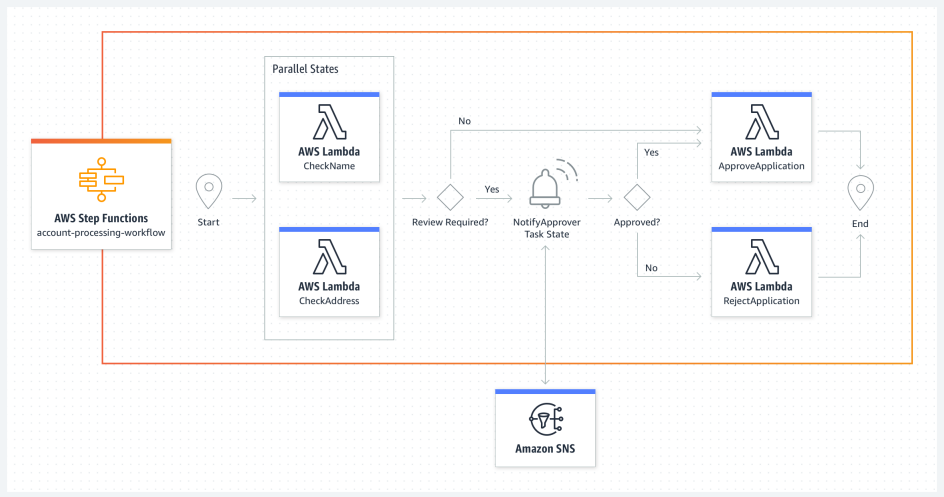

Figura 6: Esempio di Step Function

3.5.1 Step Function nella "Saga Orchestration"

Le Step Functions sono il core di tutto il pattern. Grazie alle loro caratteristiche permettono di orchestrare tutti i passaggi necessari per l'elaborazione di un ordine e gestire logicamente le transizioni tra gli step.

Come esempio di progetto, gli step scelti per l'elaborazione di un'ordine sono:

1. Controllo della disponibilità
2. Calcolo totale dell'ordine, diviso in
    - Calcolo del costo dei prodotti
    - Calcolo del costo di spedizione
3. Gestione pagamento
4. Completamento ordine
5. Gestione notifica al Client

