Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Un'Analisi del Design di un Linguaggio Coreografico per Architetture Serverless

Relatore:
Chiar.mo Prof.
Giallorenzo Saverio

Presentata da:
Tomaiuolo Alessandro

Correlatore:
Chiar.mo Prof.
Trentin Matteo

Sessione IV
Anno Accademico 2023/2024

Indice

Elenco delle figure

Capitolo 1

Introduzione

Questa tesi si propone di analizzare il design e le caratteristiche di FaaSChal, un linguaggio di programmazione coreografico ideato specificamente per architetture serverless. Il suo obiettivo principale è semplificare lo sviluppo ed il deployment di applicazioni distribuite sfruttando il paradigma Function-as-a-Service (FaaS), consentendo la definizione di interazioni tra componenti distribuiti in modo dichiarativo e strutturato.

Nel corso dell'elaborato, verranno dapprima introdotti i concetti fondamentali del cloud computing, soffermandoci sui principali servizi disponibili e sulle loro differenze, con particolare attenzione al paradigma serverless. In questo contesto, verrà approfondito il modello FaaS, analizzandone i vantaggi, le limitazioni e i possibili scenari d'uso. Verranno inoltre introdotti brevemente i concetti di locality e linguaggi di orchestrazione. Successivamente, verrà trattata la programmazione coreografica, partendo dalla definizione di *coreografia* e dalla distinzione tra *linguaggi coreografici* e *linguaggi di programmazione coreografici*, per poi esaminare le primitive di comunicazione.

Un aspetto centrale dell'analisi sarà l'applicazione concreta di questi concetti attraverso FaaSChal, illustrando la sintassi e le funzionalità del linguaggio mediante esempi pratici. Verranno presentate coreografie scritte in FaaSChal, mostrando il processo di generazione automatica del codice eseguibile attraverso la tecnica di proiezione endpoint, nonché il successivo deployment su servizi pubblici.
Inoltre, verrà affrontato un aspetto teorico di rilievo nei linguaggi coreografici: la Knowledge of Choice. Questa proprietà è cruciale per garantire la coerenza dell'esecuzione distribuita e per evitare ambiguità nelle interazioni tra i partecipanti della coreografia.

Capitolo 2

Background

In questo capitolo verranno introdotti i concetti fondamentali necessari per comprendere in modo chiaro e approfondito il lavoro sviluppato nella presente tesi. Verranno forniti i riferimenti teorici e tecnici essenziali per contestualizzare gli argomenti trattati.

2.1 Cloud Computing

Con *Cloud computing* si indica la possibilità di archiviare e accedere a dati e programmi tramite server remoti ospitati su Internet, anziché sul disco rigido del computer o su server locali. Questa tecnologia, nota anche come *Internet-Based Computing*, prevede la distribuzione delle risorse informatiche sotto forma di servizio accessibile attraverso la rete. Le informazioni archiviate nel cloud possono comprendere file, immagini, documenti e, più in generale, qualsiasi contenuto digitale.

Esistono tre principali modelli di implementazione cloud, che si differenziano principalmente per la proprietà delle risorse e per la gestione delle stesse [1]:

- ***Cloud privato***: ambiente cloud in cui tutte le risorse sono dedicate esclusivamente a un'unica organizzazione, che ne detiene il controllo completo e la responsabilità di gestione.

- ***Cloud pubblico***: ambiente in cui un provider esterno gestisce e mantiene le risorse informatiche, occupandosi direttamente della loro disponibilità, affidabilità e sicurezza, spesso garantite tramite precisi accordi sui livelli di servizio. I principali provider di cloud pubblici sono Amazon Web Services (AWS), Google Cloud, IBM Cloud e Microsoft Azure.

- ***Cloud ibrido***: ambiente che combina elementi del cloud privato e pubblico. Un'organizzazione può, ad esempio, utilizzare le proprie risorse private e integrarle con quelle pubbliche per gestire efficacemente eventuali picchi di traffico o richieste impreviste.

Quando si parla di servizi cloud, è comune utilizzare la sigla *"as a Service"* (come servizio), indicando che il servizio è offerto e gestito da un provider esterno per conto dell'utente. I modelli si differenziano tra loro principalmente per il livello di controllo e gestione che il provider delega all'utente finale [11]. I tre modelli principali di servizi cloud *as a Service* sono:

- ***Infrastructure as a Service (IaaS)***: è un modello in cui il provider cloud offre l'accesso a risorse infrastrutturali quali rete, server, virtualizzazione e storage. L'utente ha il controllo e la responsabilità di gestire sistema operativo, applicazioni e dati.

- ***Platform as a Service (PaaS)***: è un ambiente che permette agli utenti di concentrarsi esclusivamente sulla scrittura e gestione del codice delle applicazioni, senza preoccuparsi della manutenzione dell'infrastruttura sottostante, gestita interamente dal provider cloud.

- ***Software as a Service (SaaS)***: rappresenta il modello più completo di servizio cloud, in cui il provider gestisce direttamente un'intera applicazione accessibile dagli utenti tramite browser. Il fornitore cura completamente aggiornamenti, correzioni di bug e tutte le attività di manutenzione software, mentre l'utente finale utilizza semplicemente il servizio tramite un'interfaccia web dedicata.

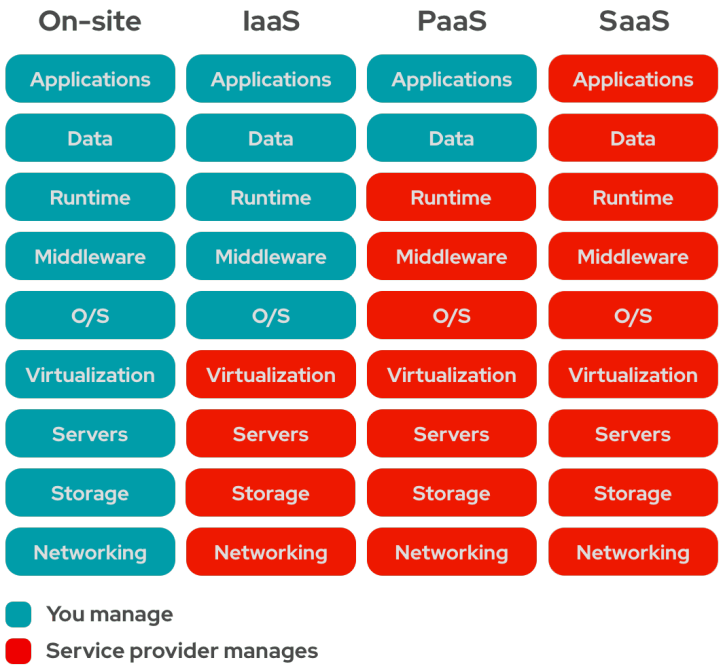

Figura 2.1: Separazione di responsabilità modelli *as a Service*

Nella produzione del software ci si riferisce a *Cloud Native* per indicare l'approccio per la creazione, l'implementazione e la gestione di applicazioni moderne in ambienti di cloud computing.

