

# **Un'euristica distribuita per il controllo dei flussi di richieste in ambiente FaaS: Un approccio basato su simulazione**

**Relatore:** Dr. Michele Ciavotta

**Correlatore:** Dott. Alessandro Tundo

**Tesi di Laurea Magistrale di:**  
Mattia Vincenzi  
Matricola 860579

**Anno Accademico 2020-2021**



*Ai miei genitori, Barbara e Andrea, che nonostante tutto, siete il mio modello di riferimento, tutto quello che sono lo devo a voi, e quindi questo traguardo è soprattutto per voi. Mamma non so come hai fatto a sopportarmi e non uccidermi mentre ripeteva uno dei mille mila esami. Babbo non so come hai fatto a non uccidere la mamma che si lamentava di me, scherzo, spero di diventare un quarto di te un giorno. Vi siete fatti il culo per permettermi di fare tutto questo, grazie per tutto, non lo dirò mai abbastanza.*

*Alla mia bellissima morosa, Arli, sei la mia roccia e la mia forza, senza le tue 'cazziate' non so se mi sarei mai rimesso in carreggiata, quindi se ho finito è anche grazie a te; anche se ammetto che la scorsa estate sei stata la mia più grande distrazione :). Mi ricordo come fosse ieri quando non mi hai parlato per 2 giorni per farmi capire che dovevo muovermi. Sono davvero fortunato che tu sia qui con me, non smetterò mai di ringraziarti di camminare fianco a fianco; a mille di questi giorni. Ti amo, anche se non è abbastanza.*

*(P.S. Metto le mani avanti, ma sono sicuro sarai bellissima! Nonostante sia da ottobre che ti chiedi cosa ti devi mettere)*

*Ai miei nonni e alla mia famiglia, a chi c'è ancora e a chi mi ha lasciato in questo ultimo anno di percorso, vi ringrazio per tutto l'appoggio che mi avete dato (anche economico ;)).*

*Ai miei amici, siete un branco di pazzi scatenati, ma vi si vuole bene. A Miki, Panza, Zullo, Rei, Cero, Bigu, Matti... siete la mia seconda famiglia. Miki sei come un fratello. Panz non credo servano le parole. Zul è come se fossi rimasto il mio compagno di banco dalla prima superiore. Rei, senza il mio braccio destro della triennale non avrei passato la metà degli esami anche in magistrale.*

*Ad Alessio, il bhrosky dell'uni, hai fatto sembrare le 5 di notte sui progetti meno disperate.*

*A Francesco, sei stato un collega, anzi un amico, che mi ha aiutato tantissimo ancor prima di conoscerci, e ancora di più dopo esserci conosciuti, grazie.*

*A Davide e Frank che nonostante la distanza abbiamo condiviso un bellissimo percorso.*

*Anche a chi non è più al mio fianco ma che mi ha accompagnato ed iniziato con me questo percorso. Nonostante tutto, grazie.*

*Al prof. Ciavotta e al prof. Tundo, grazie non solo per avermi accompagnato nel lavoro svolto ma per quanto mi avete trasmesso a livello accademico e umano e per tutte le opportunità che mi avete offerto.*



# Abstract

Il modello del Cloud Computing ha abilitato la nascita di diversi paradigmi e, negli ultimi anni, tra di essi è nato il Serverless Computing. Una declinazione di questo paradigma è quella di Function as a Service, ovvero un modello di servizio che descrive applicazioni cloud-native la cui logica è scritta sotto forma di funzioni molto limitate. FaaS può essere applicato agevolmente nel contesto dell'edge computing, per via della facilità di deployment delle applicazioni realizzate seguendo tale paradigma. Inoltre, la natura granulare delle funzioni serverless consente di gestire più accuratamente problematiche relative al bilanciamento del carico. In questa trattazione si è approfondito questo problema in un ambiente dinamico ed eterogeneo come quello dell'edge computing, definendo un algoritmo distribuito per il bilanciamento delle richieste da un agente verso gli altri raggiungibili. Tale algoritmo ideato prende il nome di **empirical strategy** proprio perché è basato su un insieme di metriche che sintetizzano lo stato di salute del nodo su cui l'agente è in esecuzione e delle singole funzioni presenti sulla piattaforma di FaaS utilizzata. Per verificare la validità di questo algoritmo è stato realizzato un simulatore; una componente software in grado di eseguire questi algoritmi astraiendo dalle problematiche a basso livello, in un ambiente isolato e controllato. La strategia empirica è stata confrontata con delle strategie di baseline e con quella implementata sul sistema DFaaS, ovvero un'architettura decentralizzata, presente in letteratura, basata su FaaS, progettata per bilanciare autonomamente il traffico tra nodi edge federati. Ottenuti buoni risultati in termini di success rate, numero di rigetti e flessibilità di applicazione, la strategia empirica è stata ritenuta congrua all'obiettivo prefissato e implementata su DFaaS.



# Indice

<b>Acronimi</b>	<b>viii</b>
<b>1 Introduzione</b>	<b>1</b>
<b>2 Dal Cloud all'Edge: evoluzione del paradigma di computazione per nuovi scenari applicativi</b>	<b>4</b>
2.1 Dal Cloud al Serverless Computing . . . . .	4
2.1.1 Backend as a Service . . . . .	5
2.1.2 Function as a Service . . . . .	6
2.2 La computazione alla periferia della rete: Fog e Edge Computing . . . . .	10
2.2.1 Fog Computing . . . . .	12
2.2.2 Edge Computing . . . . .	13
2.3 Contestualizzazione del problema affrontato . . . . .	17
2.3.1 DFaaS . . . . .	17
2.3.2 Approccio sperimentale . . . . .	21
<b>3 FaaS all'Edge nella letteratura scientifica</b>	<b>23</b>
3.1 TinyFaaS: Una piattaforma FaaS leggera per ambienti Edge . . . . .	24
3.2 Un framework decentralizzato per il Serverless Edge Computing nell'IoT	27
3.3 RACER: Resource Allocation Control Engine with Reinforcement learning	31
<b>4 Architettura del Simulatore</b>	<b>34</b>
4.1 Architettura ad alto livello . . . . .	36
4.2 Instance Generator . . . . .	38

4.3	Simulator . . . . .	41
4.3.1	Reperimento dei dati necessari alla simulazione . . . . .	43
4.3.2	Caricamento dati da Database . . . . .	45
4.3.3	Esecuzione della simulazione . . . . .	46
4.3.4	Organizzazione dei dati di output . . . . .	49
4.4	Database Manager . . . . .	53
4.5	Analyzer . . . . .	57
4.6	Simulation Controller . . . . .	60
4.7	Configuration Manager . . . . .	64
<b>5</b>	<b>Algoritmi ed Esperimenti</b>	<b>65</b>
5.1	Descrizione del setting sperimentale . . . . .	66
5.1.1	Monitoring ed Exporters . . . . .	66
5.1.2	Setup sperimentale per la raccolta dati . . . . .	71
5.2	Empirical Strategy . . . . .	74
5.2.1	Descrizione Algoritmo . . . . .	76
5.2.2	Esperimento . . . . .	90
5.3	Base Strategy . . . . .	94
5.3.1	Descrizione Algoritmo . . . . .	94
5.3.2	Esperimento . . . . .	94
5.4	Random Strategy . . . . .	96
5.4.1	Descrizione Algoritmo . . . . .	96
5.4.2	Esperimento . . . . .	97
5.5	DFaaS Static Strategy . . . . .	99
5.5.1	Descrizione Algoritmo . . . . .	99
5.5.2	Esperimento . . . . .	101
5.6	Discussione e Confronto . . . . .	103
<b>6</b>	<b>Conclusioni</b>	<b>109</b>



# Acronimi

Di seguito vengono elencati gli acronimi utilizzati in questa trattazione:

**AFET** Average Function Execution Time

**API** Application Programming Interface

**AWS** Amazon Web Services

**BaaS** Backend as a Service

**BS** Base Station

**CLI** Command-Line Interface

**CoAP** Constrained Application Protocol

**CPU** Central processing Unit

**CSV** Comma-Separated Value

**DFaaS** Distribuited FaaS

**ER** Entità Relazioni

**FaaS** Function as a Service

**GoF** Gangs Of Four

**HPE** Helwett Packard Enterprise

**HTTP** HyperText Transfer Protocol

**IoT** Internet Of Things

**IT** Information Technology

**JSON** JavaScript Object Notation

**MAPE-K** Monitor-Analyze-Plan-Execute over a shared Knowledge

**ML** Machine Learning

**NIST** National Institute of Standards and Technology

**OCR** Optical Character Recognitio

**P2P** Peer-to-peer

**PNG** Portable Network Graphics

**QoS** Quality of Service

**QR-Code** Quick Response Code

**RACER** Resource Allocation Control Engine with Reinforcement learning

**RAM** Random Access Memory

**SDN** Software Defined Networking

**SLA** Service Level Agreement

**SQL** Structured Query Language

**UDP** User Datagram Protocol

**UEs** User Equipments

**UML** Unified Modeling Language

**VM** Virtual Machine

**YAML** YAML Ain't Markup Language



# Capitolo 1

## Introduzione

Negli ultimi anni l'utilizzo di dispositivi IoT (Internet Of Things) sta trovando una diffusione pervasiva in tutti i campi di applicazione delle tecnologie IT (Information Technology). Questo cambiamento sta intaccando tantissimi ambiti di lavoro, da quelli dell'automazione industriale fino a quello dell'agricoltura o della salute.

D'altro canto, la presenza di una rete di dispositivi pervasiva e onnipresente comporta anche un aumento esponenziale dei dati prodotti e trasferiti in rete, che richiedono di essere elaborati. Questi dati crescendo in varietà, volume e velocità spesso necessitano di essere pre-elaborati prima di essere trasferiti in cloud, per evitare un trasferimento massivo di informazioni. Inoltre, il modello del Cloud Computing, non soddisfa i requisiti in termini di latenza e approccio decisionale real-time. Questo significa che per applicazioni in cui il tempo di risposta deve essere breve l'utilizzo del modello centralizzato, tradizionale del Cloud Computing, non è una via percorribile; non è infatti possibile aspettare che l'informazione arrivi in cloud, venga processata e ritrasmessa sul dispositivo che ne ha richiesto l'elaborazione. In questo contesto sono nati i paradigmi di Fog e Edge Computing.

Questi modelli pongono l'enfasi sullo spostare la computazione in prossimità o addirittura sul dispositivo che genera i dati. L'elaborazione delle informazioni avviene quindi nella parte periferica della rete, vicino all'utente che ne vuole fruire.

Il modello del Cloud Computing ha consentito la nascita di altri modelli di servizio,

tra cui quello del Serverless Computing; ovvero un modello di sviluppo di applicazioni cloud-native, che consente agli sviluppatori di sviluppare applicazioni acquistando servizi di backend, e quindi non dovendosi preoccupare dell’infrastruttura sottostante che sarà gestita dal provider, su base “pay per use”.

Una declinazione del modello Serverless è quella del paradigma FaaS (Function as a Service), che descrive applicazioni in cui la logica di backend è scritta dagli sviluppatori sotto forma di funzioni molto limitate. Queste funzioni vengono poi affidate ad un provider di una piattaforma FaaS che si occuperà della loro esecuzione in container stateless.

FaaS è un modello di servizio che può essere applicato agevolmente nel contesto dell’Edge Computing, in quanto facilita il deployment di applicazioni realizzate secondo questa filosofia e ne garantisce una migliore reattività [1] [2] [3]. La natura granulare delle funzioni serverless consente di gestire in maniera più accurata problematiche relative al bilanciamento del carico e all’ottimizzazione nell’utilizzo delle risorse degli edge [1] [3].

Problema affrontato in questa trattazione è proprio quello legato al bilanciamento del carico in ambiente FaaS. A tal proposito è stato ideato un algoritmo distribuito di bilanciamento del traffico delle richieste che si basasse su dati empirici raccolti sul campo (**Empirical Strategy**); ovvero di metriche che sintetizzano lo stato dei nodi in cui è presente la piattaforma DFaaS [1] e delle funzioni su di essa in esecuzione. DFaaS [1] rappresenta una nuova architettura decentralizzata, presente in letteratura, basata su FaaS e progettata per bilanciare autonomamente il carico tra nodi edge federati. Questa architettura viene analizzata e dettagliata in Sezione 2.3.1.

Per poter validare la correttezza dell’algoritmo di distribuzione ideato è stato implementato un simulatore che ne consentisse l’esecuzione in un ambiente isolato e ne permettesse la comparazione con altri algoritmi di baseline e la strategia attualmente implementata in DFaaS.

L’intero codice sviluppato, riguardante il simulatore e tutte le strategie di bilanciamento del carico implementate, è reperibile in [4].

Al fine di realizzare una simulazione basata su dati sperimentali reali, sul sistema

DFaaS sono state raccolte le metriche necessarie ad eseguire la simulazione, realizzando una fase di raccolta dati molto accurata.

La strategia ideata ha portato ai risultati sperati, superando, in termini di performance, le strategie di baseline implementate, ed eguagliando quella attualmente implementata su DFaaS. L'algoritmo empirico ideato però supera di gran lunga, in termini di flessibilità, la strategia presente sul sistema. Di conseguenza, si ritiene l'obiettivo di questa tesi raggiunto.

Il lavoro di tesi qui presentato è strutturato nel seguente modo: nel Capitolo 2 viene presentato il background in cui esso si colloca, ponendo l'enfasi sul paradigma FaaS e l'Edge Computing; nel Capitolo 3 viene riportato quello che è lo stato dell'arte, descrivendo alcuni lavori correlati svolti da terzi e quindi già presenti in letteratura scientifica; nel Capitolo 4 viene descritta l'architettura del simulatore ideato per validare gli algoritmi di distribuzione del carico e compararne i risultati; il Capitolo 5 descrive il setting sperimentale con cui sono stati raccolti i dati, ed esegue una descrizione dettagliata degli algoritmi implementati, ponendo particolare enfasi sulla **strategia empirica**, concludendo poi con una comparazione dei risultati ottenuti in diversi scenari di test. Infine, nel Capitolo 6, vengono riportate le conclusioni, con riferimento a possibili sviluppi futuri.

# Capitolo 2

## Dal Cloud all'Edge: evoluzione del paradigma di computazione per nuovi scenari applicativi

### 2.1 Dal Cloud al Serverless Computing

Per quanto riguarda il **Cloud Computing** non esiste una definizione univoca, ma ne esistono una moltitudine in letteratura, ognuna delle quali pone l'enfasi su un aspetto diverso di questo paradigma. Il **Cloud** è un tipo di sistema distribuito costituito da computer interconnessi e virtualizzati forniti in modo dinamico e presentati come una (o più) risorse informatiche unificate sulla base di accordi sui livelli di servizio, che prendono il nome di **SLA** (Service Level Agreement), stabiliti attraverso la negoziazione tra fornitori di servizi e consumatori [5]. Gartner, tra le tante, definisce il Cloud Computing come uno stile di computazione in cui le capacità scalabili ed elastiche abilitate dall'IT (Information Technology) vengono fornite come servizio (as a service) ai consumatori esterni, utilizzando le tecnologie Internet [6]. Amazon sottolinea che il cloud computing consiste nella distribuzione on-demand delle risorse IT tramite Internet, con una tariffazione basata sul consumo. Diventa quindi possibile accedere a servizi di vario genere, come ad esempio capacità di calcolo, archiviazione, database,

ecc. sulla base delle proprie necessità, affidandosi ad un cloud provider, senza la necessità di acquistare e mantenere i propri data center [7]. In questo tipo di modello è quindi possibile monitorare quanto effettivamente consumato dal cliente, in termini di traffico e risorse, e fargli pagare solamente quanto utilizzato, con una spesa variabile.

Il modello del **Serverless Computing** nasce ed è abilitato grazie al cloud. Questo nuovo termine rappresenta un modello di sviluppo di applicazioni cloud native, che consente agli sviluppatori di creare ed eseguire applicazioni acquistando servizi backend su base flessibile “pay-as-you-go”. Gli sviluppatori pagheranno solo per i servizi che useranno, senza doversi preoccupare dell’infrastruttura sottostante, che sarà gestita e manutenuta totalmente a carico del provider [8] [9].

Anche per quanto riguarda questo modello non esiste una singola definizione, ma si declina in due principali applicazioni: **Backend as a Service (BaaS)** e **Function as a Service (FaaS)**.

### 2.1.1 Backend as a Service

La prima declinazione che è stata attribuita al termine Serverless è stata quella di **BaaS**, ovvero un paradigma usato per descrivere applicazioni che incorporano in maniera significativa, o totale, servizi cloud di terze parti per gestire lo stato o la logica di backend [10].

Da questa prima declinazione è possibile capire meglio da dove deriva il nome “Serverless”, letteralmente “senza server”; ovvero gli sviluppatori, o più in generale i fruitori di questo tipo di servizi, non hanno nessun server da gestire, ma le funzionalità utilizzate vengono tutte erogate on-demand tramite l’utilizzo di Internet. L’intera infrastruttura fisica del server o problemi di più basso livello come possono essere la gestione del numero di repliche di ogni servizio e il load balancing del traffico sono completamente a carico del fornitore del servizio (provider).

Il cliente di questo tipo di servizi si occupa solamente di gestire la logica principale dell’applicativo, tutto il resto viene realizzato tramite delle API (Application Programming Interface) di terze parti. Tra i tipi di servizi di terze parti che spesso vengono

utilizzati ci sono ad esempio: database, message broker, servizi di autenticazione, servizi di crittografia.

Seppur il cliente debba occuparsi solo di una quantità limitata di codice, in quanto tutto il backend è gestito da terze parti, a questo si contrappone la rigidità in fase di progettazione. L'utente, sfruttando questo tipo di servizi, è costretto ad utilizzare dei backend generici a cui non è possibile apportare modifiche al codice, personalizzandolo a piacere. I fornitori di questo tipo di servizi, come ad esempio Google Firebase [11], consentono all'utilizzatore di interfacciarsi al backend tramite delle API e di configurarlo in base alle proprie necessità, seppur in maniera limitata. Inoltre, sposando questo tipo di soluzioni, seppur si riducano notevolmente i tempi e i costi di sviluppo, sono soggette al fenomeno del *vendor lock-in* [12]. Acquistando questi prodotti come black-box, ogni provider realizzerà le soluzioni con i propri protocolli, con le proprie tecnologie e tutte le peculiarità del caso; tale situazione rende più difficile il passaggio da una soluzione di un fornitore ad una realizzata da un fornitore differente.

### 2.1.2 Function as a Service

Il paradigma **FaaS** rappresenta un'altra declinazione del modello Serverless. Questo paradigma descrive applicazioni in cui la logica server-side (backend) è scritta dagli sviluppatori sotto forma di funzioni, ognuna delle quali con uno scope molto limitato ed eseguita in container stateless [10]. Le applicazioni sono quindi pensate come costituite da numerose funzioni, ognuna delle quali non è altro che un elemento software che esegue una parte della logica di business [13].

A differenza del BaaS il cliente/sviluppatore scrive la logica di backend tramite funzioni molto limitate specificando a fronte di quali eventi queste dovranno essere invocate, offrendo quindi un maggior livello di controllo agli sviluppatori. Analogamente a BaaS, anche in questo caso tutto ciò che concerne l'infrastruttura o peculiarità di basso livello, sono gestite dal cloud provider [8]. Tra queste peculiarità, ad esempio, l'hardware relativo alle macchine fisiche, la collocazione geografica dei server, la gestione dei container utilizzati per offrire il servizio, gli orchestratori e le loro configurazioni,

e così via.

Una delle tecniche abilitanti di questo paradigma è la containerizzazione, come definito da RedHat, ovvero il raggruppamento di codice e di tutti i relativi componenti necessari, come librerie, framework e altre dipendenze, in modo che risultino isolate in un proprio contenitore che chiamiamo “**container**” [14]. Tra i software più conosciuti presenti in commercio che consentono di containerizzare applicazioni vale la pena citare Docker [15].

Le funzioni scritte dagli sviluppatori vengono eseguite dai provider in container che hanno le seguenti caratteristiche [8] [10]:

- **Stateless.** Lo stato, ovvero qualsiasi dato persistente, viene memorizzato esternamente al container.
- **Effimeri.** Possono essere eseguiti per un brevissimo tempo, paradossalmente anche per una sola invocazione della funzione, per poi essere distrutti.
- **Event triggered.** Non sono sempre in esecuzione, ma vengono eseguiti in risposta a determinati eventi che causano l’invocazione della funzione.
- **Gestiti da un provider di servizi cloud.** L’utente, in questo contesto, dovrebbe pagare solamente per quello che utilizza, non per mantenere dei server o applicazioni sempre attivi.

Tra le piattaforme più conosciute che implementano il paradigma FaaS sono presenti: AWS Lambda [16], Google Cloud Functions [17], OpenFaaS [18], Kubeless [19], Microsoft Azure Functions [20].

FaaS offre agli sviluppatori un’astrazione per eseguire le applicazioni in risposta a eventi, senza gestire i server. L’infrastruttura FaaS viene quindi utilizzata on demand, principalmente mediante un modello di esecuzione basato su eventi. Un provider cloud rende la funzione disponibile quando necessario, e gestisce l’allocazione delle risorse, senza tuttavia richiedere l’esecuzione costante dei processi in background. Essendo basate sugli eventi e non sulle risorse, le funzioni sono facilmente scalabili [13].

Queste funzioni avendo scope molto limitati, possono essere avviate in pochi milisecondi, per tale ragione si prestano molto bene ad essere scalate dinamicamente a fronte di picchi di traffico. A fronte di aumenti delle richieste in ingresso il cloud provider provvederà ad aumentare il numero delle repliche della funzione congestionata (**scale up**), mentre al calare della domanda eliminerà le copie non più necessarie (**scale down**). Il vantaggio introdotto dall'esecuzione di funzioni stateless e dalla loro intrinseca propensione alla scalabilità orizzontale, si rispecchia anche in un vantaggio economico per l'utente/sviluppatore, in quanto il costo addebitato concerne solamente le risorse utilizzate, non tenendo conto dei tempi morti [13].

Come detto poc'anzi, l'utilizzatore di piattaforme di FaaS deve specificare a fronte di quali eventi devono essere eseguite le funzioni, e quindi creati i container che le eseguiranno. Nello schema in Figura 2.1 viene riportato il ciclo di vita di una funzione.