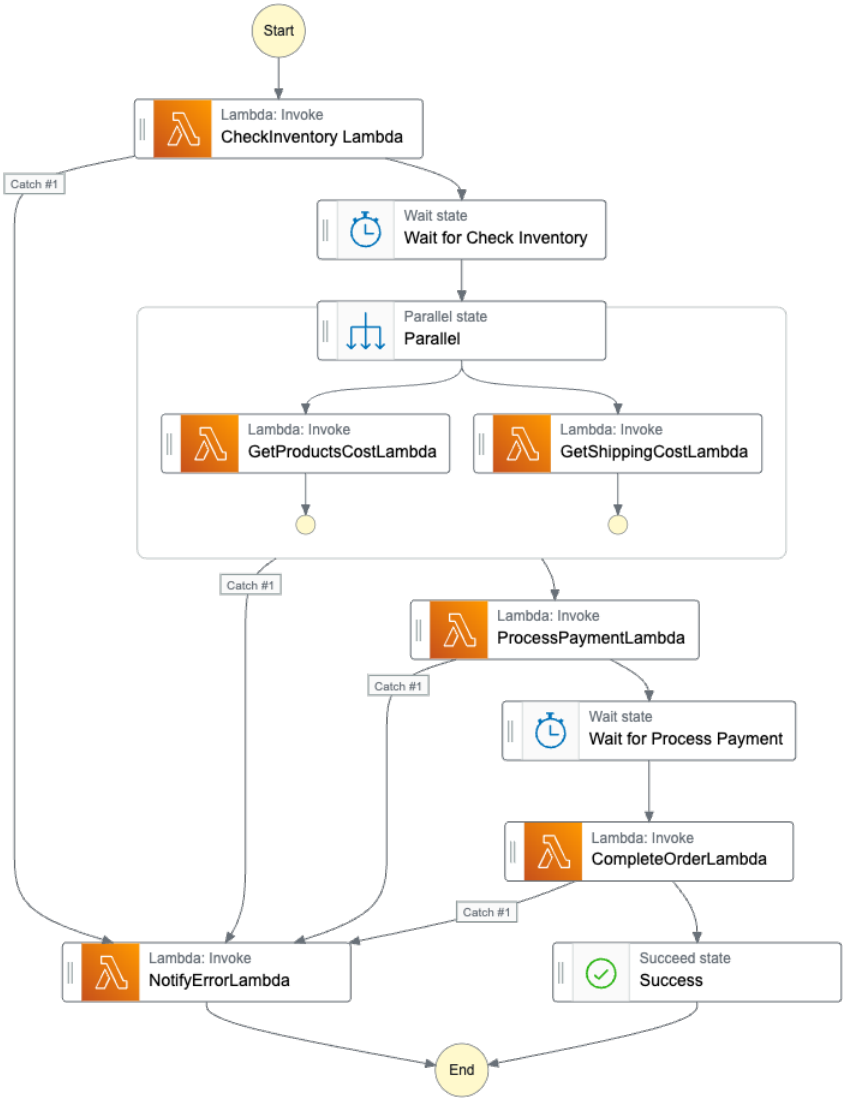

Figura 7: Workflow Step Function

3.6 DynamoDB

DynamoDB è un servizio di database gestito e scalabile progettato per archiviare e recuperare grandi quantità di dati in modo affidabile e ad alte prestazioni. DynamoDB è una soluzione di database NoSQL, il che significa che non utilizza uno schema rigido come i database relazionali tradizionali, ma piuttosto un modello di dati flessibile basato su coppie di chiave-valore.

Una delle caratteristiche chiave di DynamoDB è la sua scalabilità automatica. Può gestire carichi di lavoro di qualsiasi dimensione, dall'elaborazione di piccoli set di dati a scalare a petabyte di dati su migliaia di server. DynamoDB si adatta automaticamente per gestire la capacità richiesta, senza richiedere alcuna configurazione o provisioning manuale da parte degli sviluppatori. Ciò significa che è possibile scalare in modo dinamico il database per gestire picchi di traffico o ridurre la capacità durante i periodi di inattività.

La scalabilità in DynamoDB può essere raggiunta attraverso due concetti principali:

1. **Partizionamento dei dati**: DynamoDB divide automaticamente i dati in partizioni e distribuisce queste partizioni tra i nodi. Ogni partizione contiene un sottoinsieme dei dati totali. Questo approccio consente di distribuire il carico di lavoro su un numero maggiore di nodi e di eseguire operazioni parallele sui dati.

2. **Provisioned Throughput**: DynamoDB offre un modello di tariffe basato sulla capacità di throughput richiesta per l'applicazione. É possibile specificare il numero di unità di lettura e scrittura al secondo (Read Capacity Units - RCUs e Write Capacity Units - WCUs) necessarie per gestire il traffico previsto. DynamoDB alloca automaticamente le risorse di calcolo e di archiviazione necessarie per soddisfare la capacità richiesta.

DynamoDB offre anche una latenza estremamente bassa e una alta disponibilità dei dati. I dati vengono replicati in più zone di disponibilità all'interno di una regione AWS, garantendo la durabilità e la ridondanza dei dati. Ciò consente alle applicazioni di accedere rapidamente ai dati senza preoccuparsi di tempi di inattività o perdita di dati.

Un altro aspetto importante di DynamoDB è la sua flessibilità. Può gestire sia carichi di lavoro ad alte prestazioni che carichi di lavoro di lettura pesanti, consentendo alle applicazioni di adattarsi alle esigenze specifiche. DynamoDB supporta anche operazioni transazionali, consentendo l'esecuzione di più operazioni atomiche in un'unica transazione.

4 Vercel

Vercel è una piattaforma di deployment e hosting per applicazioni web che sfrutta il cloud computing per fornire un'infrastruttura scalabile e performante per le applicazioni web. È stata creata per semplificare il processo di distribuzione di applicazioni web e rendere più facile la loro gestione.

Cloud Vercel utilizza diversi servizi cloud, in particolare si affida ad AWS. Utilizza i seguenti servizi:

- Amazon S3 - Cloud Object Storage
- Amazon Simple Queue Service
- Auto scaling fleet of EC2 instances powered by AWS Fargate
- Amazon Global Accelerator
- AWS Global Network
- Amazon EKS
- AWS Lambda

4.1 Architettura

L'architettura cloud di Vercel è progettata per fornire un'infrastruttura altamente performante e scalabile. Con Vercel, gli sviluppatori possono godere dei vantaggi di una piattaforma senza preoccuparsi della gestione dell'infrastruttura sottostante.

L'architettura di Vercel si basa su una combinazione di deployment continuo, caching edge globale, funzioni serverless e scalabilità automatica.