2.1.1 Serverless

Il *Serverless* è un modello di sviluppo *Cloud Native* che consente agli sviluppatori di progettare ed eseguire applicazioni senza doversi occupare direttamente della gestione dei server. Il termine "serverless" non indica l'assenza di server, bensì la loro astrazione dal processo di sviluppo delle applicazioni. In questo modello, il provider cloud gestisce attività quali il provisioning (cioè la creazione e configurazione dell'infrastruttura, compresi gli accessi degli utenti e dei sistemi alle risorse), la manutenzione e la scalabilità automatica dei server [12].
Spesso i termini *serverless architecture* e *serverless computing* sono erroneamente utilizzati come sinonimi, tuttavia fanno riferimento a concetti diversi.
La ***Serverless Architecture*** descrive l'approccio progettuale in cui le applicazioni sono eseguite soltanto quando sono necessarie. Quando un evento innesca l'esecuzione dell'applicazione, il provider alloca le risorse per quel codice. L'utente smette di pagare quando il codice finisce di eseguire, esistono sistemi di protezione che, nel caso in cui una funzione non termini entro un determinato intervallo di tempo, ne forzano l'interruzione automatica. Questo meccanismo è progettato per prevenire costi eccessivi dovuti a errori o esecuzioni anomale, evitando che l'utente debba sostenere spese impreviste derivanti da processi in esecuzione prolungata.
Il ***Serverless computing***, invece, fa riferimento al modello di sviluppo dell'applicazione. Esso differisce dagli altri modelli di cloud computing poichè il provider

assume la responsabilità sia della gestione dell'infrastruttura che della scalabilità dell'app, sollevando l'utente da tali compiti [12].
Quando parliamo di serverless, i principali servizi a cui facciamo riferimento sono **Backend-as-a-Service** (BaaS), un modello utilizzato principalmente per applicazioni web e mobile che offre un back-end completamente gestito (ad es. Firebase) e **Function-as-a-Service (FaaS)**.

2.1.2 Function-as-a-Service

Il ***Function-as-a-Service*** (FaaS) è un modello di *cloud computing* che permette agli sviluppatori di costruire ed eseguire applicazioni suddividendole in **funzioni**, senza occuparsi della gestione dell'infrastruttura sottostante. Una funzione, in questo contesto, rappresenta una porzione indipendente di codice che esegue una **logica di business**. Un'applicazione può essere composta da diverse funzioni.
FaaS si basa su un approccio **orientato agli eventi** (*event-driven*). In questo paradigma, le funzioni vengono eseguite esclusivamente quando si verificano specifici **eventi** o **trigger**, ad esempio **richieste HTTP**, **caricamento di file**, **aggiornamenti nei database** o altre condizioni predefinite. Ogni funzione rimane inattiva fino al verificarsi di un evento, momento in cui un'istanza della funzione viene automaticamente attivata ed eseguita. Questo riduce drasticamente i **costi operativi**, aumentando la **scalabilità** e l'**efficienza** del sistema.
La piattaforma esegue il codice dopo aver inizializzato un **ambiente di esecuzione**, ovvero un contesto **sicuro ed isolato** che fornisce tutte le risorse necessarie per il ciclo di vita della funzione, tipicamente implementato con **macchine virtuali** o **container** [5].

Amazon è stato il primo grande provider che, nel 2014 con **AWS Lambda**, ha creato la prima offerta FaaS da parte di un fornitore di cloud pubblico, seguita da **Google Cloud Functions**, **Microsoft Azure Functions**, **OpenWhisk** (*open-source*) di IBM e **Oracle Cloud Fn** [15].

2.1.3 Architettura Serverless FaaS

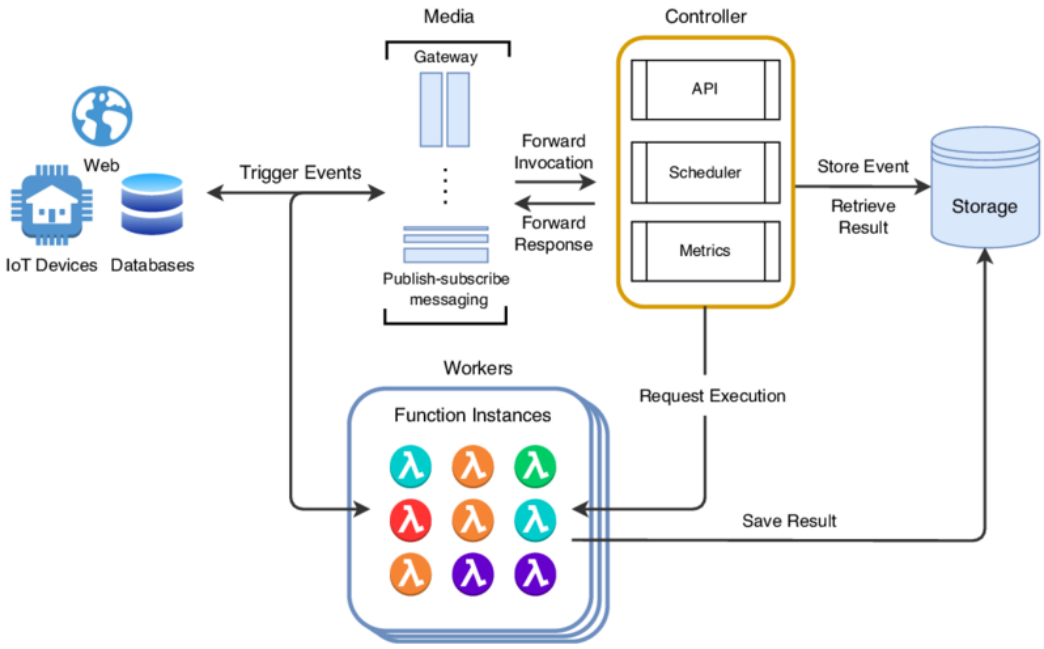

Figura 2.2: Una tipica piattaforma serverless

Come illustrato nella **Figura 2.2**, i principali componenti di una piattaforma *serverless* sono il ***Controller*** e i ***Workers***. Il Controller riceve le richieste di esecuzione delle funzioni tramite vari **media**, come ad esempio **API Gateway** o servizi di messaggistica *publish-subscribe*, tra cui il **Simple Notification Service (SNS)** di AWS.