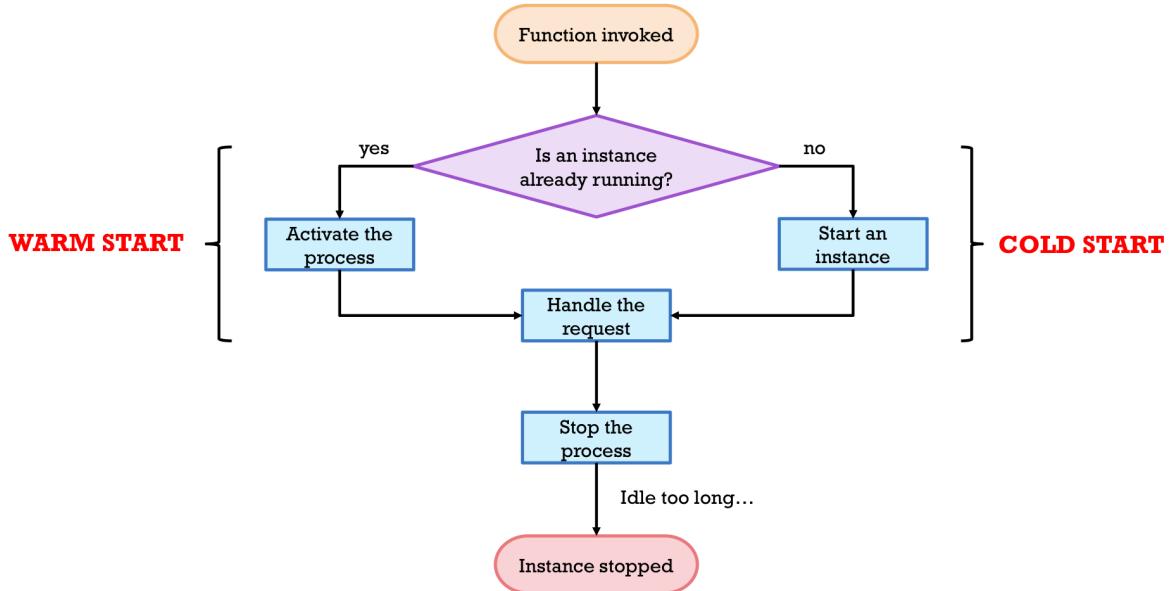


Figura 2.1: Ciclo di vita di una funzione.

Quando una funzione viene invocata si possono verificare due possibili situazioni, ovvero che un'istanza di quella funzione sia già in esecuzione (**warm start**), oppure che debba essere creata (**cold start**) [10]. Avverrà un **warm start** se al momento dell'invocazione esiste già un'istanza della funzione ed il container che la ospita, frutto di un'invocazione precedente, mentre avverrà una **cold start** nel caso in cui questa non

esista e, quindi, debba essere creata l’istanza del container e fatto partire il processo della funzione. Non sorprende che, quando si considera la latenza di avvio, siano queste **cold start** a destare maggiori preoccupazioni. Difatti queste latenze dipendono da tantissimi fattori differenti, come ad esempio il linguaggio utilizzato, il numero di librerie impiegate, quanti dati è necessario caricare all’avvio, ecc. La **cold start** rappresenta quindi un problema ricorrente nelle piattaforme FaaS. Successivamente, una volta che è attivo il container con il processo della funzione, la richiesta viene gestita, il processo della funzione stoppato e messo in uno stato di **idle**. Dopo un certo intervallo di tempo per cui questa funzione non verrà più richiamata, l’istanza del container verrà stoppata [10].

Riguardo a quanto trattato fino a questo momento, è possibile identificare una serie di vantaggi derivanti da questo modello, tra cui la rapidità nei tempi di sviluppo delle applicazioni e il conseguente aumento di produttività (è possibile scrivere le funzioni con un qualsiasi linguaggio di programmazione), la riduzione dei costi operazionali (vengono utilizzate soluzioni di outsourcing, delegando la gestione al cloud provider), riduzione dei costi di sviluppo e di esecuzione (deployment delle funzioni costituenti l’applicativo facilitato e gestito dal cloud provider) ed infine la riduzione dell’inefficienza e dei costi sostenuti dal cliente (tariffazione variabile) [10] [13].

Seppur i vantaggi siano tanti, FaaS è ancora un paradigma poco maturo che presenta anche una serie di svantaggi rilevanti di cui tenere conto in applicazioni che ne abbracciano la filosofia. Tra gli svantaggi più rilevanti: il cliente si affida a piattaforme gestite dal cloud provider (downtime inaspettati, cambiamento dei costi, forzatura nell’aggiornamento delle API, ecc.), problemi legati alla sicurezza e alle performance in architetture multi-tenant (ovvero dove vengono condivise risorse tra organizzazioni diverse) ed infine problemi legati al testing, debugging e monitoring delle applicazioni sviluppate, ovvero i classici problemi legati alla computazione distribuita.

Il modello Serverless può quindi essere utilizzato sia da applicazioni tradizionali che realizzate a microservizi e ciò che consente agli sviluppatori in questo ambito di ottenere i maggiori vantaggi è la combinazione dell’utilizzo di servizi FaaS e BaaS [13]. In questo modo è possibile superare le tradizionali architetture 3-tier (**client-server**-

**database**) e spostare l'attenzione su un'architettura Serverless, come quella mostrata in Figura 2.2

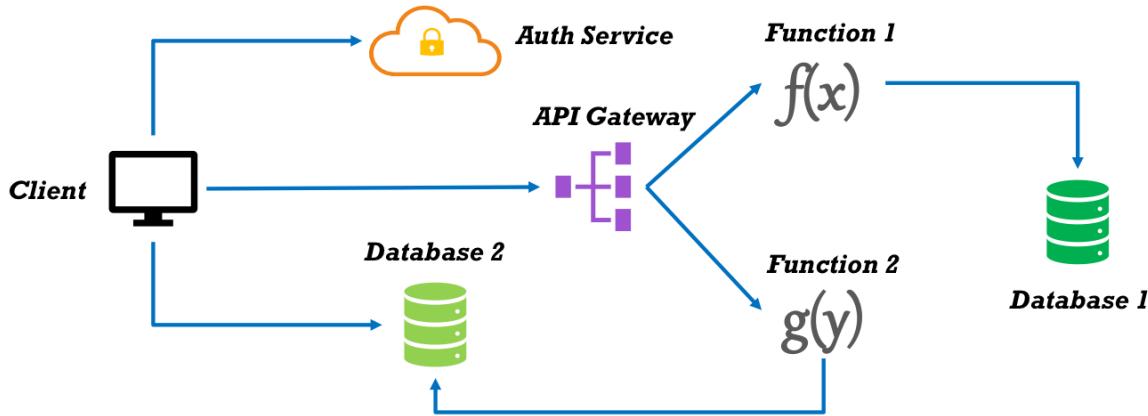


Figura 2.2: Esempio di architettura Serverless.

Come mostrato in Figura 2.2 è possibile isolare la logica di frontend sul client, rendendolo molto più snello e reattivo, realizzare parte della logica di backend tramite delle funzioni affidate al cloud provider (FaaS) e utilizzare servizi gestiti totalmente da terze parti, quali servizi di autenticazione, di messaggistica e database (BaaS).

All'interno di questo tipo di architettura Serverless la logica del sistema è distribuita tra le varie componenti, ognuna delle quali ha un compito specifico e gioca un ruolo fondamentale nel formare poi il sistema nella sua interezza. Non è presente quindi un arbitro centrale, un componente che orchestra tutte le altre, ma bensì queste sono distinte, si coordinano e comunicano con un'architettura basata su eventi (**choreography over orchestration**).

## 2.2 La computazione alla periferia della rete: Fog e Edge Computing

Negli ultimi anni i dispositivi **IoT** (Internet Of Things) stanno trovando una diffusione pervasiva in tutti i campi di applicazione delle tecnologie IT. Tale diffusione è evidente in settori come quello dell'automazione dei processi industriali (macchine intelligenti, dotate di sensori e attuatori che raccolgono dati sullo stato della macchina),

nell’industria automobilistica (veicoli intelligenti a guida autonoma in grado di prendere decisioni), nel campo della salute (monitoraggio dello stato di salute dei pazienti), nel monitoraggio ambientale (rilevamenti dello stato del terreno per la previsione di terremoti, frane, eruzioni, ecc.), nella sicurezza (telecamere e sistemi di allarme intelligenti), e tanti altri.

La sempre maggior diffusione di questi dispositivi porta ad un aumento critico non solo del volume dei dati che circolano in rete ma anche della velocità con cui essi vengono prodotti. Oltre tutto, con l’avvento di nuove tecnologie come il 5G questo fenomeno tende ad aumentare repentinamente in quanto aumenta la copertura ed il numero di dispositivi che è possibile connettere [21].

Il vero valore introdotto da questi dispositivi IoT non è tanto legato all’acquisizione di dati in sé, ma agli insight che da essi si possono trarre, ovvero da come questi possono essere utilizzati per fare previsioni e prendere decisioni, portando ad un aumento dei guadagni o un risparmio di tempo in vari campi di business [22].

Per estrarre valore dai dati è necessario che avvenga una computazione, e questa deve essere posizionata in qualche punto della rete. Per molte applicazioni questa elaborazione avviene in **cloud**, ovvero in data center centralizzati posti nel core della rete. Per far sì che però il concetto di elaborazione **real-time** sia valido, questa computazione deve avvenire in maniera molto rapida, e la latenza di trasmissione dei dati avanti e indietro nella rete deve essere limitata. Visto l’aumento in termini di velocità, varietà e volume dei dati che vengono prodotti deve essere rivisto il modello classico di **cloud centralizzato** fino ad ora utilizzato. Sembra quindi necessario il passaggio ad un modello di **cloud decentralizzato**, che sposti parte della computazione alla periferia della rete; è in questo contesto che nascono i termini di **Fog** e **Edge Computing**, due concetti che vengono spesso intercambiati e con definizioni simili, ma il cui obiettivo comune è quello dell’ottimizzazione delle performance, della minimizzazione dei tempi di risposta percepiti dall’utente e della diminuzione del quantitativo di dati trasferiti in rete.

Le tradizionali architetture di **Cloud Computing** non soddisfano i requisiti in termini di latenza e approccio decisionale real-time [22]. L’analisi dei dati in prossi-

mità di dove essi vengono prodotti può portare a tantissimi vantaggi in particolare per applicazioni che richiedono di prendere decisioni critiche in maniera tempestiva (**mission-critical**). Si consideri ad esempio un auto a guida autonoma che deve prendere una decisione relativa ad un sorpasso o ad una frenata. Questo tipo di decisioni sono critiche e devono essere prese in tempi estremamente ridotti in quanto potrebbero causare incidenti, coinvolgendo altre vetture e persone. In contesti di questo tipo non è quindi possibile pensare ad un trasferimento dei dati in un modello cloud centralizzato per poi attendere la risposta dell'elaborazione, in quanto la latenza della comunicazione di rete potrebbe causare danni irreparabili [22].

A questo tipo di problematiche ne vengono sommate altre legate alla privacy; tali normative spesso vietano l'archiviazione dei dati al di fuori della sede in cui essi vengono prodotti. Tutto ciò porta quindi a confermare che il luogo migliore in cui processare i dati è vicino a dove essi vengono prodotti, ovvero nella periferia della rete [22].

### 2.2.1 Fog Computing

Come definito da Cisco, coniatore di questo termine nel 2014: “Per fog computing si intende la decentralizzazione di un’infrastruttura di elaborazione tramite ampliamento del cloud, attraverso il posizionamento strategico di nodi tra il cloud e i dispositivi terminali” [23].

Il **Fog Computing** rappresenta quindi un'estensione del modello cloud, in cui i dati generati dai vari dispositivi non vengono caricati direttamente in cloud, ma vengono prima pre-elaborati in mini data center decentralizzati.

In tal senso i dati, l'elaborazione, la memorizzazione e le applicazioni vengono posizionate più vicine all'utente o al dispositivo IoT, dove devono essere elaborati, creando uno strato intermedio di “nebbia” (“**fog**” in inglese), all'esterno del modello centralizzato classico del cloud.

I dispositivi che compongono questo strato intermedio di pre-elaborazione vengono chiamati **Fog node** e possono essere distribuiti ovunque con una connessione di rete: in una fabbrica, in cima a un palo di alimentazione, lungo un binario ferroviario, in

un veicolo o su una piattaforma petrolifera. Qualsiasi dispositivo con elaborazione, archiviazione e connettività di rete può essere un fog node [24].

L'utilizzo di questo tipo di computazione locale, più vicina al luogo in cui i dati vengono prodotti, consente di limitare fortemente il traffico sulla rete. L'enorme quantità di informazione che viene generata ad esempio dai dispositivi IoT può essere filtrata da questo strato intermedio per poi memorizzare sui data center remoti solo l'informazione finale [24]. Questo, naturalmente, favorisce anche i tempi di risposta verso l'utente finale, diminuendo la latenza di risposta delle elaborazioni.

In Figura 2.3 è possibile notare la presenza di questi nodi intermedi di “**fog**” (gateway o hub di vario genere), che si interpongono tra gli edge ed il cloud.

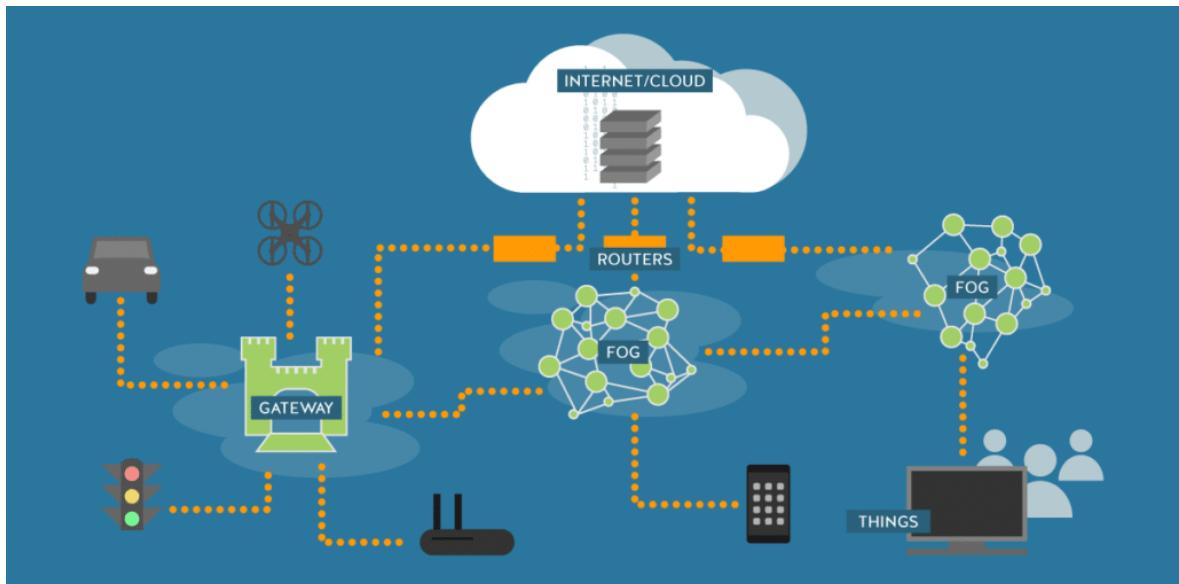


Figura 2.3: Esempio di architettura Fog Computing.

## 2.2.2 Edge Computing

HPE (Hewlett Packard Enterprise) definisce l'**Edge Computing** come una forma di elaborazione eseguita in sede o in prossimità di una particolare origine dati, riducendo al minimo la necessità di elaborare i dati in un data center remoto [25].

Spesso i termini di Fog e Edge Computing vengono intercambiati, questo è dovuto al fatto che la definizione di Fog, essendo più generale, comprende anche quella di

Edge Computing. Difatti, quest'ultimo concetto specializza ulteriormente il primo, mantenendo però i medesimi obiettivi, ovvero la riduzione dei tempi di risposta, la minimizzazione dei ritardi percepiti dalle applicazioni e la diminuzione del traffico di informazioni inviato in rete. Con Edge Computing si intende quindi la collocazione dell'intelligenza, della potenza di elaborazione e della capacità di comunicazione nella posizione più vicina ai dati di un sistema o al suo utente finale [22] [26]. Ad esempio, nel caso di un'azienda di spedizioni portuali, il nodo di edge potrebbe essere collocato in una delle banchine del porto in cui la merce viene caricata e scaricata; in questo modo vengono raccolti ed elaborati dati relativi alla merce [26]. Il nodo di edge è quindi collocato in prossimità dei dati. La declinazione di questo termine può essere ancora più estremizzata, intendendo la collocazione della computazione direttamente sul dispositivo stesso (*on-device*) che produce i dati, ovvero nella parte più periferica possibile della rete [22]. Un esempio di questo tipo di applicazione potrebbe essere in un contesto IoT, in cui si hanno sensori intelligenti e già dotati di capacità computazionale, che oltre a raccogliere il dato ne eseguono una prima elaborazione. Il concetto di Edge Computing è quindi soggetto ad interpretazione, la cui declinazione è spesso derivante dal tipo applicazione che si vuole realizzare e alla rete di dispositivi che si ha sul campo.

Seppur l'Edge Computing sia un paradigma nuovo e ben lontano ancora da quella che è la realizzazione del suo massimo potenziale, gran parte delle tecnologie o piattaforme che vengono utilizzate tutti i giorni comprendono qualche forma di Edge Computing [25]. Si possono citare vari esempi, tra cui: reti di distribuzione dei contenuti (CDN - Content Delivery Network), videogiochi, sistemi di manutenzione predittiva, fino ad arrivare alle piattaforme di streaming di musica e video che, ad esempio, memorizzano nella cache le informazioni per ridurre la latenza di risposta e il traffico di rete [25].

L'Edge Computing trova applicazione in tantissimi campi, nel settore industriale infatti consente alle aziende di monitorare lo stato dei macchinari delle catene di montaggio o addirittura di avere risposte da sistemi di anomaly detection in tempo reale, facendo previsioni e cercando di rilevare guasti prima che questi si verifichino. Nel settore sanitario, invece, è possibile proporre cure personalizzate ai pazienti, fornendo ai medici uno strumento per ottenere real time maggiori informazioni sulle persone, senza

doversi appoggiare a database di terze parti. Infine, anche in contesti di domotica e case intelligenti le soluzioni di edge computing trovano grande diffusione. Dispositivi di assistenza vocale come Amazon Alexa o Google Home senza l’elaborazione decentralizzata non sarebbero in grado di fornire prontamente risposte all’utente, danneggiando di gran lunga l’usabilità e la user experience [25].

Per dare un’idea di quelle che sono le potenzialità di questo paradigma, Gartner stima che entro il 2025 il 75% dei dati verrà elaborato al di fuori del tradizionale data center o del cloud [27].

L’Edge Computing può anche contribuire allo sviluppo di applicazioni data intensive, ovvero applicazioni intelligenti che elaborano grandi quantità di dati. Attività di machine learning, come algoritmi predittivi e riconoscimento di immagini possono essere fatte direttamente in loco, sugli edge, eliminando la necessità di trasferire grandi quantità di dati [28]. Tutte queste possibilità aprono la strada ad un tema di ricerca molto interessante come quello dell’**Edge Machine Learning** [29] [30].

Facendo riferimento a quanto esposto fino a questo momento tra i vantaggi introdotti dall’Edge Computing vengono riconosciuti: riduzione della latenza e maggiore velocità (tempi di risposta migliorati in quanto la computazione è posta vicino all’utente), maggior sicurezza (l’informazione viene elaborata localmente, all’interno della rete aziendale o in un sistema chiuso), riduzione dei costi (viene ottimizzato il flusso dei dati verso i sistemi centrali, mantenendo gran parte dei dati grezzi sugli edge), maggior privacy e affidabilità (elaborando dati localmente gli edge non soffrono di problemi di connettività con i data center remoti, oltretutto i dati possono essere aggregati e resi anonimi prima del trasferimento) e rapidità nella scalabilità (applicazioni in esecuzione sugli edge possono essere scalate facilmente e le aziende possono ampliare agevolmente le reti di dispositivi) [23].

In Figura 2.4 è possibile vedere come le tecnologie e i modelli fino ad ora esposti possano coesistere in un contesto ibrido e completo.

In un’architettura come quella mostrata in Figura 2.4 è possibile vedere come ci sia un’organizzazione piramidale. In cima alla piramide sono presenti i Cloud Data Center che sono nell’ordine dei migliaia e rappresentano i data center remoti con potentissime

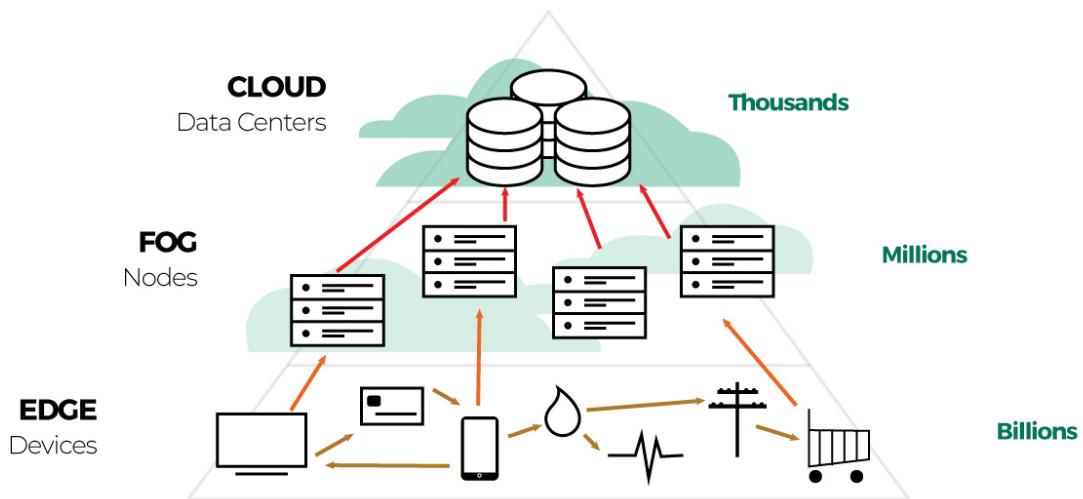


Figura 2.4: Esempio di architettura di Cloud, Fog e Edge Computing.

risorse di calcolo. Scendendo è possibile trovare il fog layer, in cui i gateway, hub, o più in generale dispositivi di aggregazione aumentano in numero, ma hanno a disposizione meno risorse computazionali. Infine, alla base sono presenti i dispositivi di edge, che sono di vario genere e natura, con risorse computazionali a disposizione molto differenti (possono andare da smart watch, smart phone e dispositivi medici, fino a computer molto potenti), nell'ordine dei miliardi.

Un'applicazione data intensive potrebbe essere suddivisa in varie fasi di elaborazione in un'architettura come quella mostrata poc'anzi. Ad esempio gli edge della rete entrerebbero in gioco per la raccolta e acquisizione dati, eseguendo una prima elaborazione prima di trasportarli in uno strato intermedio. Questi possono subire ulteriori elaborazioni o fasi di ingegnerizzazione, per poi essere trasportati in cloud, ovvero nei data center centralizzati, per essere memorizzati. Una volta che sono qui possono essere utilizzati per addestrare modelli di machine learning, che a loro volta possono essere messi in esecuzione sui dispositivi all'edge della rete per fare inferenze [28].