4.1.1 Deploy

Il deploy di un'applicazione su Vercel inizia da codice scritto in uno dei 35 framework supportati[2]. Per inizializzare la fase di deploy ci sono due alternative:

1. Attraverso la CLI
2. Attraverso l'integrazione con Git → vengono monitorati i commit, che inizializzano un nuovo deployment

Se l'utente che sta richiedendo il deploy è autorizzato, allora viene schedulata la build. Viene quindi eseguito il comando di build (diverso in base al framework scelto) e tramite la CLI o la dashboard possiamo visualizzare lo stato e i log. Tutta la fase di build è effettuata all'interno del "container build", il quale genera un output che viene eseguito su uno dei runtimes offerti e mette a disposizione risorse quali:

- Funzioni Serveless[3]
- Funzioni Edge[4]
- Ottimizzazione delle immagini[5]
- Output statico

A questo punto il nuovo deploy è disponibile tramite i CDN di Vercel, quando viene richiesto da parte di un client.

[2]https://vercel.com/docs/frameworks
[3]https://vercel.com/docs/concepts/functions/serverless-functions
[4]https://vercel.com/docs/concepts/functions/edge-functions
[5]https://vercel.com/docs/concepts/image-optimization

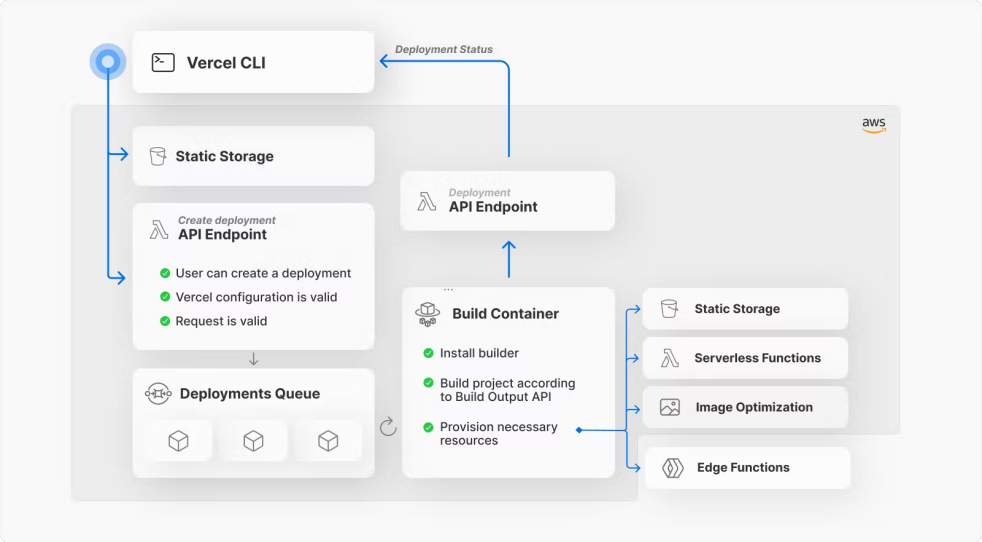

Figura 8: Deploy flow Vercerl

4.1.2 Richieste

Prima che un browser veda il sito web, viene prima effettuato il lookup DNS per trovare l'indirizzo IP. Per tutti i siti web ospitati su Vercel, questo step si risolve con un indirizzo IP anycast e il record CNAME `cname.vercel-dns.com`.

Mentre GeoDNS instrada gli utenti verso endpoint univoci in base alla loro posizione, Vercel utilizza un servizio di rete che utilizza il routing anycast per instradare il traffico al data center ottimale determinato dal numero di hop, tempo di andata e ritorno e quantità di larghezza di banda disponibile. Ciò può migliorare le prestazioni della rete instradando il traffico verso la destinazione più vicina all'origine, garantendo che gli utenti di tutto il mondo possano beneficiare di connessioni a bassa latenza.

Per cui l'indirizzo IP collega l'utente all'edge location più vicina, dopodiché la richiesta arriva in un cluster Kubernetes e:

- La richiesta viene ispezionata, filtrata in caso di utenti malintenzionati
- Viene inoltrata la richiesta ad una macchina virtuale (Amazon EKS) che:
  - Recupera la versione di deployment che deve essere servita
  - Se sono abilitati gli Edge Middleware, la risposta viene modificata di conseguenza
  - Viene inoltrata la risposta

La risposta può avere diversi tipi:

- Per le risorse statiche (es. pagina statiche, font, immagini non ottimizzate) il gateway scaricare le risorse dallo storage (AWS S3)

- Per le funzioni serverless: viene invocata la funzione (AWS Lambda)
- Per le funzioni Edge (es. pagina renderizzata lato server) che usano `edge` runtime, il gateway formatta la risposta affinchè la funzione venga eseguita "all'edge".
- Per le immagini ottimizzate, la richiesta viene inoltrata ad un servizio che ottimizza le imamgini "on-the-fly" e vengono cachate "all'edge" per le successive richieste.