Questi media espongono degli **endpoint** che possono essere invocati da **applicazioni web**, **dispositivi IoT** (*Internet of Things*), **database** o altre *funzioni serverless* per richiedere l'esecuzione di una specifica funzione.
Una volta ricevuta la richiesta, il Controller assegna l'esecuzione delle funzioni ai Workers, selezionando quelli più adatti sulla base di informazioni specifiche raccolte dal sistema (ad es. *utilizzo CPU e memoria*). Ciascun Worker, quindi, crea un'istanza della funzione, gestendone interamente l'**ambiente di esecuzione**, inclusi il **provisioning**, la **scalabilità automatica** e la successiva **dismissione** dell'ambiente stesso. [5]

Locality

Lo scheduling delle funzioni, ovvero l'allocazione delle funzioni sui *workers* disponibili, può influenzare in maniera sostanziosa le loro performance [5]. Un'allocazione inefficace può introdurre latenze elevate e ridurre l'efficienza complessiva del sistema. Lo scheduler dovrebbe prendere decisioni sul posizionamento delle funzioni considerando diversi tipi di *locality*: [14, 6].

- **Session Locality**: Se un'invocazione di una funzione fa parte di una sessione di lunga durata con connessioni TCP aperte, sarà utile eseguire la funzione sul worker in cui vengono mantenute tali connessioni. Questo evita che il traffico venga deviato attraverso un proxy, riducendo così la latenza e il consumo di risorse di rete.

- **Code Locality**: Uno scheduler consapevole del fatto che due diversi *workers* utilizzano gli stessi pacchetti o librerie può ottimizzare il posizionamento delle funzioni. Schedulare una funzione su un *worker* che ha già caricato

in memoria il pacchetto o la libreria necessaria riduce il tempo di avvio e minimizza la latenza associata al caricamento delle dipendenze.

- **Data Locality**: È particolarmente rilevante per funzioni che accedono a database o grandi insiemi di dati. Lo scheduler deve anticipare quali query o dataset verranno utilizzati da un'invocazione e cercare di eseguire la funzione il più vicino possibile a tali dati. La vicinanza spaziale tra codice e dati riduce significativamente i tempi di accesso, migliorando le prestazioni globali del sistema.

L'adozione crescente di architetture *serverless* ha portato alla necessità di strumenti che permettano la **coordinazione delle funzioni distribuite**. Poiché le funzioni **FaaS** sono *stateless* e vengono eseguite in **ambienti isolati**, diventa fondamentale disporre di meccanismi per il loro coordinamento, al fine di garantire la corretta esecuzione di **workflow complessi**.
Esistono due approcci principali per organizzare l'interazione tra i componenti: **orchestrazione** e **coreografia**.

Orchestrazione

L'**orchestrazione** si basa su un'entità centrale, chiamata **orchestratore**, che gestisce il **flusso di esecuzione** delle funzioni e coordina le loro interazioni. Questo approccio permette un **maggiore controllo** sul sistema e facilita il **monitoraggio**, ma introduce un **punto di centralizzazione** che può diventare un **collo di bottiglia**.
Esempi di orchestrazione nei sistemi *serverless* includono servizi come **AWS Step Functions** [13] e **Google Cloud Functions**.

Coreografia

La **coreografia**, invece, elimina la necessità di un **orchestratore centrale** e si basa su un modello **event-driven**, in cui le funzioni interagiscono tra loro attraverso **scambi di messaggi**, senza un'entità che controlli esplicitamente il flusso.

2.2 Programmazione coreografica

2.2.1 Coreografie e linguaggi coreografici

L'approccio coreografico utilizza le **coreografie**, schemi di **coordinamento** impiegati nei sistemi **concorrenti** e **distribuiti**, che definiscono chiaramente i **ruoli dei partecipanti** e le modalità con cui questi interagiscono [8].

Le coreografie sono descritte tramite appositi **linguaggi coreografici**, dotati di primitive specifiche che consentono di rappresentare le **interazioni** e la **comunicazione** tra processi distribuiti. Viene utilizzata una scrittura ispirata alla notazione "*Alice e Bob*" adottata nei **protocolli di sicurezza**, che prevede la primitiva `A.x -> B.y` per indicare la comunicazione di un messaggio `x` da `A` (Alice) a `B` (Bob), il quale verrà salvato nella variabile `y`.

Quando un linguaggio coreografico è sufficientemente espressivo da permettere la definizione completa e dettagliata di programmi distribuiti direttamente eseguibili, esso viene definito un **linguaggio di programmazione coreografico**, e il paradigma di programmazione in cui i programmi sono coreografie viene chiamato *programmazione coreografica* [8].

Nella Figura 2.3 è raffigurata una semplice coreografia che rappresenta il **protocollo di scambio di chiavi di Diffie-Hellman**, scritto utilizzando la notazione

menzionata precedentemente.

```
Alice.modPow(g,a,p) -> Bob.x;
Bob.modPow(g,b,p) -> Alice.y;
```

Figura 2.3: Coreografia Diffie-Hellman

Uno degli aspetti più interessanti della **programmazione coreografica** è il suo approccio **correctness-by-construction**. Ciò permette di individuare **errori nel design** fin dalle prime fasi di sviluppo e, se una coreografia è specificata correttamente, questa garantirà la **coerenza dell'interazione** fra i partecipanti.

2.2.2 Proiezione endpoint

Tipicamente, i **linguaggi di programmazione coreografici** sono accompagnati da un **compilatore**, che traduce le coreografie in **codice eseguibile** per sistemi **concorrenti** e **distribuiti** [8]. Questo processo viene chiamato ***proiezione endpoint*** (*endpoint projection* o *EPP*) e gioca un ruolo fondamentale nello sviluppo di compilatori per linguaggi coreografici.

Partendo da una descrizione delle **interazioni** tra i vari **ruoli** coinvolti, la proiezione produce il **codice per ciascun ruolo** (o **endpoint**) in maniera **coerente** e **priva di ambiguità**.