In poche parole, l'Edge Computing indica un'elaborazione dei dati che avviene ai margini della rete, in prossimità della posizione fisica dove essi vengono generati, mentre il Fog Computing funge da mediatore tra l'edge e il cloud per vari scopi, come ad esempio il filtraggio dei dati. Il Fog Computing non può sostituire l'Edge Computing, mentre quest'ultimo può vivere senza il primo in molte applicazioni [31].

## 2.3 Contestualizzazione del problema affrontato

FaaS è un modello di servizio che può essere applicato agevolmente nell'Edge Computing in quanto facilita il deployment di applicazioni realizzate secondo tale filosofia e ne garantisce una miglior reattività, migliorandone i tempi di risposta [1] [2] [3]. Inoltre, le funzioni possono essere condivise tra più applicazioni e la loro natura granulare consente una gestione più fine ed accurata degli SLA, del bilanciamento del carico e dell'ottimizzazione nell'utilizzo delle risorse nei nodi edge della rete [1] [3].

Sfortunatamente, il modello FaaS convenzionale non è però applicabile in un ambiente intrinsecamente distribuito ed eterogeneo come quello dell'edge computing, per via del traffico fortemente dinamico, della topologia di rete variabile e delle risorse limitate presenti nei nodi perimetrali della rete [1] [2]. Inoltre, in contesti come quelli delle applicazioni mobile o di auto a guida autonoma, i punti di accesso alla rete potrebbero cambiare continuamente.

Per poter mantenere i vantaggi di FaaS in un ambiente dinamico come quello edge è quindi necessario federare questi nodi in ambienti di esecuzione distribuiti per bilanciare il carico tra di essi. In [1] viene presentata un'architettura federata e decentralizzata basata su FaaS per bilanciare autonomamente il carico del traffico tra gli edge della rete appartenenti a ecosistemi FaaS edge federati; tale sistema prende il nome di **DFaaS** (Distributed FaaS) [1].

Sfruttando quindi l'architettura di DFaaS e il prototipo messo a disposizione, l'idea alla base di questo lavoro di tesi è la realizzazione ed il confronto di algoritmi per il bilanciamento del carico di richieste FaaS in un ambiente decentralizzato, con topologia di rete fortemente variabile, come quello edge.

### 2.3.1 DFaaS

DFaaS, presentata da Ciavotta et al [1], rappresenta una nuova architettura decentralizzata basata su FaaS, progettata per bilanciare autonomamente il carico di traffico tra nodi edge federati. DFaaS si basa su una rete peer-to-peer overlay per condividere informazioni sulla topologia della rete, la latenza e gli stati di carico dei nodi. Que-

ste informazioni vengono sfruttate dai nodi sovraccarichi per reindirizzare parte delle richieste di esecuzione FaaS in ingresso ai peer non in stato di sovraccarico.

## Scenario di riferimento

In Figura 2.5 viene mostrato lo scenario di riferimento di DFaaS, ovvero un insieme di nodi FaaS perimetrali della rete, geograficamente distribuiti.

Su ognuno di questi nodi è presente una piattaforma di FaaS per l'esecuzione di funzioni Serverless; nell'implementazione di DFaaS viene utilizzata OpenFaaS. Ogni nodo è collegato ad un access point per la connessione alla rete.

Ogni nodo edge può ricevere richieste di esecuzione di funzioni (ad esempio sotto forma di richieste HTTP) generate dagli utenti che sono connessi agli access point. Solitamente un nodo edge o un sotto insieme di essi sono gestiti da un edge provider, che può ad esempio essere un operatore di rete; nodi perimetrali differenti possono essere di proprietà dello stesso provider, o di provider differenti, e sono interconnessi da connessioni di rete, solitamente sottoposte a SLA.

In questo scenario viene considerato un ecosistema di edge computing federato, in cui edge provider differenti, ognuno dei quali gestisce la propria piattaforma DFaaS con le proprie tecnologie, si uniscono alla federazione e possono fare affidamento in maniera totalmente trasparente sulle risorse computazionali offerte dai nodi edge di altri provider. Questo avviene ad esempio quando i nodi vengono sovraccaricati a causa di picchi di traffico.

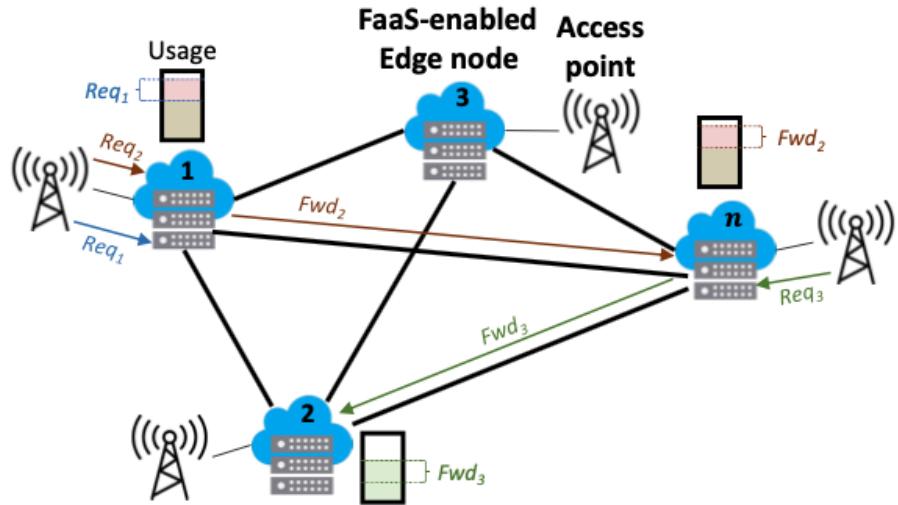


Figura 2.5: Scenario di riferimento DFaaS [1].

### Architettura ad alto livello

L’architettura ad alto livello di DFaaS [1] è quella mostrata in Figura 2.6, in particolare, per motivi di modularità, manutenibilità ed evolvibilità, DFaaS è stato realizzato servendosi di componenti preesistenti, testati e mantenuti attivamente (tra cui il Proxy e la piattaforma di FaaS), implementando quelle che sono le logiche di controllo distribuite all’interno dell’agente.

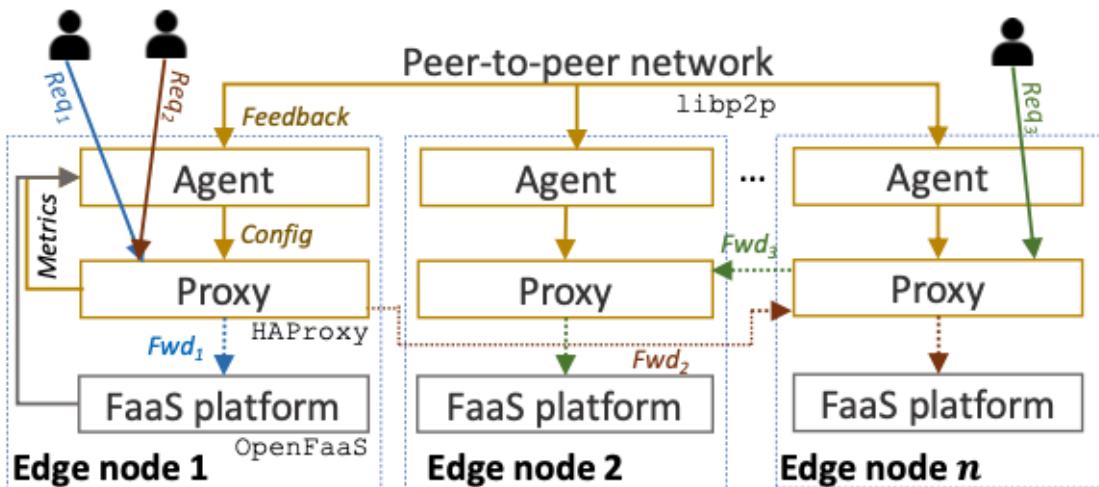


Figura 2.6: Architettura di DFaaS [1].

Le principali componenti che costituiscono l'architettura di DFaaS sono:

- **Agenti.** Gli agenti sono identici, indipendenti e in esecuzione sui singoli nodi di edge. Questi agenti comunicano attraverso una rete P2P (peer-to-peer). Ogni agente è responsabile di gestire la connessione alla rete P2P e tramite essa la comunicazione con gli altri agenti. Periodicamente deve monitorare lo stato della piattaforma di FaaS presente nel nodo su cui è in esecuzione, attraverso la raccolta di alcune metriche. Inoltre, sfruttando quanto raccolto, deve calcolare la propria capacità residua in termini di richieste al secondo per ogni classe di funzione. Deve inoltre gestire lo scambio di messaggi con tutti gli altri agenti e, tramite i dati ricevuti, calcolare i pesi di distribuzione delle richieste verso gli altri nodi. Infine, si occupa di riscrivere le configurazioni del proxy presente sul nodo con i pesi appena calcolati.
- **Proxy.** Il Proxy, in esecuzione su ogni nodo, si occupa dell'effettiva ricezione e distribuzione delle richieste HTTP (unico protocollo supportato da DFaaS). Il proxy per ogni richiesta decide se inoltrarla alla piattaforma FaaS locale o a quella in esecuzione su uno dei peer di cui l'agente è a conoscenza, inoltrando la richiesta al proxy presente su quel nodo. Come proxy è stato utilizzato HAProxy [32].
- **Piattaforma FaaS.** Una volta che una richiesta viene inoltrata alla piattaforma FaaS allora questa viene gestita dalla classe di funzione di interesse. Ogni piattaforma FaaS incorpora una serie di servizi in grado di istanziare, eseguire e gestire queste funzioni. La piattaforma FaaS può essere implementata localmente, ovvero sullo stesso nodo dell'agente e del proxy, o eseguita esternamente su uno o più nodi (qualora venga gestito un cluster FaaS). In DFaaS come piattaforma è stata utilizzata OpenFaaS [18], ma ne potrebbero essere adottate differenti.

Per distribuire autonomamente il traffico, è stato implementato un semplice algoritmo distribuito, che verrà descritto nella Sezione 5.5, a cui è stato attribuito, in

questo lavoro di tesi, il nome di **DFaaS Static Strategy**. Tale algoritmo viene eseguito in maniera sincrona da ogni agente ad intervalli di tempo regolari, e si occupa di calcolare le percentuali di richieste da inviare verso gli altri peer della rete per ogni classe di funzione sovraccaricata. Per effettuare questo calcolo tale algoritmo si basa semplicemente sul numero di richieste che ogni altro nodo può gestire, che gli sono state precedentemente comunicate.

### 2.3.2 Approccio sperimentale

L'idea alla base di questo lavoro di tesi è quella di implementare e comparare una serie di algoritmi di distribuzione del carico in un ambiente FaaS distribuito, sul prototipo del sistema appena descritto.

Vista però la complessità di DFaaS, implementare direttamente gli algoritmi sul prototipo non è sicuramente l'approccio sperimentale più adatto. Da qui è nata quindi la necessità di creare un ambiente isolato in cui poter effettuare valutazioni a priori sulla qualità e correttezza degli algoritmi, prima di implementarli direttamente sul sistema in essere.

L'approccio sperimentale scelto è stato quindi quello di ideare un algoritmo distribuito per il bilanciamento del carico in ambiente FaaS (**Empirical Strategy**, descritta nella Sezione 5.2), in un contesto fortemente dinamico come quello dell'edge computing, e di compararlo con alcuni algoritmi/strategie<sup>1</sup> di baseline (**Base** e **Random Strategy**, descritte in Sezione 5.3 e Sezione 5.4) e con la strategia statica implementata di base nel prototipo di DFaaS (**DFaaS Static Strategy**, descritta nella Sezione 5.5).

Successivamente, è stato realizzato un ambiente di simulazione, la cui architettura viene descritta nel Capitolo 4, per testare la validità di questa strategia empirica prima di riportarla effettivamente sul sistema.

Per poter simulare il funzionamento della strategia empirica e poterla confrontare efficacemente con gli altri algoritmi, è necessario avere a disposizione una buona

---

<sup>1</sup>I termini algoritmi e strategie potrebbero essere spesso intercambiati, ma entrambi indicano gli algoritmi di distribuzione del carico implementati. Strategie è un termine più legato all'aspetto implementativo.

quantità di dati raccolti sperimentalmente, che permettano di replicare il comportamento del sistema in un ambiente separato come quello di simulazione. Sulla base di tale necessità è quindi seguita un'attenta fase di raccolta dati sul prototipo di DFaaS (descritta nel Capitolo 5).

Sulla base del simulatore realizzato e dei dati sperimentalmente raccolti è stato possibile validare il funzionamento della strategia empirica ideata. Inoltre, è stato possibile comparare il funzionamento ed i risultati con le altre strategie implementate (verranno mostrati in Capitolo 5), sfruttando situazioni realistiche raccolte sul campo.

Seguendo questo approccio è stato possibile dimostrare la correttezza dell'algoritmo pensato, e solo dopo averne verificato sperimentalmente la validità è stato implementato sul prototipo di DFaaS.

# Capitolo 3

## FaaS all’Edge nella letteratura scientifica

In questo capitolo vengono riportati alcuni studi presenti nella letteratura scientifica relativi all’applicazione del paradigma FaaS in un ambiente di Edge Computing, inerenti o con obiettivi simili a quelli di questa tesi.

FaaS è un modello di servizio che si presta bene ad essere applicato all’Edge Computing [1] [2] [3]. Difatti, rappresenta un modello di programmazione flessibile e basato su eventi, idoneo ad essere integrato nell’ambito dell’IoT ove, a fronte della produzione di dati, serve eseguire un processamento di essi [2] [33].

Le piattaforme FaaS attualmente presenti in commercio forniscono già questi vantaggi per il mondo IoT ma vengono sfruttate in maniera inefficiente. Difatti, l’invio di tutti gli eventi e i dati al cloud per l’elaborazione, comporta un carico elevato sulla rete e un’elevata latenza di risposta [33]. Una piattaforma di FaaS leggera e ideata ex-novo per sfruttare le caratteristiche dell’ambiente Edge è quella presentata in Sezione 3.1.

In un contesto inerentemente distribuito e decentralizzato come quello dell’Edge viene facilitato quello che è il deployment delle applicazioni realizzate mediante il paradigma FaaS; ovvero la cui logica fondamentale è scritta per via di funzioni estremamente granulari. Inoltre, collocando la computazione vicino all’utente, viene garantita una miglior reattività e diminuita la latenza nei tempi di risposta.

Altro vantaggio importante derivante dalla granularità di queste funzioni FaaS è che esse possono essere riutilizzate tra diversi applicativi [1] [2]. Inoltre, per via del fatto che i nodi edge hanno un quantitativo di risorse limitato rispetto a quello dei tradizionali data center, è necessario affrontare il tema della bilanciamento del carico e allocazione delle risorse in questo tipo di ambienti [1] [2] [3]. I lavori presentati in Sezione 3.2 e in Sezione 3.3 mirano proprio a cercare di risolvere il problema della distribuzione del carico in ambiente edge, ovvero il medesimo obiettivo perseguito da questa tesi, il cui principale algoritmo ideato viene presentato in Sezione 5.2.

### 3.1 TinyFaaS: Una piattaforma FaaS leggera per ambienti Edge

Seppur sia il modello di computazione Serverless che quello dell’Edge Computing siano ancora agli arbori e rappresentino un trend di ricerca estremamente attuale e all'avanguardia, i primi lavori per cercare di conciliare questi due paradigmi sono già stati svolti.

In un ambiente estremamente eterogeneo e dinamico come quello dell’edge, si possono trovare dispositivi di vario genere, che spaziano da semplici sensori con poca capacità computazionale e di archiviazione, fino a raggiungere computer o nodi con a disposizione un maggior quantitativo di risorse. Per cercare di portare il paradigma FaaS in questo ambiente eterogeneo, sono stati svolti alcuni lavori atti a realizzare delle piattaforme di FaaS leggere e che potessero funzionare anche su dispositivi con poca disponibilità di risorse. Il lavoro proposto in [33] si muove proprio in questa direzione.

Pfandzelter e Bermbach [33], difatti, hanno proposto un’architettura FaaS progettata per un ambiente Edge. La piattaforma qui presentata è infatti estremamente leggera, in grado di essere eseguita su un nodo di Edge con risorse limitate.

Gli autori, dopo un’interessante analisi dei requisiti del sistema, hanno presentato l’architettura di **tinyFaaS**, ovvero di una piattaforma di FaaS realizzata ex-novo, per l’ambiente Edge. Le componenti fondamentali costituenti l’architettura di **tinyFaaS**

sono le seguenti:

- **Reverse Proxy.** Questo componente riceve le richieste Constrained Application Protocol (CoAP) [34] e le inoltra ai componenti che gestiscono le funzioni (chiamati *handler*). Per ogni funzione viene registrata, all'interno del reverse proxy, una risorsa CoAP (endpoint), in modo tale che le richieste a questa risorsa verranno trattate come chiamate alla funzione associata. Quando una richiesta arriva all'endpoint CoAP, il reverse proxy seleziona uno degli handler per processare tale richiesta. Una volta ricevuta, il reverse proxy manda una richiesta HTTP (Hypertext Transfer Protocol), al proxy HTTP presente nell'handler selezionato per quella funzione. E' stato scelto CoAP come protocollo di messaggistica tra i dispositivi IoT e tinyFaaS per evitare l'utilizzo di un message-broker; inoltre, CoAP è molto leggero ed efficiente dal momento che, a livello di trasporto, è basato su UDP (User Datagram Protocol).
- **Function Handlers.** Costituiscono la parte fondamentale di **tinyFaaS**, ognuno di essi viene eseguito in un container Docker separato. Ognuno di questi container contiene sia il runtime per il linguaggio di programmazione con cui è stata scritta la funzione, sia del codice standard per facilitare la chiamata da parte del reverse proxy. Come è possibile vedere in Figura 3.1, il container include anche un server HTTP che accetta le richieste provenienti dal reverse proxy, le esegue utilizzando i dati ricevuti all'interno della richiesta e ritorna il risultato. Mentre il reverse proxy utilizza CoAP per la comunicazione verso l'esterno, HTTP è invece utilizzato per la comunicazione interna tra le componenti di **tinyFaaS**. Ogni handler di funzione può accettare un numero arbitrario di richieste. Di conseguenza, le richieste all'interno di un container sono isolate l'una dall'altra solo a livello di applicazione utilizzando thread. Sebbene i container Docker riducano leggermente le prestazioni, semplificano notevolmente il deployment delle funzioni.
- **Management Service.** Questa componente è responsabile della creazione di nuove funzioni all'interno della piattaforma. Gli sviluppatori che vogliono mette-

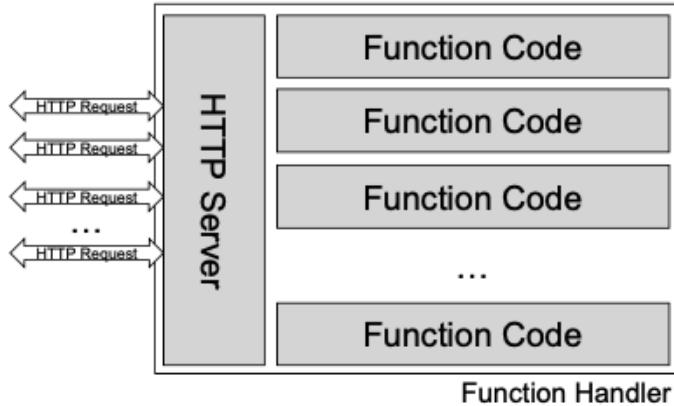


Figura 3.1: Componente *Function Handler* in **tinyFaaS** [33].

re in esecuzione sulla piattaforma una nuova funzione la invieranno, tramite una richiesta HTTP, all’endpoint del management service. Successivamente, questo servizio crea quelli che sono i container dedicati a contenere gli *handler* per questa funzione e registra la corrispondente risorsa CoAP (endpoint) nel reverse proxy. Infine, connette quelli che sono i container che ospitano gli *handler* con il reverse proxy, attraverso una rete virtuale Docker, in modo tale che essi possano liberamente comunicare. Allo stesso modo il management service esporrà degli endpoint per aggiornare o eliminare le funzioni. Così facendo le funzioni in esecuzione su **tinyFaaS** possono essere riconfigurate a runtime. Inoltre, avere un unico punto di ingresso consente anche la multi-tenancy in quanto il servizio di gestione può occuparsi dell’autenticazione dell’utente durante la creazione o la modifica delle funzioni.

**TinyFaaS** è stato poi confrontato con le piattaforme FaaS presenti allo stato dell’arte, dimostrando che per gli scenari di elaborazione IoT, supera i sistemi esistenti di almeno un ordine di grandezza.

## 3.2 Un framework decentralizzato per il Serverless Edge Computing nell’IoT

Per cercare di conciliare le necessità del paradigma FaaS con quelle di un ambiente di edge, oltre che realizzare delle piattaforme decentralizzate leggere in grado di far fronte all’eterogeneità dei dispositivi edge, è anche necessario affrontare il tema del bilanciamento del carico. Uno degli obiettivi fondamentali dell’Edge Computing è quello di spostare la computazione più vicino all’utente, in modo da minimizzare i tempi di risposta dovuti al trasferimento delle informazioni. Cicconetti et. al. [2], difatti, propongono un framework decentralizzato per la distribuzione efficiente dell’esecuzione di task stateless, con l’obiettivo di minimizzare i tempi di risposta. Questo framework FaaS è pensato per essere leggero e semplice, in modo che possa essere eseguito anche su dispositivi con risorse limitate, classici dell’ambiente di Edge Computing.

In questa trattazione si pone l’enfasi sulla realizzazione di uno scenario come quello mostrato in Figura 3.2. In questo scenario i task stateless vengono eseguiti da dei nodi edge con risorse computazionali limitate, come ad esempio degli access point WiFi o dei gateway per l’IoT. In uno scenario intrinsecamente distribuito come quello dell’Edge Computing, questi dispositivi hanno una connettività limitata verso i data center remoti, che rendono impraticabile scaricare il carico su di essi verticalmente (*vertical offloading*). Infine, sempre in questo ambiente, non esiste un unico punto di accesso, ma i client comunicheranno con l’edge node più vicino. In questo scenario gli autori hanno immaginato che ogni nodo di edge, o un piccolo cluster di essi, esegue una piattaforma FaaS presente in commercio. Questo consente di realizzare uno smistamento del carico verso gli altri nodi di edge (*horizontal offloading*), dove la distribuzione del carico avviene tra le varie piattaforme serverless nella rete di nodi edge interconnessi.

In [2] viene proposta un architettura decentralizzata in cui sono presenti le seguenti componenti, mostrate in Figura 3.3:

- **Edge Computers** (e-computers). Dispositivi di rete che offrono le proprie capacità computazionali per l’esecuzione di funzioni richieste dai client.