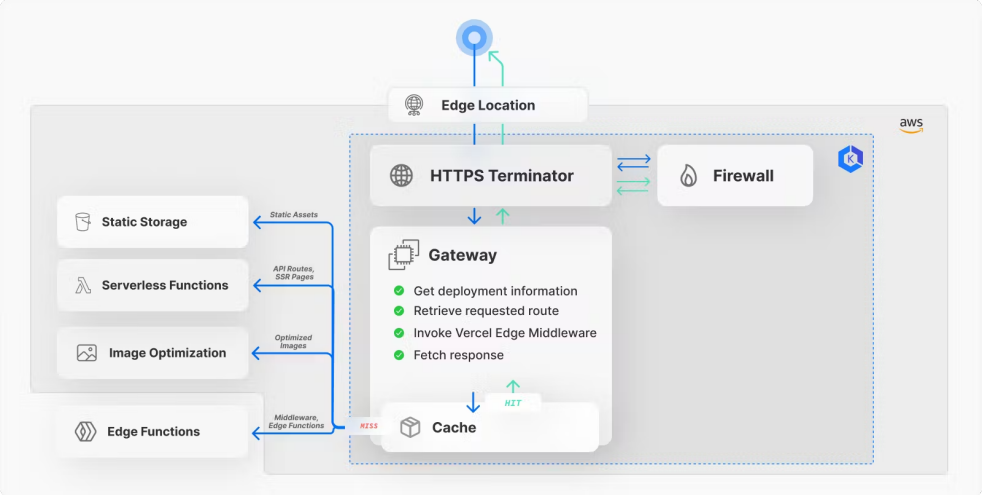

Figura 9: Request flow Vercerl

4.2 Next.js

Next.js è un framework di sviluppo web per React che combina il rendering lato server (Server-Side Rendering, SSR) e il rendering lato client (Client-Side Rendering, CSR) per creare applicazioni web performanti e scalabili. Tra le funzionalità principali:

- Server-Side Rendering (SSR): Next.js supporta il rendering lato server, consentendo di generare dinamicamente il markup HTML per ogni richiesta. Ciò permette ai motori di ricerca di indicizzare le pagine in modo efficace e offre prestazioni migliori iniziali per gli utenti.
- Static Site Generation (SSG): Next.js supporta anche la generazione di siti statici, consentendo di pre-renderizzare le pagine come file HTML statici durante la fase di compilazione. Questo migliora notevolmente le prestazioni e l'affidabilità, poiché le pagine possono essere memorizzate nella cache e servite velocemente.
- Client-Side Rendering (CSR): Next.js permette anche il rendering lato client, consentendo di creare applicazioni interattive che si basano su dati caricati in modo asincrono dal server o da API.

- Routing dinamico: Next.js offre un sistema di routing dinamico che semplifica la gestione delle rotte dell'applicazione. Supporta anche la generazione di rotte basate su parametri dinamici.

4.2.1 Next.js nella "Saga Orchestration"

Il motivo della scelta di utilizzare questo framerwork in un e-Commerce è:

- Prestazioni
- SEO-friendly
- Hosting integrato con Vercel → scalabilità e integrazioni
- Ottimizzazione delle immagini

5 Casi di successo

- Zalando [1]

Riferimenti bibliografici

[1] Alinabo. *Mastering Saga Pattern for Microservices: Best Practices and Solutions*. 2023. URL: https://alinabo.com/saga-pattern-for-microservices/.

[2] *AWS SQS*. URL: https://aws.amazon.com/it/sqs/features/.

[3] *AWS Step Function*. URL: https://aws.amazon.com/it/step-functions/.

[4] DevAx::academy. *Monoliths to microservices*. 2020. URL: https://workshops.devax.academy/monoliths-to-microservices/module8.html.

[5] *DynamoDB*. URL: https://aws.amazon.com/it/dynamodb/.

[6] Hector Garcaa-Molrna e Kenneth Salem. *SAGAS*. URL: https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf.

[7] Lydia Hallie. *Behind the scenes of Vercel's infrastructure: Achieving optimal scalability and performance*. 2023. URL: https://vercel.com/blog/behind-the-scenes-of-vercels-infrastructure.

[8] *Next.js*. URL: https://nextjs.org/.

[9] *Tailwind.css*. URL: https://nextjs.org/.

[10] *Typescript*. URL: https://www.typescriptlang.org/.