La Figura 2.4 mostra la **proiezione** della coreografia precedentemente illustrata in Figura 2.3, rappresentando separatamente il **codice destinato ad Alice** e quello **destinato a Bob**.

Poiché, in generale, una coreografia può coinvolgere **numerosi partecipanti**, la **proiezione endpoint** deve essere progettata per **distribuire correttamente** il

```
Codice per Alice:
send modPow(g, a, p) to Bob;
recv y from Bob;

Codice per Bob:
recv x from Alice;
send modPow(g, b, p) to Alice;
```

Figura 2.4: Proiezione Diffie-Hellman in pseudocodice

codice in molteplici programmi eseguibili, ciascuno corrispondente a uno specifico **partecipante** [10].

2.3 FaaSChal

Un problema significativo nello sviluppo di architetture *serverless* è l'assenza di un **modello unificato** per la definizione delle **funzioni** e dei **flussi di lavoro**, il che comporta l'adozione di **linguaggi differenti** per la loro specifica. Inoltre, l'inferenza automatica della *locality* a partire dal codice risulta complessa, causando **inefficienze** nella distribuzione sia del **codice** che dei **dati**.
Per affrontare questa mancanza, è stato proposto ***FaaSChal***, un progetto per la creazione di un **linguaggio di programmazione coreografico** pensato specificamente per le **funzioni serverless FaaS** [5]. Esso **unifica** la definizione delle funzioni con la gestione del **workflow** e introduce una nuova applicazione della **programmazione coreografica**. In particolare, utilizza la **coreografia** come fonte per l'estrazione della *locality*, con l'obiettivo di **ottimizzare le prestazioni** del sistema.

```
1  stateful: Alice, Bob
2
3  import int::modPow@(Alice,Bob)
4
5  def main(par@Alice, par@Bob):
6      modPow(par.g,par.a,par.p)@Alice <-> a@Bob
7      modPow(par.g,par.b,par.p)@Bob <-> b@Alice
8  end
```

Figura 2.5: Coreografia Diffie-Hellman in FaaSChal

Nella Figura 2.5 è riportato un **esempio preliminare** di **coreografia** scritta utilizzando **FaaSChal**, riprendendo lo scenario illustrato in precedenza nella Figura 2.3. Questo esempio permette di osservare la **struttura** di una coreografia scritta utilizzando il linguaggio, evidenziando la **definizione dei processi attivi** coinvolti nella comunicazione, in questo caso `Alice` e `Bob`, e il meccanismo di **scambio dei messaggi** tra di essi.

Un'analisi più approfondita delle **entità coinvolte** e delle **dinamiche di comunicazione** sarà trattata nel Capitolo 3.

Capitolo 3

Analisi Design FaaSChal

3.1 Introduzione

In questo capitolo presento il mio contributo al progetto, illustrando una serie di esempi sviluppati per mostrare le principali caratteristiche del linguaggio **FaaSChal**. La complessità degli esempi aumenta gradualmente, esplorando di volta in volta le principali tematiche da affrontare nello sviluppo di un linguaggio coreografico di questa tipologia.

3.2 Esempi di coreografie in FaaSChal

3.2.1 Coreografia "Hello World con insert asincrono"

La prima routine presentata ha inizio con un processo `user`, il quale definisce una variabile `message` e la invia a una funzione stateless `f`. Quest'ultima è resa disponibile tramite il *MEDIUM Amazon SNS* e, una volta invocata, inserisce il

messaggio all'interno di un database attraverso il servizio `DB` e la relativa funzione `insert`.

Questa prima coreografia ha lo scopo di mostrare le principali entità del linguaggio e di illustrare la struttura che una coreografia scritta in **FaaSChal** può assumere.

```
1  stateful: user
2  stateless: f{SNS: "aws:sns"}
3  services: DB{insert}
4
5  def main():
6      message@user = "Hello from user "@user
7      message@user -SNS-> f | msg@f |
8          msg@f -> DB: insert
9      end
10 end
```

Figura 3.1: Coreografia "Hello World con insert asincrono" in FaaSChal

All'interno di una coreografia **FaaSChal**, possiamo distinguere tre tipi di entità [5]:

- **Stateful**: entità attive all'interno della coreografia, ovvero processi che hanno la possibilità di interagire con altre entità.
- **Stateless**: funzioni *FaaS* che devono:
  - essere attivate da un'altra entità attiva che ha a disposizione il loro **endpoint**, dichiarato nella loro definizione;
  - fornire un comportamento *request-response*, dove il chiamante ha la possibilità di scartare la risposta;
  - dopo l'attivazione, l'istanza generata non può ricevere ulteriori messaggi, ma può mandare richieste in uscita.

- **Services**: entità passive che interagiscono nella coreografia fornendo un comportamento *request-response*. Questi includono servizi come API per il meteo, server di posta o database condivisi.

In **FaaSChal**, una chiamata *request-response* segue la seguente sintassi:

```
exp1@role1 <- MEDIUM -> role2 do [| opt_var@role2 |] ... end [ with exp2@role2 ]
```

Da sinistra a destra, l'espressione `exp1` viene valutata presso `@role1` e inviata attraverso il `MEDIUM`, su cui la funzione è disponibile, per attivare l'esecuzione della funzione associata a `role2`.
La funzione può, opzionalmente, associare i dati ricevuti a una variabile locale (`opt_var`) ed eseguire il codice contenuto nel blocco fino alla chiusura (`end`). Infine, la funzione può restituire una risposta vuota o, se specificato, il risultato dell'espressione `exp2@role2` tramite la clausola `with`.
È importante sottolineare l'**asincronia** della chiamata alla funzione stateless, descritta dal meccanismo di comunicazione illustrato alla riga 8 della Figura 3.1 (`-SNS->`). Concretamente, l'utente invia un messaggio alla funzione **f** senza attendere una risposta; a seconda del tipo di asincronia adottato, l'utente potrebbe persino non attendere la conferma della corretta ricezione del messaggio.

3.2.2 Coreografia "Hello World con retrieve sincrono"

A differenza del precedente esempio, questa coreografia illustra il meccanismo delle **chiamate sincrone**, evidenziando come una comunicazione possa avvenire in maniera **bloccante**, imponendo all'entità chiamante di **attendere la risposta** prima di poter proseguire con l'esecuzione successiva. Questo comportamento è determinato dal meccanismo di comunicazione illustrato alla riga 7 della Figura 3.2

(`<- Gateway ->`), che vincola il processo `user` a sospendere la propria esecuzione fino all'ottenimento della risposta.