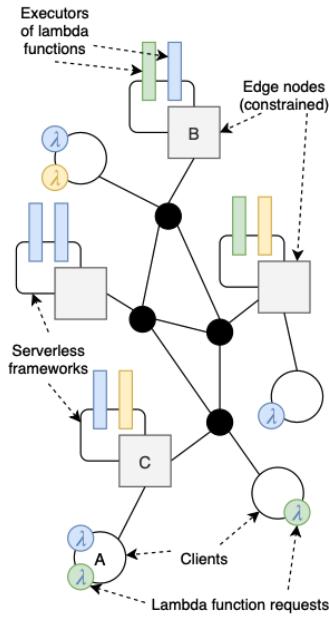


Figura 3.2: Serverless Computing in uno scenario decentralizzato di Edge Computing [2].

- **Edge Routers** (e-routers). Sono i nodi di edge, ovvero i dispositivi utente che hanno accesso alla rete. Tali dispositivi prendono, autonomamente, decisioni su dove inoltrare le richieste di esecuzione di funzioni serverless, tra i diversi tipi di esecutori (e-computers), per poi ritornare la risposta al client. Per far sì che su questi dispositivi, con capacità di elaborazione limitate, possano essere prese decisioni su dove inoltrare le richieste, il processo decisionale avviene in due sotto fasi più semplici:
  1. La prima è una fase di **aggiornamento dei pesi** ( $w_{i,d}^s$ ) che vengono assegnati dall'e-router ( $s$ ) in esame per ogni e-computer di destinazione ( $d$ ) e per ogni tipo di funzione ( $i$ ). Questi pesi rappresentano una misura del costo di eseguire la funzione  $i$  sull'e-computer  $d$ , dal punto di vista dell'e-router ( $s$ ). Minore sarà il costo, migliore sarà la destinazione.
  2. La seconda è una fase di **scelta della destinazione**. In questa fase vengono implementate diverse politiche (*policy*) di scelta della destinazione migliore con diverse funzioni obiettivo, adatte a use-case differenti. Questo dimostra

che il framework è flessibile e facilmente estendibile. Indipendentemente dalla politica di scelta, viene selezionato quello che il miglior e-computer ( $d$ ) a cui inoltrare la richiesta di esecuzione per la funzione  $i$ , tra tutte le possibili destinazioni, ad ognuna delle quali è associato un peso ( $w_{i,d}^s$ ).

- **Controller SDN** (Software Defined Networking). Ha il compito di mantenere la connettività e fornisce informazioni aggiornate sulla topologia e sulla congestione della rete.
- **Edge Controller** (e-controller). Consente di scoprire le caratteristiche degli e-computers nel dominio e configura di conseguenza gli e-router in modo che possano mantenere le loro tabelle di inoltro (e-tables).

Da questa trattazione e dall'implementazione di DFaaS [1], presentata in Sezione 2.3.1, è stata ripresa l'idea della realizzazione di un algoritmo distribuito che avesse come obiettivo quello di calcolare i pesi di inoltro delle richieste ai nodi vicini. Sempre dalla trattazione di Cicconetti et. al. [2] si è presa l'ispirazione per la comparazione della strategia implementata e descritta in Sezione 5.2, con una strategia di inoltro casuale (vedi Sezione 5.4). Difatti in [2], gli autori implementano una politica di scelta casuale per la selezione della destinazione a cui inoltrare le richieste.

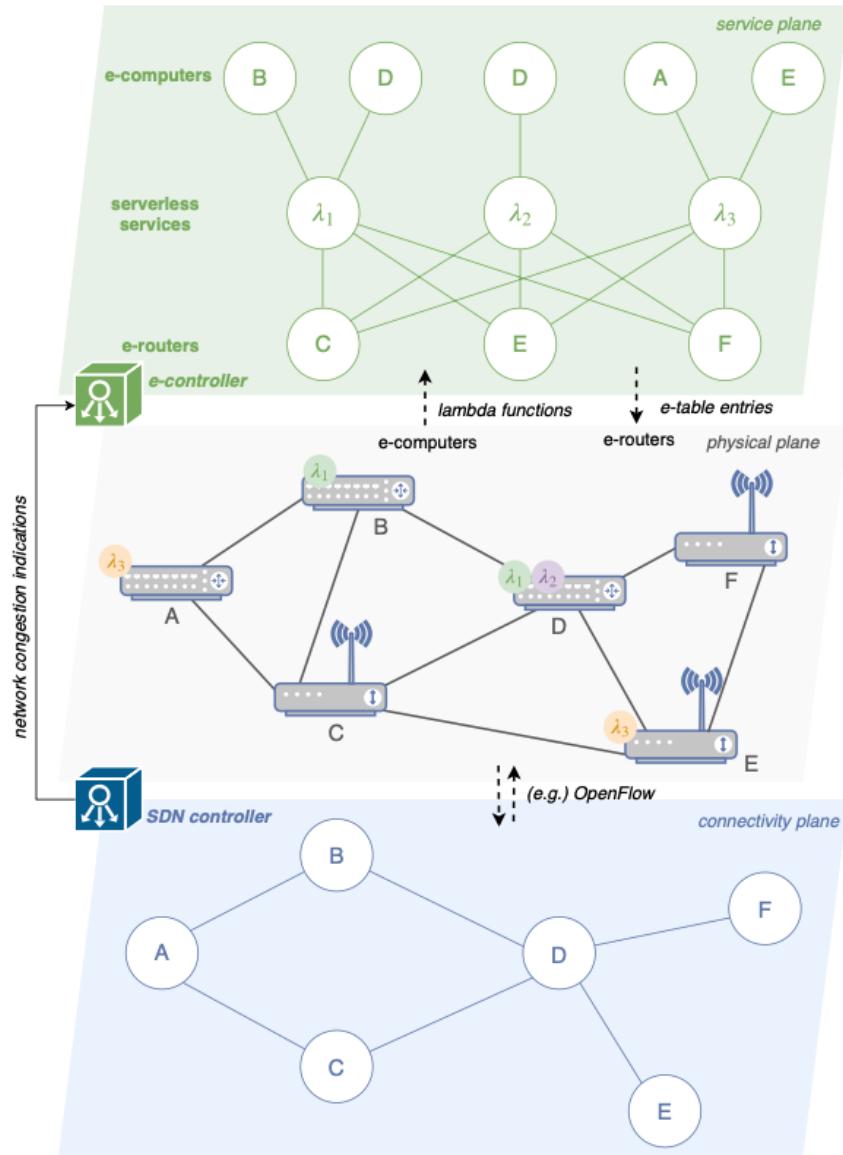


Figura 3.3: Modello del sistema proposto, che mostra il piano di servizio (come percepito dal *e-controller*) e il piano di connettività (come percepito dal *controller SDN*) [2].

### 3.3 RACER: Resource Allocation Control Engine with Reinforcement learning

In un ambiente come quello dell'Edge Computing, i nodi edge hanno a disposizione un quantitativo di risorse limitato rispetto ai tradizionali data center; proprio per questa ragione, in questo contesto, è necessario affrontare il problema relativo alla distribuzione del carico. Un altro sforzo atto a bilanciare il traffico di richieste di esecuzione di funzioni serverless come quello presentato in Sezione 3.2, è quello introdotto da Cho et. al. [3]. Gli autori, in questa trattazione, utilizzano un approccio di *Reinforcement Learning* per affrontare il problema del bilanciamento del carico in un ambiente **edge-cloud gerarchico** (Figura 3.4).

Il lavoro presentato in questa sezione, seppur condivida il medesimo obiettivo di questa tesi, ne differisce per l'approccio utilizzato. Difatti, Cho et. al. [3] presentano una tecnica basata su *Reinforcement Learning* per affrontare il problema del bilanciamento del carico in un ambiente distribuito, mentre l'algoritmo ideato in questa tesi, presentato in Sezione 5.2, si basa su evidenze raccolte e scambiate dai nodi di edge, interconnessi da una rete p2p.

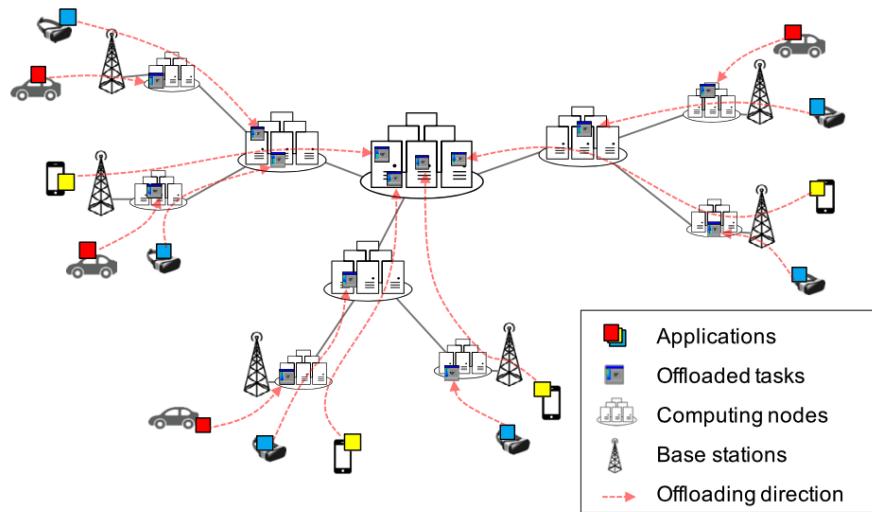


Figura 3.4: Infrastruttura di edge computing distribuita gerarchicamente [3].

In questo caso il carico viene distribuito tra gli edge e dei nodi di computazione remota, organizzati in una rete gerarchica, sulla base delle caratteristiche dei task, come ad esempio la loro QoS (Quality of Service), il costo di esecuzione, i dati in input richiesti, ecc. Per esempio, un task *mission-critical* o *delay-sensitive*, ovvero che ha dei vincoli stringenti per quanto riguarda i ritardi di esecuzione, dovrà per forza essere eseguito in un nodo edge.

Come mostrato in Figura 3.4, in questo studio viene considerata un'infrastruttura gerarchica di Edge Computing distribuita. In questo ambiente si suppone che i dispositivi utente (UEs) eseguono le applicazioni e hanno dei task, a grana fine, che devono essere scaricati sui nodi di elaborazione nell'infrastruttura d'esecuzione. L'infrastruttura di rete è organizzata gerarchicamente come segue:

- **Tier-1.** A questo livello appartengono i nodi che sono collocati in una Base Station (BS), ovvero quelli più vicino all'utente. Tali nodi eseguono i task delegati dai dispositivi utente (UEs) collegati a quella BS. Il ritardo tra UE e nodi appartenenti al tier-1 sarà minimo. D'altro canto è però necessario salvaguardare l'utilizzo dei nodi di questo livello, in quanto hanno delle risorse limitate. Un gruppo di nodi appartenenti al tier-1 sono connessi ad un nodo del tier-2.
- **Tier-2.** I nodi appartenenti a questo livello hanno a disposizione più risorse computazionali dei nodi al tier-1. Questi nodi intermedi saranno costituiti da un certo numero di host. Il ritardo tra gli UE e i nodi di questo livello sarà significativamente maggiore rispetto a quelli di tier-1, ma molto inferiore rispetto a quelli di tier-3.
- **Tier-3.** A questo livello appartiene il nodo remoto in cloud, ovvero un data center centralizzato, con un quantitativo di risorse pressoché illimitato. Seppur le risorse siano elevatissime, il ritardo sperimentato dagli UE, inoltrando i task a questo nodo remoto, è molto più significativo rispetto a quello dei livelli inferiori.

Per affrontare questo problema Cho et. al. hanno proposto **RACER** (Resource Allocation Conrol Engine with Reinforcement learning), ovvero un approccio di

*Deep Reinforcement Learning* per la risoluzione del problema della distribuzione del carico, cercando di rispettare quelli che sono dei vincoli di QoS legati al tempo di risposta per ogni task. RACER, sfruttando questo tipo di tecniche, si presta bene ad affrontare problemi fortemente dinamici come quelli del bilanciamento del traffico, in quanto questo può variare in un qualsiasi momento. Infatti, ripetendo il processo di *osservazione-azione-ricompensa* nell’ambiente edge-cloud gerarchico, l’agente di *Reinforcement Learning* apprende una sequenza di azioni desiderabile per una distribuzione efficiente del carico di lavoro.

Se ad esempio un task è *time-sensitive* e non richiede un utilizzo massivo di risorse per essere eseguito, allora l’UE preferirà inoltrarlo per l’esecuzione ad un nodo appartenente al tier-1, ovvero collocato sulla BS. Questo perché il ritardo tra un UE e un nodo di tier-1 sarà minimo. D’altro canto, un task molto pesante ma senza un particolare vincolo di QoS legato al tempo di risposta, verrà eseguito su un nodo appartenente al tier-2, o sul nodo remoto.

Infine, RACER è stato implementato e testato su *DFaaSCloud* [35], un simulatore ad eventi discreti per l’esecuzione di FaaS su ambienti di edge-cloud distribuiti. *DFaaSCloud* rappresenta un’estensione di *CloudSim* [36], un noto framework di simulazione per i sistemi di cloud computing.

Anche in questo lavoro di tesi è stato valutato l’utilizzo di *DFaaSCloud* [35] per l’implementazione e la verifica dell’algoritmo empirico (presentato in Sezione 5.2), ma poi si è prediletta l’implementazione di un simulatore ex-novo e pensato ad-hoc per il problema affrontato (presentato nel Capitolo 4).

# Capitolo 4

## Architettura del Simulatore

Di seguito viene presentata l'architettura del simulatore che è stato realizzato al fine di ottenere un ambiente isolato in cui poter testare e comparare le diverse strategie di distribuzione del carico. L'intero simulatore è stato scritto cercando di rispettare i principi di una buona architettura software: modularizzando le componenti, cercando di attribuire ad ognuna di essi una singola responsabilità e rendendolo quanto più generico possibile. Il tutto per cercare di garantire una buona manutenibilità, riusabilità ed evolvibilità del sistema. L'intero simulatore è stato scritto utilizzando come linguaggio di programmazione Python [37], ed è reperibile nella cartella *simulation* in [4].

Di seguito vengono elencati, con una breve spiegazione, i termini principali utilizzati nella descrizione delle componenti costituenti il simulatore:

- **Step.** Rappresenta un'iterazione del ciclo di simulazione; ovvero un istante di simulazione.
- **Simulation Controller.** Componente dedicata a coordinare l'esecuzione di tutti gli altri moduli. Rappresenta l'entry-point dell'applicazione.
- **Instance Generator.** Componente dedicata alla generazione dell'istanza che dovrà essere simulata.
- **Simulator.** Componente dedicata all'esecuzione vera e propria della simulazione.

- **Analyzer.** Componente dedicata all’analisi dei dati prodotti dal simulator per produrre gli indici di confronto.
- **Data Loader.** Componente che si interpone tra il simulator e il database manager; tra la richiesta e l’effettivo caricamento dei dati. Si occupa di caricare alla prima invocazione tutti i dati sfruttando l’interfaccia offerta dal database manager.
- **Db Manager.** Componente che rappresenta un generico database manager; gestisce la connessione al database.
- **Exp Db Manager.** Componente dedicata alla gestione e dell’interfacciamento al database contenente i dati di tutti gli esperimenti effettuati.
- **Config Manager.** Componente dedicata a memorizzare tutte le configurazioni: percorsi, nomi di algoritmi, indici utilizzati per il confronto ecc.
- **Config Request.** Classe che rappresenta una richiesta di configurazione. Utilizzata per reperire le metriche relative a una determinata configurazione dal database; si compone di una lista di function request.
- **Function Request.** Classe che rappresenta una richiesta per reperire i dati relativi ad una determinata funzione.
- **Max Rate.** Rappresenta il numero massimo di richieste accettabili, al secondo, da una funzione (r/s).
- **Invocation Rate.** Numero di invocazioni/richieste al secondo che vengono effettuate ad una funzione (r/s).
- **Strategy.** Interfaccia che rappresenta un generico algoritmo di bilanciamento implementato sul simulatore. Tutti gli algoritmi dovranno aderire a questa interfaccia.
- **Agent.** Classe che rappresenta un agente che viene simulato. Esegue una determinata strategia di bilanciamento del carico.

- **Overloaded.** Stato in cui si trova una funzione quando è sovraccaricata, ovvero quando l'invocation rate supera il max rate.
- **Pesi.** Pesi di inoltro, che rappresentano la percentuale di richieste inviate da ogni agente verso gli altri che può raggiungere. Vengono calcolati per ogni funzione nello stato di sovraccarico. Rappresentano, per ogni funzione, una distribuzione di probabilità.
- **Seed.** Seme utilizzato per la generazione dei numeri pseudo casuali. Utilizzato per la riproducibilità degli esperimenti.

## 4.1 Architettura ad alto livello

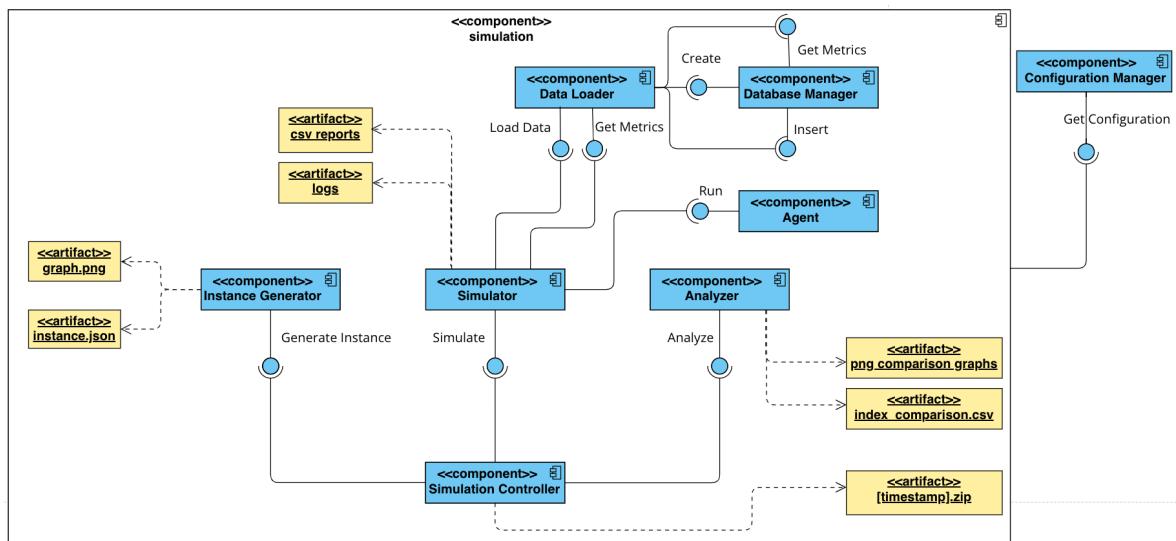


Figura 4.1: Diagramma delle componenti del simulatore.

In Figura 4.1 viene mostrato il diagramma UML (Unified Modeling Language) delle componenti del simulatore. Questo diagramma mostra ad alto livello quali sono i principali moduli che lo costituiscono, rappresentandone anche le interfacce di comunicazione. Le parti in blu rappresentano le componenti del simulatore, mentre quelle in giallo gli artefatti prodotti in output. Ogni modulo a sua volta sarà costituito da un certo numero di classi e/o script, tutte realizzate in Python.

La simulazione avviene in **step**, chiamati anche **istanti di simulazione**; ognuno di essi corrisponde ad una diversa situazione o configurazione che deve essere simulata in termini di traffico da bilanciare. Concettualmente, la simulazione avviene in tre macro-fasi fondamentali, che si traducono poi nelle componenti ad alto livello che verranno descritte di seguito. Le tre macro-fasi sono le seguenti:

1. **Generazione dell'istanza.** In questa prima fase viene generata l'istanza da simulare. Tra le informazioni che devono essere generate per ogni istante di simulazione sono presenti: il numero di agenti, la topologia della rete di comunicazione (ovvero i nodi con cui ogni agente può comunicare), il numero e il tipo delle funzioni che dovranno essere eseguite su ogni agente ed il numero di richieste che dovranno essere inviate ad ognuna di esse.
2. **Simulazione.** In questa seconda fase avviene la simulazione effettiva di ogni configurazione generata per ogni step nella fase precedente. Per ogni configurazione di input sarà necessario reperire dinamicamente i dati necessari a far funzionare gli algoritmi di distribuzione del carico (es. utilizzo di CPU/RAM dei nodi, tempo medio di esecuzione delle funzioni, quante richieste queste funzioni possono ancora servire, ecc...). Per ogni step simulato, una volta reperiti i dati, verranno distribuite le richieste sfruttando i diversi algoritmi di bilanciamento del carico.
3. **Analisi.** Durante quest'ultima fase devono essere analizzati i dati prodotti dalla simulazione e calcolati alcuni indici su cui poter effettuare il confronto dei vari algoritmi.

Nel diagramma riportato in Figura 4.1 vengono riportate tutte le componenti fra di esse, le principali che generano gli artefatti di output e che consentono di realizzare la tre macro-fasi sopra descritte, sono le seguenti:

- **Simulation Controller.** Si occupa di controllare e gestire il flusso del simulatore, coordinando quelle che sono le altre principali componenti e la loro esecuzione. Gestisce inoltre quello che è l'output finale della fase di simulazione, raccogliendo

all'interno di un archivio compresso (in formato zip) l'output delle singole fasi di simulazione.

- **Instance Generator.** Si occupa di generare l'istanza che dovrà essere simulata. Come output produce due artefatti, ovvero la rappresentazione dell'istanza in formato json, e l'immagine contenente il grafo rappresentante la rete di comunicazione degli agenti.
- **Simulator.** Si occupa di eseguire la simulazione vera e propria. Per ogni configurazione creata dall'*instance generator* richiederà i dati per poter eseguire la simulazione al *data loader*, che a sua volta utilizzerà il *database manager* per reperirli. Come output genera due categorie di artefatti: i report in formato csv (Comma-Separated Values) che riepilogano come è avvenuto l'inoltro delle richieste per ogni istante di simulazione e per ogni algoritmo simulato; i file di log per ogni agente simulato, che mostrano i valori dei pesi calcolati da ognuno di essi per ogni algoritmo e ogni istante di simulazione.
- **Analyzer.** Si occupa di analizzare i dati forniti in output dal *simulator* e calcolare gli indici di confronto per i vari algoritmi implementati. Come artefatti di output produce un file csv che compara in forma tabellare gli indici calcolati per le varie strategie e una serie di grafici di confronto.

## 4.2 Instance Generator

Questa componente si occupa di generare l'istanza che dovrà essere simulata. Accetta in input tre argomenti forniti al fine di consentire all'utente di personalizzare l'istanza da creare e per permettere la riproducibilità degli esperimenti. Gli argomenti passati alla CLI (Command-Line Interface) sono tutti opzionali in quanto ad essi è stato attribuito un valore di default. Gli argomenti che è possibile passare sono i seguenti:

- **Numero di agenti.** Il numero di agenti corrisponde al numero di nodi che devono essere utilizzati durante la simulazione. Siccome su ogni nodo viene eseguito un agente, questi due numeri corrispondono.
- **Seed.** Ovvero il numero casuale che può essere utilizzato per garantire la riproducibilità degli esperimenti, questo tema è molto importante in un contesto di simulazione e analisi dati.
- **Probabilità di creazione di un edge.** Rappresenta la probabilità di creazione di un edge tra ogni coppia di nodi.