Successivamente, la funzione stateless `g` interroga il database per recuperare un messaggio precedentemente memorizzato. Una volta completata questa operazione, il messaggio viene **restituito all'utente**, il quale avrà la possibilità di **processare i dati** a suo piacimento.

```
1  stateful: user
2  stateless: g {Gateway}
3  services: DB{retrieve}
4
5  def main():
6
7          Unit@user <-Gateway-> g
8              Unit@g <-> DB: retreive |> message@g
9          end with message@g |> message@user
10
11 end
```

Figura 3.2: Coreografia "Hello World con retrieve sincrono" in FaaSChal

3.3 Processo di proiezione e deployment

FaaSChal, in quanto linguaggio coreografico, supporta la cosiddetta **proiezione endpoint** (Sezione 2.2.2), ovvero la generazione automatica del **codice locale** che implementa la semantica della coreografia di partenza [5].

Dal momento che esistono numerosi linguaggi con cui è possibile sviluppare architetture *stateless FaaS*, si è posto l'obiettivo di creare un **compilatore multi-target**, offrendo allo sviluppatore l'opportunità di specificare, mediante delle **annotazioni**, quale linguaggio di destinazione adottare.

La caratteristica che rende **FaaSChal** unico rispetto agli altri linguaggi coreografici è la capacità di supportare lo sviluppatore nel processo di **deployment**, automatizzandolo tramite un **deployer dedicato**. L'obiettivo è rendere il linguaggio **indipendente dalla piattaforma**, permettendo agli sviluppatori di scegliere quella più adatta alle proprie esigenze.
Per rendere possibile tale indipendenza, occorre integrare delle informazioni che non sono deducibili direttamente dalla coreografia. Si tratta di **metadati** che forniscono dettagli essenziali, come gli **endpoint** dei servizi impiegati, eventuali **chiavi di autenticazione** e parametri di **configurazione** per il servizio di deployment adottato.
Questi metadati possono essere:

- **Esternalizzati** in un file di configurazione, redatto con un formato appropriato come **TOML** (*Tom's Obvious Minimal Language*), permettendo di separare le responsabilità e mantenere **leggera la sintassi della coreografia**.

- **Inclusi direttamente nel codice**, semplificando il **parsing** e riducendo il rischio di **inconsistenze**.

Per sperimentare la fase di deployment, mi sono avvalso di **AWS Lambda**, utilizzando **AWS CLI** (Sezione 3.3.1) e della piattaforma **LocalStack** (Sezione 3.3.2), per simulare i servizi AWS in locale. Di seguito, presento brevemente questi strumenti e successivamente mostrerò come sono stati integrati nel progetto.

3.3.1 AWS Command Line Interface

La **AWS Command Line Interface (AWS CLI)** è uno strumento a riga di comando offerto da **Amazon Web Services**. Consente di interagire con i servizi AWS utilizzando comandi shell direttamente dalla linea di comando.
Con una configurazione minima, AWS CLI permette di eseguire comandi che implementano funzionalità equivalenti a quelle fornite dalla **Console di gestione AWS** basata su browser [2]. L'uso di AWS CLI consente di **gestire e configurare** un'ampia collezione di **servizi cloud**, semplificando lo sviluppo e il deployment di applicazioni serverless.

3.3.2 LocalStack

LocalStack è una piattaforma che fornisce un **ambiente di test locale** per molteplici servizi **Amazon Web Services** [7].
Grazie all'utilizzo di **container Docker**, librerie di emulazione e meccanismi di instradamento delle richieste, LocalStack è in grado di **intercettare e replicare** il normale flusso di richieste ai servizi AWS, simulandone il comportamento. Questo permette agli sviluppatori di eseguire attività di **sviluppo e testing** di soluzioni cloud direttamente in locale, senza necessità di accedere a un'infrastruttura cloud reale.

3.3.3 Proiezione "Hello World con insert asincrono"

Presentiamo la **proiezione** della coreografia illustrata in Figura 3.1. Da essa derivano due blocchi di codice: uno per la parte relativa all'**utente** e l'altro per la **logica della funzione stateless** `f`.

Poiché i `servizi` operano come **entità passive**, il codice locale generato non comprende l'implementazione di `DB`.

Dal momento che il **compilatore** è ancora in fase di sviluppo, mostrerò degli esempi **generati manualmente**, evidenziando le decisioni che il compilatore dovrà essere in grado di prendere per la traduzione da **coreografia a codice locale**.

Ho scelto di utilizzare come linguaggio di riferimento per gli esempi **JavaScript**, in quanto è uno dei linguaggi più diffusi nella creazione di **architetture serverless**, grazie alla sua vasta disponibilità di moduli e alla sua natura **asincrona**.

```
//codice user
const { SNSClient, PublishCommand } = require('@aws-sdk/client-sns');

const snsClient = new SNSClient({
  region: 'us-east1',
  endpoint: 'http://localhost:4566',
});

const topicArn = 'arn:aws:sns:us-east-1:000000000000:fTopic';
const message = "Hello World"

const params = {
  Message: JSON.stringify({ msg: message }),
  TopicArn: topicArn,
};

const publishCommand = new PublishCommand(params);

snsClient.send(publishCommand)
  .then()
```

```
//funzione stateless f
const url = "http://host.docker.internal:5000/insert"; //endpoint DB {insert}

exports.handler = (event) => {
    const message = JSON.parse(event.Records[0].Sns.Message);
    fetch(url, {
        method: 'POST',
        headers: {"Content-Type': 'application/json"},
        body: JSON.stringify({ msg: message })
    });

    return;
};
```

Figura 3.3: Proiezione coreografia "Hello World" con insert asincrono

Nel codice generato per la funzione **stateless** `f`, possiamo osservare che l'invocazione della funzione **insert**, fornita dal servizio `DB`, avviene tramite la **Fetch**

API. Questa API nativa di *JavaScript* consente di effettuare **richieste HTTP** e gestirne le risposte.

Per quanto riguarda il **codice utente**, invece, notiamo l'importazione della libreria **AWS SDK** (*Software Development Kit*) per *JavaScript*, che permette l'interazione con i servizi **AWS**. In questo caso specifico, la libreria viene impiegata per **pubblicare un messaggio** su un **topic SNS**, in conformità con quanto definito nella coreografia, la quale specifica che la funzione è accessibile attraverso il *medium* **SNS**.

Di seguito, mostro come **AWS CLI** e **LocalStack** sono stati integrati nel mio lavoro. I passaggi illustrati rappresentano le operazioni che il **deployer automatico** dovrà eseguire per implementare correttamente la funzione **stateless** `f`, descritta all'interno della coreografia:

1. **Creazione funzione Lambda**. Il primo passo consiste nella creazione della funzione `f`. Sono necessari parametri come il runtime, necessario per eseguire il codice. Viene indicato il ruolo IAM (AWS Identity and Access Management) che la funzione utilizzerà per ottenere i permessi necessari all'esecuzione. Il parametro `-handler` definisce il punto di ingresso della funzione, mentre `-zip-file` indica il percorso del pacchetto contenente il codice sorgente e le dipendenze.

```
awslocal lambda create-function \
    --function-name f \
    --runtime nodejs18.x \
    --role arn:aws:iam::000000000000:role/execution_role \
    --handler index.handler \
    --zip-file fileb://function.zip
```

2. **Creazione del topic SNS**. Una volta definita la funzione Lambda, è necessario creare un topic SNS `fTopic` che fungerà da `MEDIUM` per la funzione `f`.

```
awslocal sns create-topic --name fTopic
```

3. **Iscrizione della funzione Lambda**. Infine, si procede alla sottoscrizione della funzione stateless `f` al topic SNS, in modo che possa ricevere i messaggi pubblicati su di esso. Il parametro `-topic-arn` identifica il topic a cui la funzione verrà associata, mentre `-protocol` indica il metodo di comunicazione, in questo caso `lambda`. Infine, `-notification-endpoint` specifica l'ARN (Amazon Resource Name) della funzione che dovrà essere notificata ogni volta che un messaggio viene pubblicato sul topic, ottenibile dopo la creazione tramite il comando `get-function`.

```
awslocal sns subscribe \
--topic-arn arn:aws:sns:us-east-1:000000000000:fTopic \
--protocol lambda \
--notification-endpoint
↪  arn:aws:lambda:us-east-1:000000000000:function:f
```

3.3.4 Proiezione "Hello World retrieve sincrono"

```
//funzione stateless f
const url = "http://host.docker.internal:5000/retrieve"; //endpoint DB
↪ {retrieve}
exports.handler = async () => {
    response = await fetch(url, {
        method: 'GET',
        headers: {"Content-Type': 'application/json"},
    });
    response  = await response.json()
    return { statusCode: 200, body: JSON.stringify(response) };
};
```

```
//codice user

const gUrl = "http://localhost:4566/g" //endpoint g

(async () => {
    const response = await fetch(gUrl, {
        method: "GET",
        headers: { "Content-Type": "application/json" }
    });

    const data = await response.json();
    return data
})();
```

Figura 3.4: Proiezione coreografia "Hello World con retrieve sincrono"

La seguente proiezione è molto simile a quella generata in Figura 3.3, con l'unica differenza rappresentata dal MEDIUM tramite cui è resa disponibile la funzione stateless g all'interno della coreografia, ovvero *Gateway*.

Di conseguenza, nel codice utente non è necessario importare la libreria AWS SDK, a differenza dell'esempio precedente. In questo caso, l'unica informazione di cui il codice utente ha bisogno per eseguire la funzione è il suo endpoint, che

verrà ottenuto tramite il *deployer* una volta che la funzione Lambda sarà stata correttamente caricata sul servizio scelto.
Un'ulteriore differenza rispetto all'esempio precedente è l'utilizzo della notazione `async/await`, coerente con il fatto che nella coreografia la comunicazione tra `user` e `f` avviene in modo sincrono. Questo garantisce che il codice utente attenda la risposta prima di proseguire con l'esecuzione.

3.4 Ulteriori esempi

3.4.1 Coreografia "Bookseller Protocol"

Uno degli aspetti piu' caratterizzanti dei linguaggi coreografici è la **Knowledge of Choice**. Una coreografia ha la proprietà della Knowledge of choice quando si assicura che, quando viene effettuata una scelta tra due diversi comportamenti, tutti i partecipanti agli esiti di quella scelta siano informati della decisione effettuata [8]. L'esempio mostrato in Figura 3.5 rappresenta una coreografia in cui un **buyer** richiede la disponibilità di un libro comunicando il relativo `bookTitle` al **seller**. Quest'ultimo interroga un servizio esterno per verificare la disponibilità del libro. Se il libro è disponibile, il **buyer** procede al pagamento tramite un servizio bancario esterno. L'esito della transazione viene poi comunicato al **seller**, che, una volta confermata la corretta elaborazione del pagamento, avvia la spedizione del libro attraverso un servizio di logistica.

```
stateful: buyer, seller
services: BookDB {checkBook, getRestockDate }
          Bank { processPayment }
          Shipping { shipBook }

def main(queries@buyer):

    queries.bookTitle@buyer -> bookTitle@seller

    bookTitle@seller <-> BookDB: checkBook |> available@seller
    if(available@seller)
        then
            queries.credentials@buyer <-> Bank: processPayment |>
            ↪  result@buyer
            result@buyer -> result@seller
            (result,bookTitle)@seller <-> Shipping: shipBook

        else
            queries.book@buyer <-> BookDB: getRestockDate

end
```

Figura 3.5: Coreografia "Bookseller Protocol" senza Knowledge of Choice in FaaSChal

Nel caso in cui il libro non sia disponibile, invece di avviare il processo di acquisto, il **buyer** contatta direttamente il servizio per ottenere una data indicativa del prossimo rifornimento.

Come si può osservare, la **guardia** del condizionale if viene valutata dal **seller**, e il **buyer** esegue due comportamenti distinti a seconda del risultato. La coreografia in questo stato non garantisce che il **buyer** sia esplicitamente informato dell'esito della condizione. Di conseguenza, il **buyer** non ha conoscenza diretta del percorso eseguito nella coreografia dopo la valutazione della guardia, il che implica che

questa coreografia **non soddisfa la proprietà di Knowledge of Choice**.

La mancanza della *Knowledge of Choice* ha effetti significativi nella pratica. In questo caso, costringerebbe il compilatore a individuare un meccanismo alternativo per garantire che il **buyer** venga a conoscenza della scelta effettuata. Tuttavia, ciò comprometterebbe la rappresentatività della coreografia, poiché l'effettiva proiezione includerebbe comunicazioni non descritte esplicitamente nella routine originale [9].