Al momento per creare l’istanza vengono utilizzati i dati raccolti sperimentalmente, che verranno poi descritti nel Capitolo 5, in modo che il *simulator* possa poi effettuare richieste di dati presenti nel database gestito dal *database manager*.

Per far questo viene caricato un numero di configurazioni pari al numero di agenti che si vogliono simulare. Successivamente, viene creato un grafo non orientato e connesso con topologia casuale, nel quale ogni possibile edge viene creato con una probabilità pari a quella passata come argomento. Il grafo viene creato connesso in modo da assicurare che non esistano nodi isolati nella rete, che corrisponderebbero ad agenti che non sono in grado di comunicare con nessun altro nodo.

Come è possibile vedere in Figura 4.2 uno dei due artefatti prodotti da questa componente è l’immagine rappresentante la topologia di rete creata casualmente.

Infine, viene creata e poi esportata su file la rappresentazione dell’istanza in formato json, in modo che possa essere consultata e utilizzata dalle componenti successive.

Come è possibile vedere in Figura 4.3, l’istanza creata è costituita dai parametri con cui è stata generata (**seed**, **nodes\_number** e **edge\_prob**) e, per ogni chiave rappresentata dall’identificativo di un nodo (es. **node\_1**), le seguenti informazioni:

- **Replicas.** Contiene il numero di repliche per ogni funzione in esecuzione su quel nodo. Il valore di questa chiave è un json innestato in cui per ogni funzione (chiave) è associato il numero di repliche (valore).

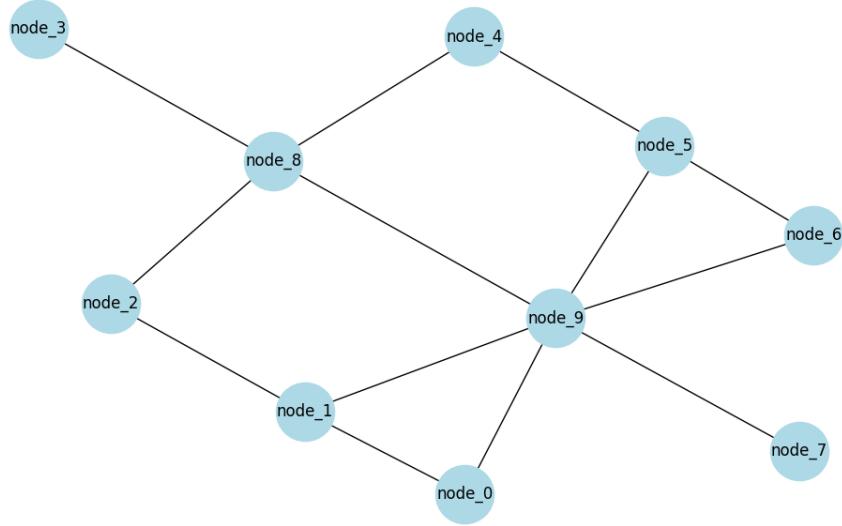


Figura 4.2: Immagine rappresentante la topologia di rete.

- **Node Type.** Rappresenta il tipo di nodo; come descritto nel Capitolo 5 sono stati utilizzati tre tipi di nodi per la raccolta dati, ognuno dei quali con diverse caratteristiche in termini di risorse.
- **Neighbours.** Contiene la lista dei vicini per il nodo di interesse. Questa lista contiene gli identificativi dei nodi con cui è possibile scambiare informazioni al fine di bilanciare il traffico.
- **Exp. History.** Contiene una lista di json innestati. Ogni elemento della lista corrisponde alla configurazione del nodo in un diverso istante di simulazione (o step). Ogni elemento di questa lista associa ad una chiave “**functions**” una lista di json, ognuno dei quali è costituito da due coppie chiave-valore:
  - **Name.** A questa chiave viene associato il nome della funzione.
  - **Invoc. Rate.** A questa chiave viene associato il numero di invocazioni che si vogliono inviare a quella classe di funzione, in quello specifico nodo ed in quel determinato step.

```

1  {
2      "seed": 2833,
3      "nodes_number": 10,
4      "edge_prob": 0.1,
5      "node_0": {
6          "replicas": {
7              "funca": 2,
8              "qrcode": 2,
9              "ocr": 1
10         },
11         "node_type": "node_1",
12         "neighbours": [
13             "node_9",
14             "node_1"
15         ],
16         "exp_history": [
17             {
18                 "functions": [
19                     {
20                         "name": "funca",
21                         "invoc_rate": 0
22                     },
23                     {
24                         "name": "ocr",
25                         "invoc_rate": 0
26                     },
27                     {
28                         "name": "qrcode",
29                         "invoc_rate": 0
30                     }
31                 ]
32             },
33             {
34                 "functions": [
35                     {
36                         "name": "funca",
37                         "invoc_rate": 18
38                     },
39                     {
40                         "name": "ocr",
41                         "invoc_rate": 0
42                     },
43                     {
44                         "name": "qrcode",
45                         "invoc_rate": 1
46                     }
47                 ],
48             }
49         ]
50     },
51     "functions": [
52         {
53             "name": "funca",
54             "invoc_rate": 0
55         },
56         {
57             "name": "qrcode",
58             "invoc_rate": 0
59         },
60         {
61             "name": "ocr",
62             "invoc_rate": 0
63         }
64     ]
65 }

```

Figura 4.3: Parte del file json rappresentante l’istanza generata.

## 4.3 Simulator

Questa componente rappresenta il vero e proprio cuore pulsante del simulatore in quanto si occupa di simulare la distribuzione del carico utilizzando i diversi algoritmi, sulla base dell’istanza che è stata generata precedentemente dall’*instance generator* o che è

stata fornita dal *simulation controller*.

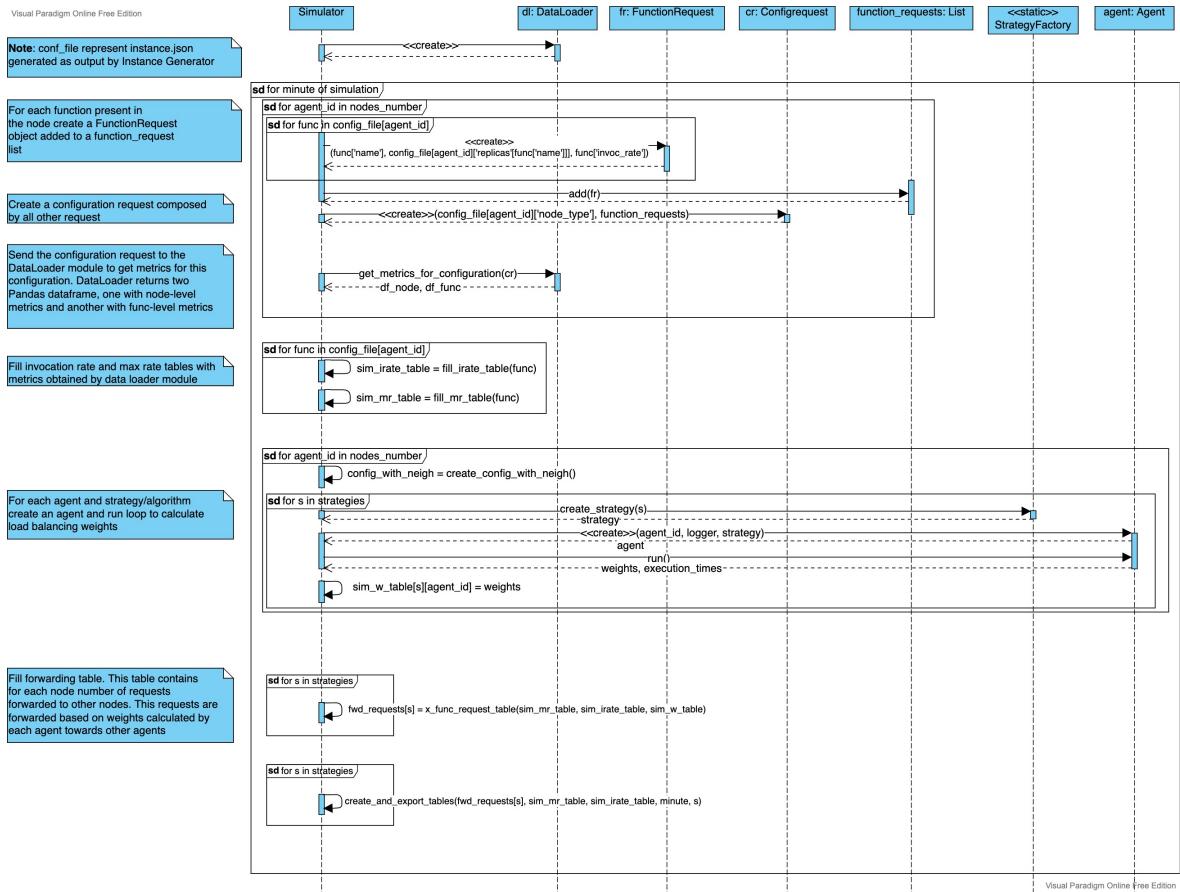


Figura 4.4: Diagramma di sequenza che rappresenta le operazioni eseguite dal componente *simulator*.

In Figura 4.4 viene riportato il diagramma UML di sequenza, che rappresenta come il *simulator* organizza ed esegue le operazioni, e di quali oggetti si avvale per farlo.

Il *simulator* lavora utilizzando come base un file di configurazione (*config\_file* in Figura 4.4), che corrisponde al file di istanza creato dall'*instance generator*.

Di seguito, per ogni step di simulazione (rappresentato dal ciclo più esterno come minuto di simulazione), vengono eseguite una moltitudine di operazioni, che porteranno poi in output i pesi calcolati da ogni agente per ogni strategia applicata.

Come si evince dal diagramma in Figura 4.4 anche il simulatore implementa diverse operazioni che vengono eseguite in sequenza. Di seguito, queste verranno raggruppate in macro-sezioni e descritte più nel dettaglio.

### 4.3.1 Reperimento dei dati necessari alla simulazione

Per prima cosa il *simulator* si deve occupare di analizzare il file di istanza ricevuto in input e reperire per ogni istante di simulazione tutte le metriche relative alle funzioni in esecuzione su ogni nodo, in base alle configurazioni descritte nel file di istanza (vedi Figura 4.3). Per ogni funzione in esecuzione su ogni nodo, viene riportato il numero di repliche e nei diversi step temporali viene riportato il numero di invocazioni a cui deve essere sottoposta (**invoc\_rate**).

Tali dati sono utilizzati per creare le richieste (*FunctionRequest* e *ConfigRequest*) per poi effettuare la richiesta di metriche al *data loader*.

Queste metriche, a differenza degli altri simulatori, non sono stimate sfruttando dei modelli, ma bensì sono degli indici di performance raccolti sperimentalmente sul sistema di DFaaS [1]. Infatti, come descritto in Sezione 5.1.2, queste metriche sono state ottenute effettuando benchmarking sul sistema reale; rappresentano quindi delle evidenze empiriche e non dei dati stimati o interpolati.

Come è possibile vedere in Figura 4.5, una *ConfigRequest* è una classe rappresentante una richiesta di configurazione da parte del *simulator* al *data loader* e che contiene: il tipo di nodo a cui la configurazione fa riferimento (le tipologie di nodo utilizzate per la raccolta dati sono dettagliata in Tabella 5.1) e una lista di richieste di funzioni (*FunctionRequest*), una per ogni funzione in esecuzione nella configurazione di interesse. La classe *FunctionRequest* rappresenta una richiesta per una specifica funzione in esecuzione su un determinato nodo, identificata da un nome, un numero di repliche e a cui è associato un workload (ovvero un rate di invocazioni al secondo, denominato **invoc\_rate** nel file di istanza).

Per ogni minuto di simulazione viene estratta per ogni agente la configurazione da utilizzare per creare le richieste da inoltrare al *data loader*. Questo viene fatto ciclando su tutte le funzioni presenti su un nodo in un determinato istante e creando una *FunctionRequest*, che incapsula i dati relativi al nome della funzione e al rate di invocazione a cui deve essere sottoposta.

Come è possibile vedere in Figura 4.5 il *simulator* per ogni step di simulazione crea

una richiesta di configurazione (*ConfigRequest*) relativa ad ogni agente (quindi ne creerà in numero pari al numero di agenti, specificato nel file di istanza in corrispondenza della chiave **nodes\_num**). Tale richiesta di configurazione contiene il tipo di nodo a cui fa riferimento e si compone di una lista di richieste di funzioni, ognuna delle quali rappresentata da un oggetto della classe *FunctionRequest*. Ogni *FunctionRequest* è dettagliata da un nome di funzione, un numero di repliche e un workload (ovvero un invocation rate). Come è possibile notare, tutte le informazioni necessarie a creare tali richieste sono reperibili dal file di istanza.

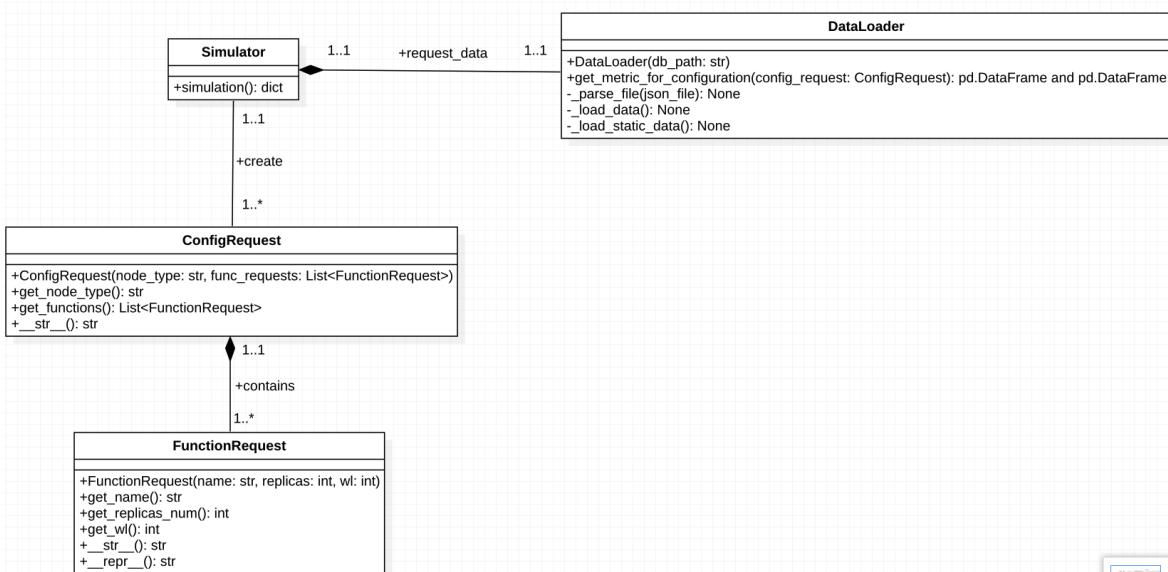


Figura 4.5: Diagramma delle classi che rappresenta come interagiscono e sono organizzati il *simulator* e il *data loader*, sfruttando le *FunctionRequest* e le *ConfigRequest*.

Fatto questo il simulatore avrà all'interno della richiesta di configurazione tutti i dati necessari da inoltrare al *data loader* per richiedere le metriche che serviranno poi all'esecuzione degli algoritmi. Tale richiesta avviene invocando il metodo `get_metrics(config_request)` e passando come parametro la richiesta di configurazione appena creata. Il modo in cui il *data loader* caricherà i dati sarà descritto nelle sezioni successive.

### 4.3.2 Caricamento dati da Database

In risposta alla ricezione di una richiesta di caricamento delle metriche da parte del *simulator*, il *data loader* si deve occupare di caricare effettivamente i dati, e per fare questo sfrutta un altro componente, il *database manager*.

Il *data loader* è quel componente che si interpone tra le richieste del *simulator* e l'effettivo caricamento dei dati da database. Tale componente, quando creato si occupa di caricare per la prima volta tutti i dati raccolti sperimentalmente sul database, interfacciandosi al *database manager* per eseguire le effettive query di inserimento.

Come si può vedere in Figura 4.6 il *data loader* effettua la richiesta di ottenimento delle metriche, richiamando il metodo *get\_metrics(config\_request)* del *database manager*, e ritornandone i risultati. Quest'ultimo componente si occuperà di gestire il database (realizzato con SQLite [38]) e di esporre tutti i metodi per creare tabelle, inserire dati, reperirli, ecc. La descrizione di come è stato organizzato il database è riportata in Sezione 4.4.

Il *data loader* contiene infatti un'istanza del *database manager*, rappresentato dalla classe *DbManager*. All'interno di questa istanza, sarà però contenuto un oggetto di tipo *ExpDbManager*, ovvero un oggetto appartenente alla classe che rappresenta il database contenente i dati relativi agli esperimenti raccolti. La classe *DbManager* rappresenta la superclasse che si occupa di creare la connessione al database. La sottoclasse *ExpDbManager*, invece, gestisce l'accesso effettivo ai dati per il database utilizzato in questo lavoro di tesi.

Una volta servita la richiesta il *data loader* ritorna quelle che sono le metriche reperite dal database per quella specifica richiesta di configurazione. I dati vengono ritornati sotto forma di due DataFrame Pandas, ovvero organizzati in maniera tabulare: uno contiene le metriche a livello di nodo (es. utilizzo di CPU, RAM sul nodo) e uno a livello di funzione (es. margine di richieste servibili, tempo medio di esecuzione ecc...).

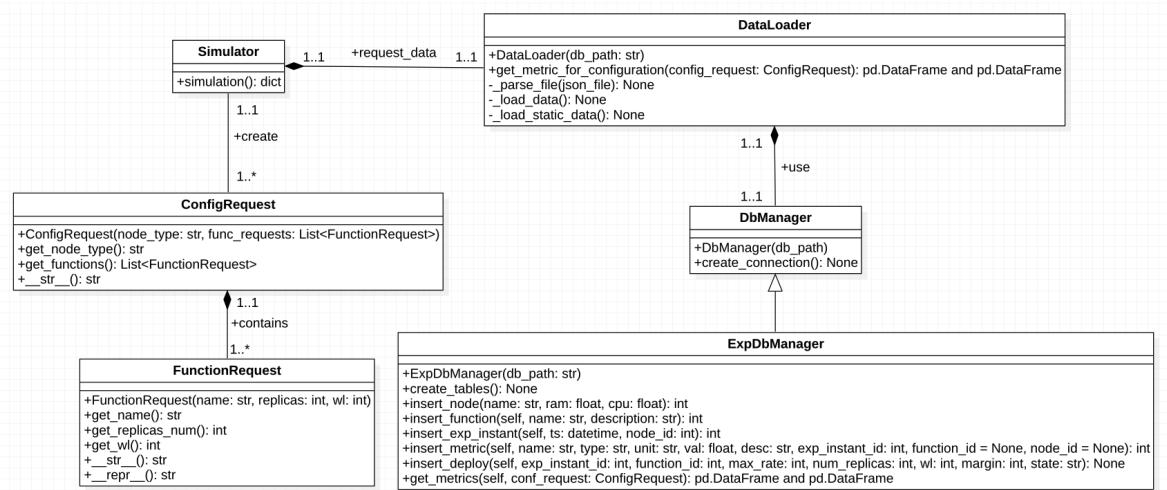


Figura 4.6: Diagramma delle classi che mostra l’organizzazione del *data loader* e del *database manager*, connesse alle componenti precedenti.

### 4.3.3 Esecuzione della simulazione

Una volta reperiti i dati necessari ad eseguire i vari algoritmi, il *simulator* procede con la loro esecuzione. Ogni agente, che utilizza la propria configurazione, esegue le strategie di distribuzione del carico implementate (descritte nel Capitolo 5), e con ognuna di esse calcola i pesi di inoltro delle richieste verso gli altri nodi della rete.

Alcuni degli algoritmi implementati richiedono una fase di comunicazione di alcune informazioni con il proprio vicinato, ma questa fase durante la simulazione è stata rimossa. Per evitare di implementare effettivamente la comunicazione, ad ogni agente viene passato un file di configurazione completo, con le informazioni del proprio vicinato. Questo è stato fatto in modo tale da testare l’effettiva efficacia degli algoritmi, senza concentrarsi su aspetti relativi alla comunicazione tra i diversi agenti.

Come si evince in Figura 4.7, il *simulator* crea le diverse strategie utilizzando la classe *StrategyFactory*, ovvero una classe che si occupa della creazione della corretta strategia indicata come parametro. Questa classe rappresenta l’implementazione del pattern creazionale **Factory** [39].

Ogni agente incapsula un campo di tipo *Strategy*, ovvero del tipo dell’interfaccia che rappresenta una generica strategia. Ogni strategia implementata nel simulatore

realizzerà questa interfaccia e, come da contratto, dovrà implementare un metodo *run()* che ritornerà un dizionario contenente i pesi calcolati. Ogni agente quindi, ad ogni invocazione, eseguirà il metodo *run()* e ritornerà al simulatore il dizionario da esso restituito.

Il simulatore, a questo punto, ha una visione globale dei pesi ritornati da ogni agente verso gli altri nodi, ed in base ad essi sarà in grado di calcolare le richieste inoltrate verso ognuno degli altri nodi. I pesi ritornati da ogni agente potranno essere diversi da zero solo per i nodi appartenenti al suo vicinato, mentre tutti gli altri saranno uguali a zero.

Implementando le varie strategie utilizzabili dall'agente in questo modo è stato utilizzato il pattern comportamentale **Strategy** [40], che consente di realizzare una famiglia di algoritmi al di sotto di un'interfaccia comune e di renderli intercambiabili in maniera trasparente. L'agente richiamerà quindi semplicemente il metodo *run()*, senza preoccuparsi di quale algoritmo andrà ad eseguire.

L'utilizzo di questo pattern consente quindi una rapida espansione di quelle che sono le strategie implementate nel simulatore. Ne può essere aggiunta una molto semplicemente, creando una classe che aderisca all'interfaccia *Strategy*, ed implementi il nuovo algoritmo all'interno del metodo *run()*, ritornando un dizionario contenente i pesi calcolati.

In questo modo è anche possibile cambiare in maniera trasparente la strategia utilizzata dall'agente, e modificarla in corso d'opera, tramite l'utilizzo del metodo *set\_strategy(\_ behaviour)*.

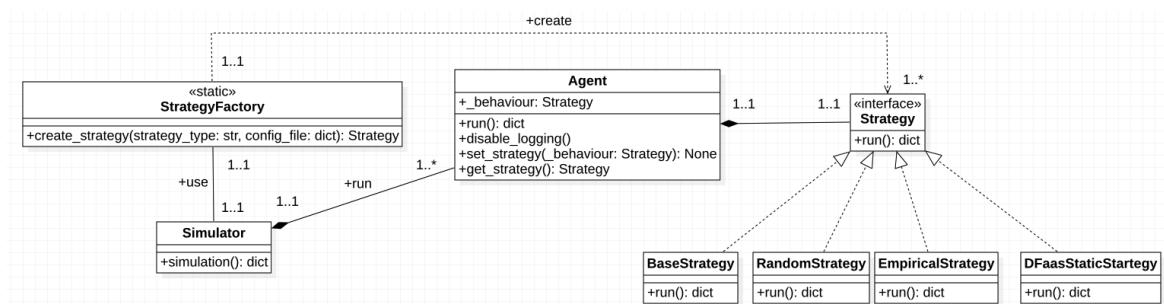


Figura 4.7: Diagramma delle classi che mostra come interagiscono il *simulator* e gli *agenti*.

Gli agenti, durante la loro esecuzione, scrivono quelli che rappresentano il primo artefatto prodotto dal *simulator*, ovvero i file di log. Ogni algoritmo scrive informazioni a diversi livelli di logging, ma quelle che più comunemente vengono utilizzate ai fini di debug sono i pesi prodotti al termine dell'esecuzione di ogni strategia, per ogni step/minuto di simulazione. In Figura 4.8 viene riportato quello che è il file di log prodotto da un agente durante un esperimento.

```

10   ----- MINUTE 1 -----
11   | > STRATEGY: base_strategy <
12   Weights normalized for func funca: {'node_0': 0, 'node_2': 0}
13   Weights normalized for func ocr: {'node_0': 0, 'node_2': 0}
14   | > STRATEGY: random_strategy <
15   Weights normalized for func funca: {'node_0': 1.36986301369863, 'node_2': 98.63013698630137}
16   Weights normalized for func ocr: {'node_0': 48.19277108433735, 'node_2': 51.80722891566265}
17   | > STRATEGY: empirical_strategy <
18   Weights normalized for func funca: {'node_0': 30.823059523131107, 'node_2': 69.1769404768689}
19   Weights normalized for func ocr: {'node_0': 24.801007853871667, 'node_2': 75.19899214612835}
20   | > STRATEGY: dfaas_static_strategy <
21   Weights normalized for func funca: {'node_0': 9.75609756097561, 'node_2': 90.2439024390244}
22   Weights normalized for func ocr: {'node_0': 16.666666666666664, 'node_2': 83.3333333333334}
23
24
25   ----- MINUTE 2 -----
26   | > STRATEGY: base_strategy <
27   Weights normalized for func ocr: {'node_0': 0, 'node_2': 0}
28   Weights normalized for func funca: {'node_0': 0, 'node_2': 0}
29   | > STRATEGY: random_strategy <
30   Weights normalized for func ocr: {'node_0': 61.80555555555556, 'node_2': 38.1944444444444}
31   Weights normalized for func funca: {'node_0': 3.79746835443038, 'node_2': 96.20253164556962}
32   | > STRATEGY: empirical_strategy <
33   Weights normalized for func ocr: {'node_0': 32.9229561325489, 'node_2': 67.0770438674511}
34   Weights normalized for func funca: {'node_2': 100.0, 'node_0': 0}
35   | > STRATEGY: dfaas_static_strategy <
36   Weights normalized for func ocr: {'node_0': 16.666666666666664, 'node_2': 83.3333333333334}
37   Weights normalized for func funca: {'node_0': 0.0, 'node_2': 100.0}
38   ----- MINUTE 3 -----
39   | > STRATEGY: base_strategy <
40   Weights normalized for func ocr: {'node_0': 0, 'node_2': 0}
41   Weights normalized for func funca: {'node_0': 0, 'node_2': 0}
42   | > STRATEGY: random_strategy <
43   Weights normalized for func ocr: {'node_0': 53.142857142857146, 'node_2': 46.85714285714286}
44   Weights normalized for func funca: {'node_0': 62.03703703703704, 'node_2': 37.96296296296296}
45   | > STRATEGY: empirical_strategy <
46   Weights normalized for func ocr: {'node_0': 16.817150058281005, 'node_2': 83.18284994171898}
47   Weights normalized for func funca: {'node_2': 100.0, 'node_0': 0}
48   | > STRATEGY: dfaas_static_strategy <
49   Weights normalized for func ocr: {'node_0': 16.666666666666664, 'node_2': 83.3333333333334}
50   Weights normalized for func funca: {'node_0': 0.0, 'node_2': 100.0}
51
52
53
54
55   ----- MINUTE 4 -----
56   | > STRATEGY: base_strategy <
57   Weights normalized for func funca: {'node_0': 0, 'node_2': 0}
58   Weights normalized for func ocr: {'node_0': 0, 'node_2': 0}
59   | > STRATEGY: random_strategy <
60   Weights normalized for func funca: {'node_0': 48.87640449438202, 'node_2': 51.12359550561798}
61   Weights normalized for func ocr: {'node_0': 10.714285714285714, 'node_2': 89.28571428571429}
62   | > STRATEGY: empirical_strategy <
63   Weights normalized for func funca: {'node_2': 100.0, 'node_0': 0}
64   Weights normalized for func ocr: {'node_0': 25.313991419153226, 'node_2': 74.68600858084676}
65   | > STRATEGY: dfaas_static_strategy <
66   Weights normalized for func funca: {'node_0': 0.0, 'node_2': 100.0}
67   Weights normalized for func ocr: {'node_0': 16.666666666666664, 'node_2': 83.3333333333334}

```

Figura 4.8: File di log di un agente che viene eseguito sul nodo1 durante una simulazione; vengono riportati per ogni minuto e strategia i pesi calcolati per ogni funzione.

#### 4.3.4 Organizzazione dei dati di output

Il *simulator*, avendo raccolto i pesi di inoltro delle richieste calcolati da ogni agente verso il proprio vicinato, possiede una visione globale di tutto il sistema al termine di ogni step di simulazione, ed è quindi in grado di calcolare il bilanciamento del carico effettuato da ognuno di essi.

Una volta raccolti i pesi per tutti gli agenti, al termine dello step di simulazione, il *simulator* calcola le richieste inoltrate da ogni istanza di funzione in stato di sovraccarico (*overloaded*) verso gli altri nodi presenti nel proprio vicinato, per ogni strategia utilizzata (questo avviene invocando la funzione *xfunc\_request\_table()*). Per eseguire questo calcolo servono tre tipologie di tabelle:

- **Tabella dei max rate.** Contiene per ogni nodo il numero massimo di invocazioni effettuabili in un secondo per ogni funzione in esecuzione su di esso. Un esempio di questa tabella viene riportato in Tabella 4.1, dove ad esempio per il **nodo\_1** il numero massimo di invocazioni per la funzione **funca** è di 100 r/s.
- **Tabella degli invocation rate.** Contiene per ogni nodo il numero di invocazioni al secondo a cui ogni funzione su esso eseguita deve essere sottoposta in questo step di simulazione. Un esempio di questa tabella viene riportato in Tabella 4.2, dove ad esempio l'invocation rate per la funzione **funca** in esecuzione sul **nodo\_1** è di 177 r/s.
- **Tabelle delle richieste inoltrate.** Ognuna di esse rappresenta una matrice quadrata, in cui per ogni nodo (riga) è rappresentato il numero di richieste inoltrate verso ogni altro nodo (colonna), calcolate utilizzando i pesi ritornati dagli agenti sfruttando le diverse strategie. Lungo la diagonale è possibile trovare il numero di richieste gestite dal nodo stesso. Questo valore corrisponderà all'invocation rate se è minore del max-rate, altrimenti al max-rate, in quanto ogni nodo non può gestire più di max-rate richieste per ogni funzione (riportate nella tabella dei max-rate). Di queste tabelle ne vengono create tante quante sono le funzioni in esecuzione sul sistema, per ogni strategia e per ogni istante di simulazione. Un

esempio di queste tabelle è possibile vederlo per la funzione **funca** in Tabella 4.3, per **qrcode** in Tabella 4.4 e per **ocr** in Tabella 4.5. Ad esempio, come è possibile vedere in Tabella 4.3 il **nodo\_1** letto nella seconda riga, gestisce 100 r/s (pari al suo **max-rate**, vedi Tabella 4.1) e ne inoltra 49 al **node\_2** e 28 al **node\_3**. Il totale delle richieste, e quindi della somma sulla riga del nodo, è pari a 177 r/s, ovvero all'**invocation rate** per la **funca** sul **node\_1** riportata in Tabella 4.2. Come è possibile notare, l'inoltro riportato nelle tabelle sottostanti rispetta quella che è la topologia di rete generata per questa simulazione, riportata in Figura 4.9. Ad esempio il **node\_1**, per tutte e tre le funzioni, inoltra solamente ai nodi 2 e 3, che rappresentano il suo vicinato.

Al termine di questa fase verranno quindi prodotti l'insieme di file di report in formato csv che corrispondono al secondo artefatto esportato dal *simulator*, che serviranno come input alla fase di analisi (eseguita dall'*analyzer*). Per ogni strategia utilizzata dall'agente viene creata in output una cartella, che ne conterrà a sua volta una per ogni istante di simulazione (7 minuti di simulazione di default), al cui interno sarà presente un file csv per la tabella dei max-rate, un file csv per la tabella degli invocation rate e infine un file csv contenente la tabella delle richieste inoltrate per ogni funzione in esecuzione nel sistema.

Tabella 4.1: Tabella riportante i max rate per ogni nodo e per ogni funzione in esecuzione su di esso.

	<b>funca</b>	<b>qrcode</b>	<b>ocr</b>
<b>node_0</b>	30	4	1
<b>node_1</b>	100	10	3
<b>node_2</b>	120	20	5
<b>node_3</b>	30	4	1
<b>node_4</b>	100	10	3

Tabella 4.2: Tabella riportante i gli invocation rate per ogni nodo e per ogni funzione in esecuzione su di esso.

	<b>funca</b>	<b>qrcode</b>	<b>ocr</b>
<b>node_0</b>	39	4	0
<b>node_1</b>	177	0	9
<b>node_2</b>	9	31	0
<b>node_3</b>	19	0	7
<b>node_4</b>	158	0	0

Tabella 4.3: Tabella riportante le richieste inoltrate da ogni nodo verso gli altri nodi della rete, relativa alla funzione funca.

<b>funca</b>	<b>node_0</b>	<b>node_1</b>	<b>node_2</b>	<b>node_3</b>	<b>node_4</b>
<b>node_0</b>	30	0	9	0	0
<b>node_1</b>	0	100	49	28	0
<b>node_2</b>	0	0	9	0	0
<b>node_3</b>	0	0	0	19	0
<b>node_4</b>	0	0	51	7	100

Tabella 4.4: Tabella riportante le richieste inoltrate da ogni nodo verso gli altri nodi della rete, relativa alla funzione qrcode.

<b>qrcode</b>	<b>node_0</b>	<b>node_1</b>	<b>node_2</b>	<b>node_3</b>	<b>node_4</b>
<b>node_0</b>	4	0	0	0	0
<b>node_1</b>	0	0	0	0	0
<b>node_2</b>	0	4	20	0	7
<b>node_3</b>	0	0	0	0	0
<b>node_4</b>	0	0	0	0	0

Tabella 4.5: Tabella riportante le richieste inoltrate da ogni nodo verso gli altri nodi della rete, relativa alla funzione ocr.

<b>ocr</b>	<b>node_0</b>	<b>node_1</b>	<b>node_2</b>	<b>node_3</b>	<b>node_4</b>
<b>node_0</b>	0	0	0	0	0
<b>node_1</b>	0	3	6	0	0
<b>node_2</b>	0	0	0	0	0
<b>node_3</b>	0	0	0	1	6
<b>node_4</b>	0	0	0	0	0

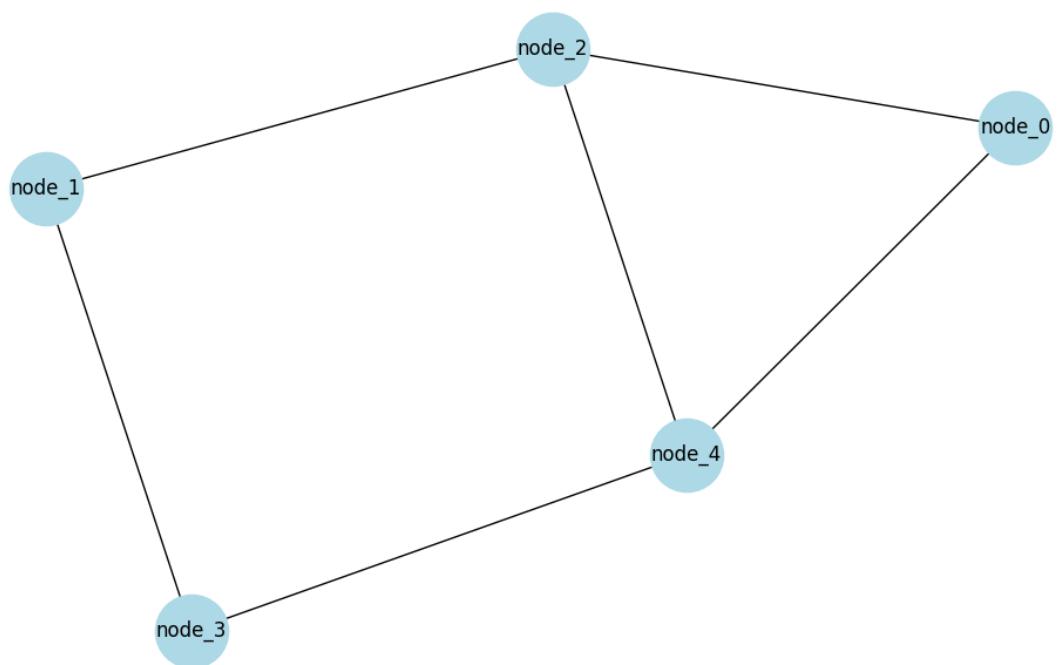


Figura 4.9: Immagine rappresentante la topologia di rete dell'esempio di simulazione riportato nelle tabelle.

## 4.4 Database Manager

Il *DbManager* rappresenta quella componente che viene interrogata dal *dbloader* per inserire e selezionare effettivamente i dati dal database. Rappresenta quindi la componente che comunica direttamente con il database, eseguendo le query su di esso.

Come mostrato in Figura 4.10 la classe *DbManager* rappresenta la superclasse che si occupa di creare la connessione al database. La sottoclasse *ExpDbManager*, invece, gestisce l'accesso effettivo ai dati per il database utilizzato in questo lavoro di tesi, ovvero SQLite.

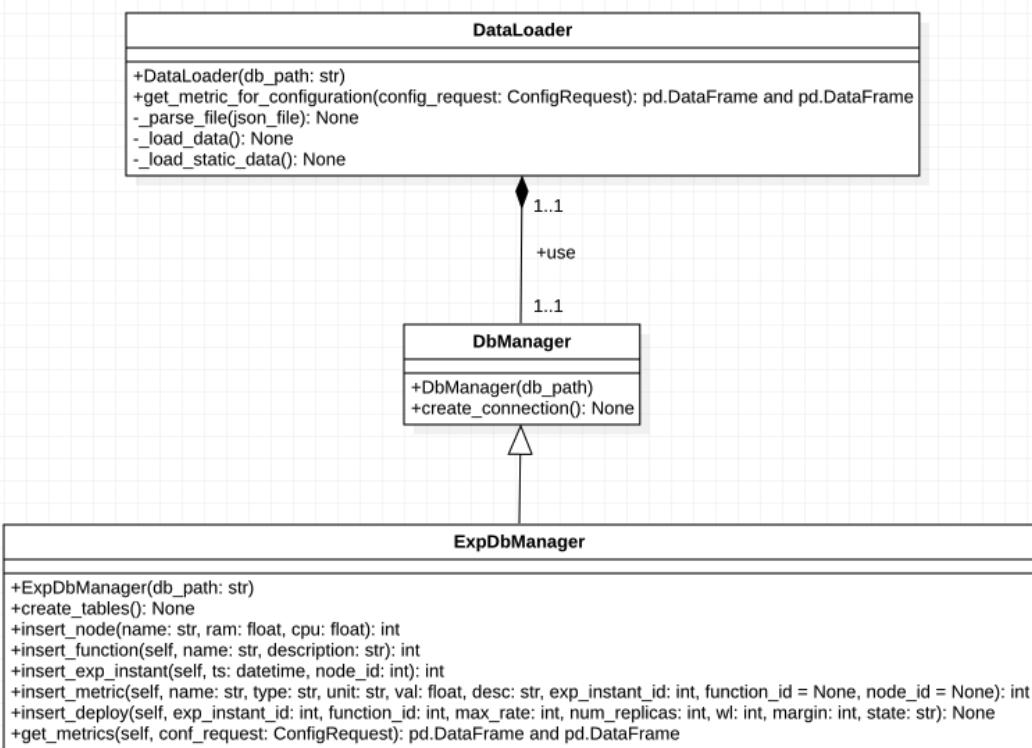


Figura 4.10: Immagine rappresentante la componente *database manager*.

Il *data loader* infatti come prima cosa utilizzerà il metodo `create_tables()` per creare le tabelle necessarie ad ospitare i dati raccolti sperimentalmente, e subito dopo adopererà i metodi di insert per caricare i dati nelle opportune tabelle. Il metodo `get_metrics()`, invece, consente di eseguire il reperimento delle metriche dal database.

se. Le query parametrizzate sono contenute all'interno di una cartella denominata **sql**, nel package **database manager**, e vengono caricate all'occorrenza dalla classe *ExpDbManager* che ne valorizzerà i parametri.

Il database è stato realizzato utilizzando SQLite [38], ed è stato progettato sfruttando la procedura standard di progettazione di database relazionali, ovvero partendo dalla progettazione concettuale, per poi raffinare ed arrivare alla progettazione logica [41].

Il database sottostante deve contenere un'insieme di dati raccolti su diverse tipologie di nodo (sostanzialmente suddivisi in tre tipi, con diverse caratteristiche in termini di CPU e RAM). Su ogni nodo vengono eseguite diverse classi di funzioni. Una funzione in esecuzione su un nodo in un determinato istante ha un certo numero di repliche, ha associato un certo max-rate ed è sottoposta ad un certo workload che ne definisce lo stato (sovraffollato o non sovraffollato). Inoltre, ogni funzione non ancora saturata ha un margine di richieste che può ancora servire. Durante i diversi istanti di simulazione per queste funzioni in esecuzione vengono raccolte delle metriche che possono essere utili agli algoritmi di bilanciamento del carico, come ad esempio l'utilizzo di RAM e CPU di ogni funzione, il tempo medio di esecuzione, ecc. Vengono inoltre raccolte anche altre metriche a livello di nodo, come ad esempio l'utilizzo di CPU e di RAM complessivi del nodo di interesse. Tutti questi concetti sono stati realizzati nel **diagramma E-R** (Entità-Relazioni) riportato in Figura 4.11.

Successivamente, tale diagramma è stato ristrutturato e raffinato al fine di eseguire una progettazione logica efficace. L'unica operazione eseguita è stata quella dell'eliminazione della gerarchia **Metric**. Per eliminare tale gerarchia è stato eseguito un accorpamento verso l'alto visto che le entità figlie non possedevano attributi e dal momento che la maggior parte degli accessi verrà eseguita dall'entità **Experiment Instant**, navigando la relazione **gather**, al fine di reperire tutte le metriche relative a quell'istante dell'esperimento. Portando le relazioni delle entità figlie al padre è stato necessario renderle opzionali (cardinalità minima uguale a zero). Il risultato di questa operazione è possibile vederlo in Figura 4.12.

Infine, una volta fatta questa operazione, è stato possibile passare alla progettazione

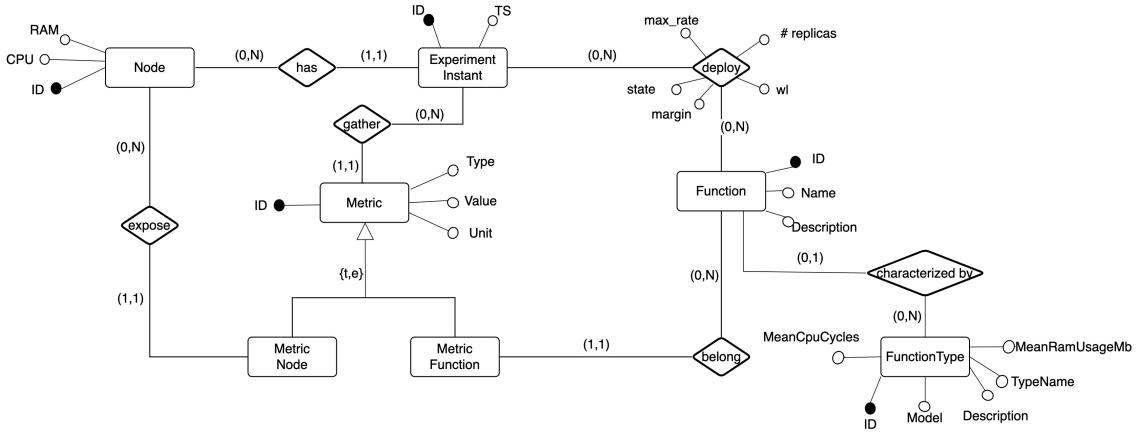


Figura 4.11: Diagramma ER del database utilizzato in questo lavoro di tesi.