Il processo che permette la riparazione di coreografie di cui non è possibile fare la proiezione dato il problema della knowledge of choice si chiama **amendment** [3]. Per rendere la coreografia in Figura 3.5 corretta, è sufficiente introdurre una comunicazione che permetta al **buyer** di dedurre quale azione intraprendere. Questo può essere ottenuto attraverso l'uso delle **selezioni**. Una selezione è una comunicazione che trasporta dei valori costanti, chiamati *selection labels* [4], spesso rappresentati da un'istanza di un tipo `enum` che descrive i possibili esiti della valutazione della guardia. [9]

```
stateful: buyer, seller
services: BookDB {checkBook, getRestockDate }
          Bank { processPayment }
          Shipping { shipBook }

def main(queries@buyer):

    queries.bookTitle@buyer -> bookTitle@seller

    bookTitle@seller <-> BookDB: checkBook |> available@seller
    if(available@seller)
        then
            enum[left]@seller -> exp@buyer
            queries.credentials@buyer <-> Bank: processPayment |>
            ↪  result@buyer
            result@buyer -> result@seller
            (result,bookTitle)@seller <-> Shipping: shipBook

        else
            enum[right]@seller -> exp@buyer
            queries.book@buyer <-> BookDB: getRestockDate

end
```

Figura 3.6: Coreografia "Bookseller Protocol" con Knowledge of Choice in Faa-SChal

La coreografia in Figura 3.6 è diversa dalla precedente data la presenza di selezioni in ogni branch del condizionale. Nel ramo `then`, il **seller** comunica esplicitamente `left` al **buyer**, mentre nel ramo `else` invia l'etichetta `right`. In questo modo, il **buyer** è informato della scelta effettuata dal **seller** e può eseguire l'azione appropriata in base all'esito ricevuto. Siccome le label sono **costanti**, il compilatore sarà in grado di verificare che il **buyer** riceverà differenti labels per differenti branch e sarà in grado di sapere in quale direzione la coreografia sta andando.

Una caratteristica notevole di FaaSChal è quella che non c'è bisogno di coordinazione tra i processi in costrutti come i cicli e i condizionali quando l'interazione coinvolge solamente funzioni **stateless** [5].
Consideriamo il seguente esempio per chiarificare il rapporto tra le funzioni stateless e la knowledge of choice

```
if exp@f then f -SNS-> g else f -SNS-> h
```

Se `g` e `h` fossero processi *stateful*, sarebbe necessario informare entrambi sulla direzione della coreografia. Tuttavia, poiché tutti i ruoli sono **stateless**, non è necessario notificare `g` nel caso in cui `f` scelga il ramo `else`, dato che `g` non viene eseguito fino a quando non viene invocato da `f`, evitando così il rischio di deadlock.

3.4.2 Coreografia "Registrazione utente"

```
stateful: user
stateless: a {Gateway}, b {Gateway}
services: DB{checkUser, regUser}

import JS::bcryptjs::genSalt as gensalt@b
import JS::bcryptjs::hash as hashpw@b

def main(credentials@user) {

    credentials@user <-Gateway-> a do |credentials@a|
        credentials@a.username <-> DB: checkUser |> userExists@a
        if (userExists@a)
            then
                call_status@a = "User Already Exists"@a
            else
                credentials@a.password <-Gateway-> b |pwd@b|
                    gensalt() |> salt@b
                    hashpw(salt@b,pwd@b) |> encPwd@b
                end with encPwd@b |> encPwd@a
                (encPwd@a, credentials@a.username) <-> DB: regUser
                call_status@a = "Registered Successfully"@a
    end with call_status@a |> call_status@user
end
}
```

Figura 3.7: Coreografia "Registrazione utente" in FaaSChal

Questa coreografia descrive un **caso d'uso reale** di una routine di **registrazione** di un utente a un servizio.

La funzione **stateless** `a` verifica l'eventuale esistenza dell'utente tramite la funzione `checkUser`, la quale restituisce un valore booleano (**flag**). Se tale flag è **vero**, significa che l'utente è già presente nel sistema e viene restituito un **messaggio di avviso**; in caso contrario, si invoca la funzione **stateless** `b`, che ha il compito

di **crittografare la password** e completare la procedura di registrazione tramite la funzione `regUser`.

In questo esempio mostriamo come sia possibile **importare operazioni** (ad es., da librerie esterne) all'interno della coreografia. Come illustrato alle righe 4-7, la funzione **stateless** `b` utilizza l'istruzione `import` per accedere alle funzioni `encode`, `gensalt` e `hashpw` della libreria `bcryptjs`. Poiché l'istruzione `import` è legata a una specifica funzione target (ad es., `b`), l'operazione è disponibile **esclusivamente** all'interno di quella funzione [5].

Di seguito, nelle Figure 3.8 e 3.9, sono riportate le **proiezioni endpoint** dei partecipanti all'interno della coreografia in esame.

```
1  // Codice user
2  const aUrl = "http://localhost:4566/a"; // endpoint a
3  (async (credentials) => {
4      call_status = await fetch(aUrl, {
5          method: "POST",
6          headers: { "Content-Type": "application/json" },
7          body: JSON.stringify(credentials)
8      });
9
10     return;
11 })();
```

Figura 3.8: Proiezione codice utente della coreografia "Registrazione Utente"

Osservando la proiezione, è possibile individuare tutti gli elementi discussi in precedenza. Tutte le funzioni **stateless** (`a`, `b`) sono rese disponibili tramite il *MEDIUM* **Gateway**, il che implica la presenza dei loro **endpoint** nel codice, consentendo l'invocazione delle funzioni attraverso la funzione `fetch` di *JavaScript*.