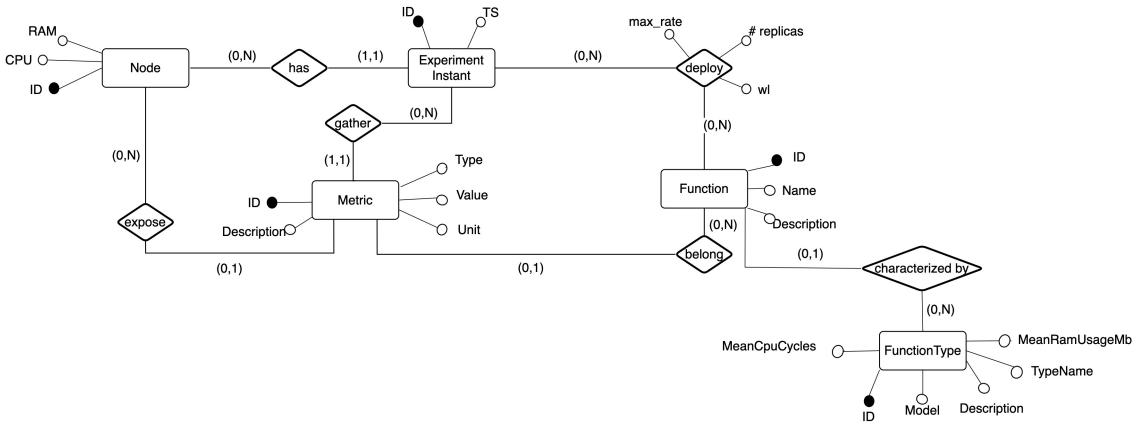


Figura 4.12: Diagramma ER ristrutturato del database utilizzato.

logica. Eseguendo la progettazione logica la relazione multi-a-molti **deploy** è stata trasformata in una tabella di pivot. Le relazioni risultanti sono quindi le seguenti:

- NODE (ID, Name, Ram, Cpu)
- FUNCTION\_TYPE (ID, Model, TypeName, Description, MeanCpuCycles, MeanRamUsageMb)
- FUNCTION (ID, Name, Description, FunctionTypeID\*)
  - FOREIGN KEY: FunctionTypeID REFERENCES FUNCTION\_TYPE
- EXPERIMENT\_INSTANT (ID, Timestamp, NodeID)

- FOREIGN KEY: NodeID **REFERENCES** NODE
- METRIC (ID, Type, Value, Unit, Description, ExpInstantID, NodeID\*, FunctionID\*)
  - FOREIGN KEY: ExpInstantID **REFERENCES** EXPERIMENT\_INSTANT
  - FOREIGN KEY: NodeID **REFERENCES** NODE
  - FOREIGN KEY: FunctionID **REFERENCES** FUNCTIONS
- DEPLOY (ExpInstantID, FunctionID, MaxRate, NumReplicas, Workload, Margin, State)
  - FOREIGN KEY: ExpInstantID **REFERENCES** EXPERIMENT\_INSTANT
  - FOREIGN KEY: FunctionID **REFERENCES** FUNCTIONS

Una volta descritto come sono stati organizzati i dati è possibile definire come è stato realizzato il reperimento di metriche all'interno del database, partendo dalla richiesta di configurazione inoltrata dal *data loader*. Il metodo *get\_metrics(config\_request)* prende in input come parametro una richiesta di configurazione, che contiene le informazioni elencate nella Sezione 4.3.1. Tale parametro è caratterizzato dal tipo di nodo e, per ogni funzione, dal nome, dal numero di repliche e dall'invocation rate ad essa associato. Le operazioni che vengono eseguite per il reperimento delle metriche sono le seguenti:

1. **Reperimento id degli esperimenti.** Viene interrogato il database per ottenere, per ogni funzione, gli id degli **Experiment Instant** in cui essa è in esecuzione.
2. **Reperimento id degli esperimenti con una determinata configurazione.** Sulla base della *config\_request* viene interrogato il database per ottenere gli id degli esperimenti (o meglio degli **Experiment Instant**) in cui sono in esecuzione quelle determinate funzioni, con quel numero di repliche e sottoposte a quel determinato workload.

3. **Eliminazione id relativi ad esperimenti con più funzioni in esecuzione di quelle richieste.** Il problema qui evidenziato è che la query eseguita allo step precedente comprende tutti gli id degli **Experiment Instant** che contengono le funzioni richieste, ma può comprendere anche esperimenti che hanno altre funzioni oltre a quelle comprese nella *config\_request*. Sfruttando la lista reperita allo step 1, vengono esclusi dagli id estrapolati dallo step 2 tutti quelli relativi a funzioni non comprese nella *config\_request*.
4. **Selezione delle metriche per gli id degli esperimenti ottenuti.** Sulla base degli id ottenuti dal processamento precedente, vengono reperite le metriche relativi ad essi. Dal momento che ci possono essere più esperimenti che condividono la medesima configurazione in termini di funzioni in esecuzione, numero di repliche e carico, allora di questi verrà eseguita la media dei valori associati alle metriche stesse.
5. **Separazione delle metriche in due DataFrame.** Le metriche, in base al tipo, ovvero se sono relative al nodo (Type=“node”) o se sono riferite alla funzione (Type=“func”) vengono divise in due DataFrame Pandas, per poi essere ritornati al *data loader*.

## 4.5 Analyzer

Una volta terminata la simulazione vera e propria e prodotti gli artefatti necessari alla fase di analisi, entrerà in gioco l’*analyzer* per calcolare gli indici utili al confronto delle diverse strategie implementate.

Gli indici scelti per eseguire il confronto delle diverse strategie sono i seguenti:

- **Success Rate.** Percentuale di successo delle richieste inoltrate al sistema.
- **Success Rate durante il periodo di stress.** Rateo di successo durante i minuti centrali dell’esperimento, ovvero quando il carico, in termini di invocation

rate, è pressoché costante e non affronta il periodo di salita iniziale e discesa finale.

- **Reject Number.** Numero totale di richieste rigettate.

Per ogni strategia utilizzata e per ogni step di simulazione vengono caricati i file csv prodotti dal *simulator* relativi ai max-rate e agli invocation rate. Inoltre, per ogni funzione, vengono anche caricate le tabelle delle richieste inoltrate. Per ogni strategia, per ogni step/minuto di simulazione e per ogni funzione è quindi necessario calcolare quelli che sono gli indici sopra riportati. Il **Success rate**, per una specifica funzione, è possibile calcolarlo come mostrato in Figura 4.13, seguendo i seguenti step:

1. Eseguendo la somma per colonna delle tabella delle richieste inoltrate si ottiene il numero di richieste ricevute da ogni nodo al termine di ogni step di simulazione (**total\_invoc\_rate**).
2. Viene comparato il valore ottenuto dallo step precedente con il corrispondente max-rate per quel nodo e quella determinata funzione (in questo esempio **funca**). A questo punto si possono verificare due situazioni:

- Se il numero totale di invocazioni, calcolato nello step 1, è maggiore o uguale al max-rate allora si somma il max-rate al **success\_rate** →  $\text{success\_rate} += \text{max-rate}$ .
- Se il numero totale di invocazioni, calcolato nello step 1, è minore del max-rate allora si somma il valore ottenuto precedentemente al **success\_rate** →  $\text{success\_rate} += \text{total\_invoc\_rate}$ .

Questo si traduce nel fatto che un nodo non può gestire più di max-rate richieste per quella funzione e che quindi eventuali richieste in eccesso verranno scartate.

3. Viene divisa la somma eseguita allo step precedente (il **success\_rate**) per la somma della colonna relativa a quel nodo e a quella funzione (sempre **funca** nell'esempio) della tabella dell'invocation rate.

The diagram illustrates the calculation of the success rate metric. It shows three tables: 'FORWARDED REQUESTS FUNCA', 'MAX RATES', and 'INVOCATION RATES'. Arrows point from specific values in the first two tables to the corresponding columns in the third table.

FORWARDED REQUESTS FUNCA			
	node_0	node_1	node_2
node_0	30	0	9
node_1	0	100	77
node_2	0	0	9

MAX RATES			
	funcfa	qrcode	ocr
node_0	30	4	1
node_1	100	10	3
node_2	120	20	5

INVOCATION RATES			
	funcfa	qrcode	ocr
node_0	39	4	0
node_1	177	0	9
node_2	9	31	0

Annotations:

- 1) Total invocations: 30    100    95
- 2) Compare with max\_rates:    30    100    120
- 3) Success rate:  $(30+100+95) / (39+177+9) = 225 / 225 = 1$

Figura 4.13: Esempio di calcolo della metrica success rate.

Sulla base di questo indice, espresso in valore percentuale, compreso tra zero e uno, viene calcolato il **Reject Rate** ( $\text{reject\_rate} = 1 - \text{success\_rate}$ ). Oltre a questo viene calcolato anche il **Reject Number**, contando lo scarto tra il numero totale di invocazioni calcolato nello step precedente (**total\_invoc\_rate**) ed il **max\_rate** ( $\text{reject\_num} = \text{tot\_invoc\_rate} - \text{success\_rate}$ ). Dal momento che però questo valore è calcolato istantaneamente, ed è relativo solo allo step/minuto di simulazione attuale, deve essere moltiplicato per 60, per ottenere il **Reject Number** per il minuto appena simulato.

Questi indici vengono calcolati per ogni strategia e per ogni step/minuto di simulazione e poi di essi vengono calcolati: **media**, **varianza**, **mediana** e **percentile al 90%**. Queste operazioni vengono calcolate per ognuno dei tre indici sopra descritti, unica eccezione viene fatta dal **Reject Number** che al posto della media ne viene calcolata la somma.

Il numero totale di indici calcolati, per ogni strategia, è quindi pari a 12. Questi vengono tutti raccolti all'interno di un file csv, che rappresenta il primo artefatto prodotto dall'*analyzer* (vedi Figura 4.14).

	Mean success rate	Success rate variance	Success rate median	Success rate 90% percentile	Mean success rate (stress period)	Success rate variance (stress period)	Success rate median stress period	Success rate 90% percentile stress period	Tot. rejected requests	Reject num variance	Reject num median	Reject num 90% percentile
base_strategy	66.056690244690	728.307618731563	68.57142857142857	100.0	53.20133243422578	396.57201280292027	64.17910447781194	68.57142857142857	45600.0	10515640.8162053	660.0	8640.0
random_strategy	84.65122347079179	222.004147061892	88.57142857142857	100.0	78.517128695116	179.71823905426305	79.6099517412030	92.0	20680.0	4889191.836734994	240.0	5160.0
empirical_strategy	92.2220302477870	96.962860268156	100.0	100.0	89.67109154687392	105.2961706204483	94.0298507462666	100.0	9240.0	566142.8571428572	0.0	1140.0
dflas_static_strategy	93.27715166648509	98.2588149172350	100.0	100.0	90.589012319356	112.494877773956	94.5273631840796	100.0	6240.0	235020.4081620538	0.0	1320.0

Figura 4.14: Esempio di un file csv esportato dall'*analyzer*, contenente tutti gli indici calcolati.

Infine, per ogni funzione in esecuzione sul sistema, viene anche esportato un grafico che rappresenta l'andamento del **Success rate** durante gli step di simulazione, mettendo a confronto le varie strategie sullo stesso grafico (vedi Figura 4.15).

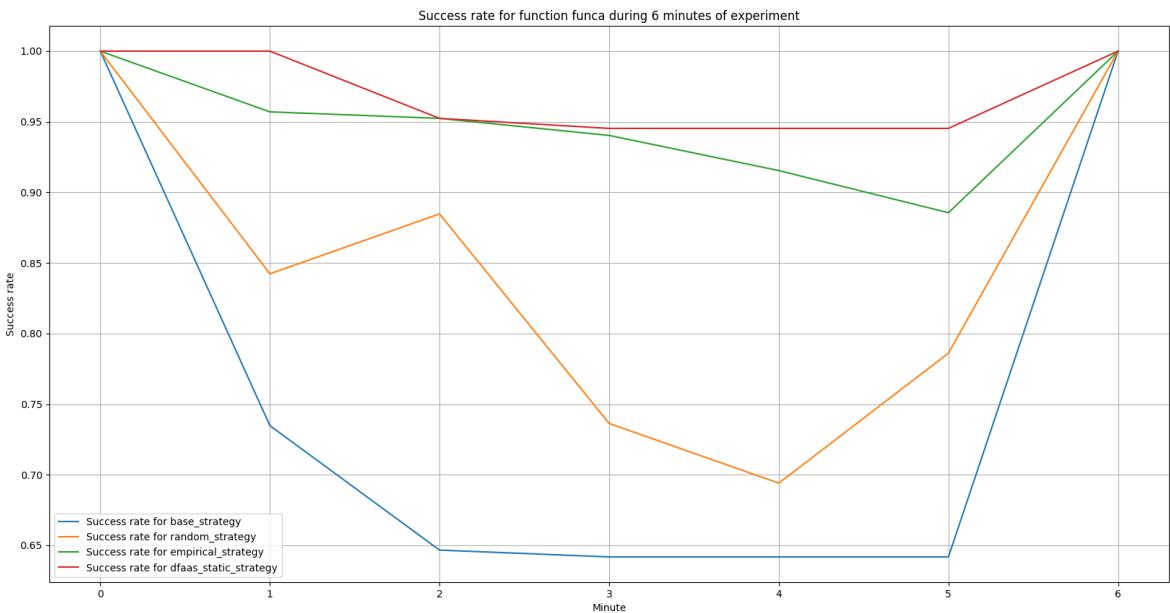


Figura 4.15: Esempio di grafico di confronto del success rate per la funzione funca.

## 4.6 Simulation Controller

Questa componente funge da coordinatore ed entry point per l'applicativo del simulatore. Innanzitutto può ricevere in input un file di istanza, in formato json, da linea di comando. Questo serve per poter riprodurre nuovamente esperimenti partendo da un'istanza precedentemente creata.

Questa componente funge da coordinatore in quanto si occupa di lanciare l'esecuzione di tutte le altre componenti in maniera sequenziale. Il comportamento del *simulation controller* si realizza in cinque fasi:

1. **Generazione dell'istanza.** In questa prima fase viene richiamato l'*instance generator* e gli vengono passati i parametri forniti da CLI da parte dell'utente per generare l'istanza in formato json. Questa prima fase è opzionale, in quanto il *simulation controller* si occuperà di eseguirla se non viene passato nessun file di istanza in input, altrimenti viene saltata. Questa fase viene eseguita una sola volta, in quanto l'istanza da simulare è una sola. Le fasi successive, invece, verranno eseguite più volte, generando un seed per la riproducibilità dei risultati ad ogni iterazione.

2. **Simulazione dell'istanza.** Viene eseguito il *simulator* per simulare l'istanza appena generata o fornita in input. Questa fase viene eseguita più volte.
3. **Analisi dell'output.** In questa fase viene invocato l'*analyzer* per eseguire l'analisi dei dati appena generati dalla fase precedente. Questa fase viene eseguita più volte.
4. **Caricamento dei file contenenti le metriche di comparazione.** In questa fase viene caricato il file csv contenente gli indici di comparazione prodotti dall'*analyzer* nella fase precedente, per raggrupparle in un nuovo DataFrame insieme alle altre simulazioni dell'istanza. Questa fase viene eseguita più volte.
5. **Esportazione del DataFrame di riepilogo delle simulazioni.** Siccome vengono eseguiti gli step 2, 3 e 4 più volte, in questa fase viene esportato il DataFrame contenente i dati riassuntivi di tutte le esecuzioni, producendo il primo artefatto prodotto dal *simulation controller*. Questa fase viene eseguita una sola volta.

E' importante notare che la prima fase e l'ultima fase vengono eseguite una sola volta, mentre tutte le altre vengono eseguite più volte (con un numero di default definito nel *Configuration Manager* pari a 5) per garantire validità statistica dei risultati.

Prima dell'inizio della seconda fase viene generato un **seed** casuale ed impostato nelle librerie principali utilizzate. Questo viene fatto in quanto il **seed** è molto utile per la riproducibilità dei risultati, qualora si voglia simulare un esperimento precedentemente eseguito. Il **seed** rappresenta infatti anche uno dei valori che viene riportato nel DataFrame generato dallo Step 5, e ne viene associato uno diverso ad ogni nuova esecuzione della simulazione.

Un esempio di un DataFrame esportato dal *simulation controller* si può vedere in Figura 4.16.

Il secondo artefatto prodotto dal *simulation controller* è un file di testo (in formato txt) che riporta le stesse informazioni riassuntive riportate a console, per una veloce comparazione delle varie strategie (vedi Figura 4.17).

seed	strategy	Mean success rate	Success rate variance	Success rate median	Success rate 90% percentile	Mean success rate (stress period)	Success rate variance (stress period)	Success rate median stress period	Success rate 90% percentile stress period	Tot. rejected requests	Reject num variance	Reject num median	Reject num 90% percentile	
3795	base_strategy	66.290500004469	728.307907193103	68.57142857142857	53.291332432578	396.372013020111	50.571428571194	68.57142857142857	45600.0	10515640.81630533	660.0	8640.0		
3796	empirical_strategy	92.0704020703748	96.49034914764	100.0	92.398617111111	268.462021784099	92.398617111111	92.398617111111	2260.0	2450734.68377501	300.0	5800.0		
3797	empirical_strategy	92.0704020703748	96.49034914764	100.0	92.398617111111	105.20000532071017	92.398617111111	92.398617111111	9120.0	533012.203018327	0.0	1760.0		
3798	dfaas_static_strategy	92.3416100539818	98.10238190367192	100.0	90.5380370752745	111.8921451711104	94.5273631840798	100.0	6420.0	354204.4887959183	0.0	1320.0		
3598	base_strategy	92.636600244498	728.307907193103	68.57142857142857	100.0	53.291332432578	396.372013020111	64.5273631840798	68.57142857142857	45600.0	10515640.81630533	660.0	8640.0	
3599	random_strategy	83.7956443208453	365.90918032245	88.57142857142857	100.0	77.26330579807	304.6887417802536	82.8239142857142857	97.9996278037942	19980.0	253823.244897959	240.0	4080.0	
3599	empirical_strategy	92.959548734294	97.3038478504288	100.0	90.143924783017	108.45026708514252	94.5273631840798	100.0	7740.0	362383.874895877	0.0	1320.0		
3599	dfaas_static_strategy	93.1854005707795	98.037650710033	100.0	90.4216700674912	111.0318170703374	94.5273631840798	100.0	6840.0	297453.061224489	0.0	1320.0		
1974	base_strategy	66.360600244498	728.307907193103	68.57142857142857	100.0	53.291332432578	396.372013020111	64.5273631840798	68.57142857142857	45600.0	10515640.81630533	660.0	8640.0	
1974	random_strategy	84.55688716308	340.7898718912	91.5867240474336	100.0	78.3796441205971	343.8763837920196	88.57142857142857	98.871142857142868	19820.0	2862024.48979518	240.0	2460.0	
1974	empirical_strategy	92.837244701833	96.398130067067	100.0	89.972114285714113	106.50391241413	93.6307027372927	100.0	8280.0	416787.346087755	0.0	1560.0		
1974	dfaas_static_strategy	93.22941262307054	86.11520598815364	100.0	90.5211768881131	111.8904573090447	94.5273631840798	100.0	6480.0	256897.9591830743	0.0	1320.0		
1120	base_strategy	64.824021982721	288.6712711148005	68.88990000886899	100.0	53.291332432578	396.372013020111	64.5273631840798	68.57142857142857	45600.0	10515640.81630533	660.0	8640.0	
1120	random_strategy	64.824021982721	288.6712711148005	68.88990000886899	100.0	78.7870770501995	273.8130131420298	85.7142857142857	90.0	20040.0	239308.775100400	300.0	2460.0	
1120	empirical_strategy	92.8591185243996	96.34770271402144	100.0	100.0	100.0077665041457	106.19116154405314	94.5273631840798	100.0	8040.0	370448.916914384	0.0	1320.0	
1120	dfaas_static_strategy	93.2771516666596	98.20687149172335	100.0	100.0	90.5890123193056	112.244687739355	94.5273631840798	100.0	6240.0	230202.49914320530	0.0	1320.0	
2320	base_strategy	66.360600244498	728.307907193103	68.57142857142857	100.0	53.291332432578	396.372013020111	64.5273631840798	68.57142857142857	45600.0	10515640.81630533	660.0	8640.0	
2320	random_strategy	84.6512347037078	222.4041740718592	88.57142857142857	100.0	78.5117338693169	179.7182305403030	78.090591742935	92.0	29820.0	498919.80734994	240.0	5160.0	
2320	empirical_strategy	92.62202802776708	96.3620502606157	100.0	100.0	89.6710915498792	105.26617706024485	94.5273631840798	100.0	9240.0	569142.8571428572	0.0	1440.0	
2320	dfaas_static_strategy	93.2771516666596	98.20687149172335	100.0	100.0	90.5890123193056	112.244687739355	94.5273631840798	100.0	6240.0	230202.49914320530	0.0	1320.0	

Figura 4.16: Esempio del DataFrame esportato dal simulation controller, che contiene tutte le metriche per ogni ri-esecuzione della simulazione. Per ogni singola esecuzione viene riportato il seed con cui essa è stata generata.

```
> Strategy base_strategy
  > Mean success rate: 66.64
  > Success rate variance: 728.37
  > Mean success rate (stress period): 53.29
  > Success rate variance (stress period): 396.37
  > Tot. rejected requests: 45600.00
  > Reject num variance: 10515640.82

> Strategy random_strategy
  > Mean success rate: 84.02
  > Success rate variance: 310.12
  > Mean success rate (stress period): 77.62
  > Success rate variance (stress period): 290.62
  > Tot. rejected requests: 21816.00
  > Reject num variance: 3016140.41

> Strategy empirical_strategy
  > Mean success rate: 92.79
  > Success rate variance: 96.76
  > Mean success rate (stress period): 89.91
  > Success rate variance (stress period): 106.34
  > Tot. rejected requests: 8484.00
  > Reject num variance: 452610.61

> Strategy dfaas_static_strategy
  > Mean success rate: 93.24
  > Success rate variance: 98.17
  > Mean success rate (stress period): 90.53
  > Success rate variance (stress period): 111.82
  > Tot. rejected requests: 6444.00
  > Reject num variance: 255683.27
```

Figura 4.17: File txt contenente le informazioni riassuntive utili alla comparazione dei vari algoritmi.

Infine, tutti gli artefatti intermedi, creati dalle varie componenti, vengono raggruppati in un archivio compresso, in formato zip. Tale archivio avrà come nome il timestamp di inizio della simulazione e sarà salvato in una cartella **archive** al di sotto di quella di **outputs** nel package **simulation**. All'interno dell'archivio zippato sarà possibile trovare i seguenti file:

- Il file json rappresentante l'istanza simulata, esportato dall'*instance generator*.
- Il file png rappresentante la topologia di rete, esportato dall'*instance generator*.
- Il file txt con le informazioni riassuntive generate dal *simualtion controller*.
- Il file csv contenete il DataFrame finale con le informazioni dei vari esperimenti, esportato dal *simulation controller*

- Per ogni iterazione del ciclo simulazione-analisi-raccolta dati (step 2, 3 e 4), eseguita dal *simulation controller*, vengono esportati i seguenti file, raggruppati in diverse cartelle:
  - I file di log degli agenti contenenti i pesi calcolati nei vari step di simulazione sfruttando le varie strategie, esportati dal *simulator* (saranno pari al numero di agenti simulati).
  - Il file csv contenente la comparazione delle varie strategie sulla base degli indici calcolati dall'*analyzer*.
  - I file png rappresentanti i grafici di comparazione del success rate delle diverse strategie durante la simulazione, anche questi esportati dall'*analyzer* (saranno pari al numero di funzioni in esecuzione sul sistema).

## 4.7 Configuration Manager

Questo componente è stato realizzato mediante una classe di utility, che isola tutte quelle che sono le variabili di configurazione del simulatore nella sua interezza, ed è utilizzato da tutte le altre componenti per reperirle.

Questa componente è stata realizzata sfruttando il pattern creazionale **Singleton** [42], ovvero creando una singola istanza di questa classe e fornendone un metodo di accesso globale dall'intero applicativo.

All'interno di questa classe sono presenti tutte le variabili che determinano i percorsi dove andare a salvare i dati e gli output intermedi, i nomi degli indici utilizzati per il confronto, i nomi delle funzioni in esecuzione sul sistema, i nomi dei nodi e delle strategie implementate, ecc. (vedi Figura 4.18).