Si può inoltre notare come, in questo caso, siano presenti dei **parametri di chiamata** e come essi vengano trasmessi correttamente all'interno del corpo della richiesta HTTP (*body*), utilizzando il formato **JSON**, che rappresenta lo standard più appropriato per questo tipo di comunicazione.
Un ulteriore aspetto di rilievo riguarda la funzione **stateless** `b`, nella quale vengono importate correttamente le funzioni specificate dalla coreografia. Poiché tali importazioni hanno come target esclusivamente la funzione `b`, le funzionalità importate risultano **accessibili unicamente** all'interno del codice a essa associato, garantendo un **corretto isolamento** e una **chiara separazione delle responsabilità**.

```
//funzione stateless a
const checkUrl = "http://host.docker.internal:5000/checkUser"; //endpoint DB {checkUser}
const regUrl = "http://host.docker.internal:5000/regUser"; //endpoint DB {regUser}
const bUrl = "http://localhost:4566/b" //endpoint b

exports.handler = async (credentials) => {
    userExists = await fetch(checkUrl, {
        method: 'POST',
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify(credentials?.username)
    });
    if (userExists) {
        call_status = "User Already Exists"
    }
    else {
        encPwd = await fetch(bUrl, {
            method: "POST",
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify(credentials?.password)
        });
        await fetch(regUrl, {
            method: 'POST',
            headers: { "Content-Type": "application/json" },
            body: JSON.stringify(credentials?.username, encPwd)
        });
        call_status = "Registered Successfully"
    }
    return { statusCode: 200, body: JSON.stringify(call_status) };
}
```

```
//funzione stateless b
import {genSalt, hash} from bcryptjs
exports.handler = async (password) => {
    const salt = await genSalt()
    const hash = await hash(password,salt)
    return { statusCode: 200, body: JSON.stringify(hash) };
};
```

Figura 3.9: Proiezione funzioni stateless della coreografia "Registrazione Utente"

Capitolo 4

Conclusioni e sviluppi futuri

In questa tesi ho presentato un'analisi del linguaggio di programmazione coreografico *FaaSChal*, attualmente in fase di sviluppo. Le coreografie illustrate hanno permesso di evidenziare le caratteristiche del linguaggio, partendo da esempi introduttivi che ne presentano la sintassi e il meccanismo di comunicazione tra le entità coinvolte. Successivamente, è stata analizzata la proiezione endpoint, che consente di generare codice locale a partire dalla coreografia di partenza, e l'elemento distintivo che rende unico il linguaggio: il deployment automatico delle funzioni *stateless* derivate dalla coreografia.

Particolare attenzione è stata dedicata al ruolo del compilatore e del *deployer*, analizzando le operazioni necessarie per la generazione e l'implementazione delle funzioni, nonché agli strumenti impiegati, come AWS Lambda e LocalStack, utilizzati per testare gli esempi sviluppati in un ambiente locale. Inoltre, è stata affrontata la problematica della *Knowledge of Choice*, una proprietà fondamentale nei linguaggi coreografici. Sebbene nelle interazioni che coinvolgono entità *stateless* questa caratteristica non sia rilevante, nelle comunicazioni tra elementi

stateful è essenziale garantirne la corretta gestione. A tal fine, è stata proposta una soluzione basata sulle selezioni, ovvero comunicazioni di etichette costanti che consentono ai partecipanti di comprendere la direzione della coreografia, assicurando coerenza nell'esecuzione distribuita. Tra i possibili sviluppi futuri, un aspetto centrale riguarda l'ottimizzazione del codice attraverso l'inlining, una tecnica che incide direttamente sui tempi di esecuzione e sull'uso della memoria. La sua applicazione deve essere guidata da euristiche ben definite: la mancanza di inlining può introdurre costi di chiamata delle funzioni non necessari, mentre un utilizzo eccessivo può generare codice ridondante e aumentare il consumo di memoria. Lo sviluppo di strategie automatiche per determinare l'applicazione ottimale di questa trasformazione rappresenta un'area di ricerca di grande interesse, con potenziali miglioramenti ottenibili tramite l'analisi statica e dinamica del codice.

Un altro elemento chiave per l'evoluzione del sistema riguarda l'integrazione delle informazioni sui servizi, come indirizzi, endpoint e protocolli di comunicazione. La questione principale è stabilire se tali informazioni debbano essere gestite attraverso file esterni (TOML, YAML o JSON) o incluse direttamente nel codice tramite una sintassi dedicata. L'uso di file esterni offre maggiore flessibilità e separazione tra dati e logica applicativa, mentre l'inserimento diretto nel codice potrebbe semplificare il parsing e ridurre il rischio di inconsistenze. La ricerca di un equilibrio tra queste due soluzioni è fondamentale per migliorare la configurabilità e la manutenibilità del sistema.

Infine, l'obiettivo ultimo di questo lavoro è la progettazione e l'implementazione di un compilatore e di un deployer, con un'attenzione particolare all'efficienza, alla scalabilità e alla coerenza delle comunicazioni distribuite.

Bibliografia

[1] Amazon Web Services. Cos'è il cloud privato? `https://aws.amazon.com/it/what-is/private-cloud/`.

[2] Amazon Web Services. *AWS Command Line Interface - User Guide*. Amazon Web Services, 2023.

[3] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party sessions. In Roberto Bruni and Juergen Dingel, editors, *Formal Techniques for Distributed Systems*, pages 1–28, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[4] Luís Cruz-Filipe and Fabrizio Montesi. Now it compiles! certified automatic repair of uncompilable protocols. `https://arxiv.org/abs/2302.14622`, 2023.

[5] Giuseppe De Palma, Saverio Giallorenzo, Jacopo Mauro, Matteo Trentin, and Gianluigi Zavattaro. Towards a function-as-a-service choreographic programming language: Examples and applications. *arXiv preprint*, arXiv:2406.09099, 2024.

[6] Andrea Fenati. *Data Locality in Serverless Computing*. PhD thesis.

[7] LocalStack. *LocalStack: A fully functional local AWS cloud stack*, 2025.

[8] Fabrizio Montesi. *Introduction to Choreographies*. Cambridge University Press, 2023.

[9] Fabrizio Montesi. Knowledge of choice. `https://www.fabriziomontesi.com/bliki/KnowledgeOfChoice`, 2023.

[10] Fabrizio Montesi. Choreographic programming. `https://www.fabriziomontesi.com/bliki/ChoreographicProgramming`, 2024.

[11] Red Hat. Iaas vs. paas vs. saas. `https://www.redhat.com/it/topics/cloud-computing/iaas-vs-paas-vs-saas`, 2025.

[12] Red Hat. What is serverless? `https://www.redhat.com/en/topics/cloud-native-apps/what-is-serverless`, 2025.

[13] Amazon Web Services. Aws step functions - developer guide. 2023.

[14] Mohammad Shahrad, Rodrigo Fonseca, Bruce Maggs, and Mor Harchol-Balter. Architectural implications of function-as-a-service computing. *Proceedings of the ACM on Measurement and Analysis of Computing Systems (SIGMETRICS)*, 4(1):1–26, 2020.

[15] Wikipedia. Function as a service — wikipedia, l'enciclopedia libera. `http://it.wikipedia.org/w/index.php?title=Function_as_a_service&oldid=140379534`, 2024.