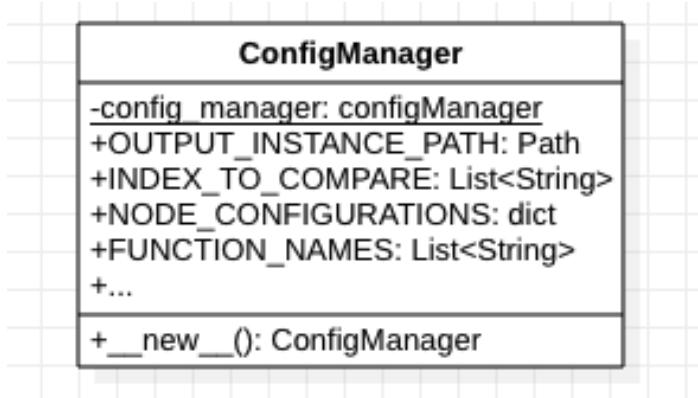


Figura 4.18: Diagramma delle classi del Configuration Manager.

# Capitolo 5

## Algoritmi ed Esperimenti

In questo capitolo viene in primis descritta la metodologia utilizzata per raccogliere i dati usati per poi simulare il funzionamento dei diversi algoritmi (o strategie) di distribuzione del carico. Tali dati sono stati raccolti sfruttando il prototipo del sistema DFaaS [1], descritto in Sezione 2.3.1, che è stato espanso al fine di raccogliere tutte le metriche utili ad un bilanciamento efficacie del carico.

Successivamente, segue una descrizione dettagliata per ogni strategia implementata, ognuna accompagnata da un esperimento che ne dimostri l'effettivo funzionamento.

Queste strategie vengono eseguire dagli agenti per distribuire il carico. Il comportamento di ogni agente è organizzato secondo un ciclo (*loop*) di operazioni, specifiche per ogni strategia, che verrà eseguito periodicamente.

Naturalmente l'accento viene posto sulla strategia **empirica** (Sezione 5.2), ovvero la strategia distribuita ideata inizialmente per il bilanciamento del carico. Tale strategia ha portato alla necessità di raccogliere delle metriche utilizzate per il calcolo dei pesi, e vuole essere validata eseguendo un confronto con altre strategie di **baseline** come la **base strategy** (Sezione 5.3) e la **random strategy**, (Sezione 5.4) o con quella precedentemente implementata su DFaaS, che prende il nome di **DFaaS static strategy** (Sezione 5.5).

Infine, in Sezione 5.6, vengono presentati alcuni esempi di confronto di queste strategie e discusse eventuali considerazioni.

## 5.1 Descrizione del setting sperimentale

Per prima cosa è stato necessario raccogliere i dati utili ad eseguire simulazioni sulla base di dati reali. Tali dati sono stati raccolti sperimentalmente su DFaaS, dotando il sistema delle componenti necessarie al monitoraggio dello stato e predisponendo degli scenari specifici per la raccolta dati.

### 5.1.1 Monitoring ed Exporters

Le metriche raccolte dagli agenti di DFaaS si limitano semplicemente ai max-rate associati alle funzioni in esecuzione, l'invocation rate e l'Average Function Execution Time (AFET), tutte ottenibili interrogando, tramite delle richieste http, il *gateway* di OpenFaaS (la piattaforma di FaaS utilizzata) [43]. Tali metriche sono raccolte da *Prometheus* [44], ma non forniscono nessuna indicazione relativa allo stato di salute del nodo o del cluster su cui la piattaforma è in esecuzione oppure informazioni a livello dei singoli container.

Siccome la strategia empirica escogitata ha la necessità di raccogliere metriche di questo tipo, che sintetizzino lo stato di salute del nodo o delle singole funzioni, è stato necessario espandere quello che è l'insieme di base delle metriche raccolte da *Prometheus* in OpenFaaS. Questo è stato fatto tramite l'utilizzo di **exporters** [45], in modo che le nuove metriche raccolte siano reperibili sempre tramite richieste http al *gateway*. Gli **exporters** che sono stati utilizzati sono due:

- **Node Exporter** [46]. Consente di raccogliere metriche di basso livello relative al nodo su cui l'exporter è in esecuzione, come ad esempio a livello di utilizzo di risorse hardware (es. utilizzo di RAM o CPU), o a livello di sistema operativo.
- **cAdvisor** [47]. Consente di raccogliere metriche a livello dei singoli container, come ad esempio il loro utilizzo di risorse (utilizzo di CPU e RAM), le loro performance, ecc. cAdvisor è utilizzabile tramite la sua interfaccia grafica, ma è stato utilizzato sempre tramite *Prometheus* in questo lavoro di tesi. Inoltre, ha un supporto nativo per i container Docker.

Come è possibile vedere in Figura 5.1, vengono rappresentati *Prometheus* e i due **exporters** utilizzati. Si può notare che *Prometheus* ogni 5 secondi raccoglie quelle che sono le metriche dagli **exporters**, tramite delle richieste http, per poi memorizzarle in un database interno. Il **node exporter** monitora lo stato del nodo, mentre **cAdvisor** lo stato dei container in esecuzione su di esso.

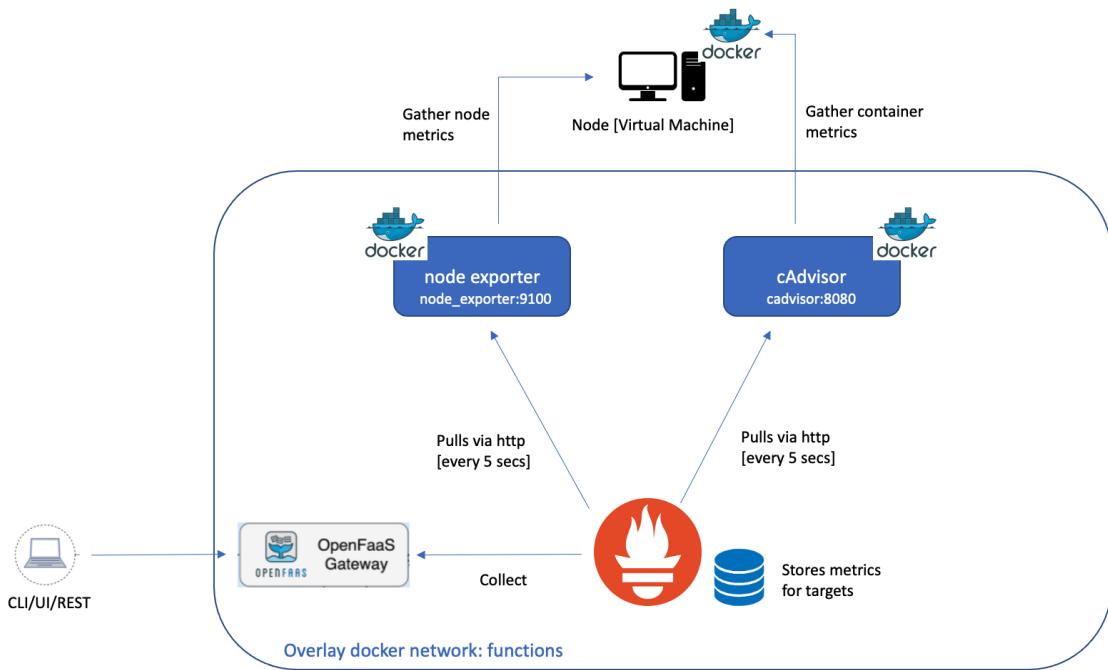


Figura 5.1: Architettura utilizzata per effettuare *monitoring*.

Dal momento che OpenFaaS viene eseguito sul prototipo di DFaaS sfruttando il file di **Docker Compose** [48], e utilizza come orchestratore, nella sua versione attuale, **Docker Swarm** [49], sono stati aggiunti gli exporters ai servizi del file docker compose, tramite sintassi yaml (Yet Another Markup Language), come si può vedere in Figura 5.2. Tutti i servizi all'interno di questo file sono stati inseriti all'interno della stessa rete di overlay, in modo tale che anche gli exporters fossero raggiungibili da *Prometheus*. All'interno di questo file è stato anche cambiato il tempo di mantenimento dei dati prima che vengano cancellati dal database interno di *Prometheus* (**retention policy**) a 72 ore, ovvero tre giorni. E' stato inoltre aggiunto un volume al servizio di *Prometheus* per consentire la condivisione dei dati con l'host.

```

# [NEW] Add node exporter service.
# See: https://github.com/prometheus/node\_exporter
node_exporter:
    image: quay.io/prometheus/node-exporter:latest
    #container_name: node_exporter
    restart: unless-stopped
    networks:
        - functions
    volumes:
        - /home/debian/node_exporter:/host:ro,rslave
    deploy:
        mode: global # Deploying on all nodes (Docker Swarm to be replaced).
        resources:
            limits:
                memory: 200M

# [NEW] Add cAdvisor service.
# Remember that cAdvisor has its own WebUI, bit for this purpose only prom is used.
# See: https://prometheus.io/docs/guides/cadvisor/
cadvisor:
    image: gcr.io/cadvisor/cadvisor
    hostname: '{{.Node.Hostname}}'
    #container_name: cadvisor
    #ports:
    #    - 9400:8080 # 9400 is a random-chosen port; instead, 8080 is default port for cAdvisor.
    restart: unless-stopped
    volumes:
        - /:/rootfs:ro
        - /var/run:/var/run:rw
        - /sys:/sys:ro
        - /var/lib/docker/:/var/lib/docker:ro
        #- /cgroup:/sys/fs/cgroup:ro
        - /dev/disk/:/dev/disk:ro
    networks:
        - functions
    deploy:
        mode: global # Deploying on all nodes (Docker Swarm to be replaced).
        resources:
            limits:
                memory: 128M
            reservations:
                memory: 64M

```

Figura 5.2: Servizi aggiunti all'interno del file di docker compose per eseguire gli exporters.

Infine, per eseguire il deploy automatico degli exporters durante le fasi dell'installazione del prototipo di DFaaS, è stato predisposto un playbook **Ansible** [50]. Ansible è un software open source, sviluppato da RedHat, che consente di automatizzare il

provisioning, la gestione della configurazione, il deployment delle applicazioni, l'orchestrazione e molti altri processi [51]. Il playbook utilizzato comprende le seguenti fasi:

1. Viene caricato su tutti i nodi di DFaaS il nuovo file di configurazione di *Prometheus*.
2. Viene caricato su tutti i nodi di DFaaS il nuovo file di docker compose per lo stack di OpenFaaS.
3. Viene rimosso lo stack di OpenFaaS in esecuzione.
4. Viene eseguito nuovamente lo stack di OpenFaaS basato sul nuovo file di docker compose, ovvero quello contenente anche gli exporters.

Tra le metriche di base già presenti sul *gateway* di OpenFaaS e le metriche raccolte dal **node exporter** e da **cAdvisor**, è stato selezionato il seguente sottoinsieme:

- **Service count.** Numero di repliche in esecuzione per ogni funzione.
- **Invocation rate.** Numero di invocazioni effettuate per ogni funzione nell'ultima finestra temporale (di default un minuto).
- **AFET.** Tempo medio di esecuzione per ogni funzione, espresso in secondi, nell'ultima finestra temporale (di default un minuto).
- **Node CPU usage.** Utilizzo di CPU del nodo, mediato nell'ultima finestra temporale (di default 1 minuto).
- **Node RAM usage.** Utilizzo di RAM del nodo, mediato nell'ultima finestra temporale (di default 1 minuto).
- **Per-function CPU usage.** Utilizzo di CPU per classe di funzione rispetto la CPU disponibile sul nodo; anche questa mediata nell'ultima finestra temporale (di default 1 minuto).

- **Per-function RAM usage.** Utilizzo di RAM per classe di funzione rispetto la RAM disponibile sul nodo; anche questa mediata nell'ultima finestra temporale (di default 1 minuto).

Gli agenti di DFaaS, presenti in ogni nodo, vengono eseguiti di default una volta al minuto, proprio per questo le metriche raccolte vengono mediate sull'ultima finestra temporale passata tra due istanti successivi di monitoring (60 secondi).

Importante sottolineare che tutte queste metriche sono state ottenute eseguendo delle query al *gateway* di OpenFaaS, utilizzando l'API e il linguaggio di query di *Prometheus* [52].

L'agente di DFaaS, scritto in linguaggio Go [53], è stato quindi espanso per poter effettuare queste query in una fase iniziale di monitoring, esportando ad ogni sua ciclica esecuzione (di default una volta al minuto) le metriche raccolte sullo stato del nodo in un file in formato json (un esempio di file viene riportato in Figura 5.3).

```

1  {
2     "input": {
3         "node": "node_3",
4         "func_a_num": 5,
5         "func_a_wl": 82,
6         "qrcode_num": 0,
7         "ocr_num": 0,
8         "qrcode_wl": 0,
9         "ocr_wl": 0
10    },
11    "output": [
12      {
13          "ram_usage": 0.14887352874442017,
14          "cpu_usage": 0.11916580301459934,
15          "functions": [
16              {
17                  "name": "func_a",
18                  "service_count": 5,
19                  "margin": 150,
20                  "invoc_rate": 0,
21                  "afet": 0.001,
22                  "ram_xfunc": 0.0237396889974967,
23                  "cpu_xfunc": 0.000681562687186386
24                  "max_rate": 150,
25                  "state": "Underload",
26                  "prom_invoc_rate": 0
27              }
28          ]
29      },
30      {
31          "ram_usage": 0.1487412994546824,
32          "cpu_usage": 0.12026938597448078,
33          "functions": [
34              {
35                  "name": "func_a",
36                  "service_count": 5,
37                  "margin": 150,
38                  "invoc_rate": 0,
39                  "afet": 0.001,
40                  "ram_xfunc": 0.023742702548301905
41                  "cpu_xfunc": 0,
42                  "max_rate": 150,
43                  "state": "Underload",
44                  "prom_invoc_rate": 0
45              }
46          ]
47      }
48  ],
49  [
50      {
51          "ram_usage": 0.14904473892303993,
52          "cpu_usage": 0.34013636363636335,
53          "functions": [
54              {
55                  "name": "func_a",
56                  "service_count": 5,
57                  "margin": 73,
58                  "invoc_rate": 77,
59                  "afet": 0.011405204592025982,
60                  "ram_xfunc": 0.0237659291162199,
61                  "cpu_xfunc": 0.07627500154795012,
62                  "max_rate": 150,
63                  "state": "Underload",
64                  "prom_invoc_rate": 72.58909890909091
65              }
66          ]
67      },
68      {
69          "ram_usage": 0.14977702276418392,
70          "cpu_usage": 0.39144348079055724,
71          "functions": [
72              {
73                  "name": "func_a",
74                  "service_count": 5,
75                  "margin": 74,
76                  "invoc_rate": 76,
77                  "afet": 0.008125630872268729,
78                  "ram_xfunc": 0.023782595671388247,
79                  "cpu_xfunc": 0.08418661234100563,
80                  "max_rate": 150,
81                  "state": "Underload",
82                  "prom_invoc_rate": 81.912494361751
83              }
84          ]
85      },
86      {
87          "ram_usage": 0.15029278250825973,
88          "cpu_usage": 0.365727272727272729,
89          "functions": [
90              {
91                  "name": "func_a",
92                  "service_count": 5,
93                  "margin": 71,
94                  "invoc_rate": 79,
95                  "afet": 0.01192435100753876,
96                  "ram_xfunc": 0.02380814914788888
97              }
98          ]
99      },
100     {
101         "ram_usage": 0.15029736849518682,
102         "cpu_usage": 0.3479426098670892,
103         "functions": [
104             {
105                 "name": "func_a",
106                 "service_count": 5,
107                 "margin": 72,
108                 "invoc_rate": 78,
109                 "afet": 0.00879232455239312,
110                 "ram_xfunc": 0.02382156663388381
111                 "cpu_xfunc": 0.07971656821652977
112                 "max_rate": 150,
113                 "state": "Underload",
114                 "prom_invoc_rate": 82.4239311850
115             }
116         ]
117     },
118     {
119         "ram_usage": 0.15063995263767394,
120         "cpu_usage": 0.35009090909090909,
121         "functions": [
122             {
123                 "name": "func_a",
124                 "service_count": 5,
125                 "margin": 72,
126                 "invoc_rate": 78,
127                 "afet": 0.0088467678023952542,
128                 "ram_xfunc": 0.0239822399615873,
129                 "cpu_xfunc": 0.102295180993317502
130                 "max_rate": 150,
131                 "state": "Underload",
132                 "prom_invoc_rate": 81.981010101010101
133             }
134         ]
135     }
136  ]
137  [
138      {
139          "ram_usage": 0.15063995263767394,
140          "cpu_usage": 0.35009090909090909,
141          "functions": [
142              {
143                  "name": "func_a",
144                  "service_count": 5,
145                  "margin": 72,
146                  "invoc_rate": 78,
147                  "afet": 0.0088467678023952542,
148                  "ram_xfunc": 0.0239822399615873,
149                  "cpu_xfunc": 0.102295180993317502
150                  "max_rate": 150,
151                  "state": "Underload",
152                  "prom_invoc_rate": 81.981010101010101
153              }
154          ]
155      }
156  ]
157  [
158      {
159          "ram_usage": 0.15063995263767394,
160          "cpu_usage": 0.35009090909090909,
161          "functions": [
162              {
163                  "name": "func_a",
164                  "service_count": 5,
165                  "margin": 72,
166                  "invoc_rate": 78,
167                  "afet": 0.0088467678023952542,
168                  "ram_xfunc": 0.0239822399615873,
169                  "cpu_xfunc": 0.102295180993317502
170                  "max_rate": 150,
171                  "state": "Underload",
172                  "prom_invoc_rate": 81.981010101010101
173              }
174          ]
175      }
176  ]
177  [
178      {
179          "ram_usage": 0.15063995263767394,
180          "cpu_usage": 0.35009090909090909,
181          "functions": [
182              {
183                  "name": "func_a",
184                  "service_count": 5,
185                  "margin": 72,
186                  "invoc_rate": 78,
187                  "afet": 0.0088467678023952542,
188                  "ram_xfunc": 0.0239822399615873,
189                  "cpu_xfunc": 0.102295180993317502
190                  "max_rate": 150,
191                  "state": "Underload",
192                  "prom_invoc_rate": 81.981010101010101
193              }
194          ]
195      }
196  ]
197  [
198      {
199          "ram_usage": 0.15063995263767394,
200          "cpu_usage": 0.35009090909090909,
201          "functions": [
202              {
203                  "name": "func_a",
204                  "service_count": 5,
205                  "margin": 72,
206                  "invoc_rate": 78,
207                  "afet": 0.0088467678023952542,
208                  "ram_xfunc": 0.0239822399615873,
209                  "cpu_xfunc": 0.102295180993317502
210                  "max_rate": 150,
211                  "state": "Underload",
212                  "prom_invoc_rate": 81.981010101010101
213              }
214          ]
215      }
216  ]
217  [
218      {
219          "ram_usage": 0.15063995263767394,
220          "cpu_usage": 0.35009090909090909,
221          "functions": [
222              {
223                  "name": "func_a",
224                  "service_count": 5,
225                  "margin": 72,
226                  "invoc_rate": 78,
227                  "afet": 0.0088467678023952542,
228                  "ram_xfunc": 0.0239822399615873,
229                  "cpu_xfunc": 0.102295180993317502
230                  "max_rate": 150,
231                  "state": "Underload",
232                  "prom_invoc_rate": 81.981010101010101
233              }
234          ]
235      }
236  ]
237  [
238      {
239          "ram_usage": 0.15063995263767394,
240          "cpu_usage": 0.35009090909090909,
241          "functions": [
242              {
243                  "name": "func_a",
244                  "service_count": 5,
245                  "margin": 72,
246                  "invoc_rate": 78,
247                  "afet": 0.0088467678023952542,
248                  "ram_xfunc": 0.0239822399615873,
249                  "cpu_xfunc": 0.102295180993317502
250                  "max_rate": 150,
251                  "state": "Underload",
252                  "prom_invoc_rate": 81.981010101010101
253              }
254          ]
255      }
256  ]
257  [
258      {
259          "ram_usage": 0.15063995263767394,
260          "cpu_usage": 0.35009090909090909,
261          "functions": [
262              {
263                  "name": "func_a",
264                  "service_count": 5,
265                  "margin": 72,
266                  "invoc_rate": 78,
267                  "afet": 0.0088467678023952542,
268                  "ram_xfunc": 0.0239822399615873,
269                  "cpu_xfunc": 0.102295180993317502
270                  "max_rate": 150,
271                  "state": "Underload",
272                  "prom_invoc_rate": 81.981010101010101
273              }
274          ]
275      }
276  ]
277  [
278      {
279          "ram_usage": 0.15063995263767394,
280          "cpu_usage": 0.35009090909090909,
281          "functions": [
282              {
283                  "name": "func_a",
284                  "service_count": 5,
285                  "margin": 72,
286                  "invoc_rate": 78,
287                  "afet": 0.0088467678023952542,
288                  "ram_xfunc": 0.0239822399615873,
289                  "cpu_xfunc": 0.102295180993317502
290                  "max_rate": 150,
291                  "state": "Underload",
292                  "prom_invoc_rate": 81.981010101010101
293              }
294          ]
295      }
296  ]
297  [
298      {
299          "ram_usage": 0.15063995263767394,
300          "cpu_usage": 0.35009090909090909,
301          "functions": [
302              {
303                  "name": "func_a",
304                  "service_count": 5,
305                  "margin": 72,
306                  "invoc_rate": 78,
307                  "afet": 0.0088467678023952542,
308                  "ram_xfunc": 0.0239822399615873,
309                  "cpu_xfunc": 0.102295180993317502
310                  "max_rate": 150,
311                  "state": "Underload",
312                  "prom_invoc_rate": 81.981010101010101
313              }
314          ]
315      }
316  ]
317  [
318      {
319          "ram_usage": 0.15063995263767394,
320          "cpu_usage": 0.35009090909090909,
321          "functions": [
322              {
323                  "name": "func_a",
324                  "service_count": 5,
325                  "margin": 72,
326                  "invoc_rate": 78,
327                  "afet": 0.0088467678023952542,
328                  "ram_xfunc": 0.0239822399615873,
329                  "cpu_xfunc": 0.102295180993317502
330                  "max_rate": 150,
331                  "state": "Underload",
332                  "prom_invoc_rate": 81.981010101010101
333              }
334          ]
335      }
336  ]
337  [
338      {
339          "ram_usage": 0.15063995263767394,
340          "cpu_usage": 0.35009090909090909,
341          "functions": [
342              {
343                  "name": "func_a",
344                  "service_count": 5,
345                  "margin": 72,
346                  "invoc_rate": 78,
347                  "afet": 0.0088467678023952542,
348                  "ram_xfunc": 0.0239822399615873,
349                  "cpu_xfunc": 0.102295180993317502
350                  "max_rate": 150,
351                  "state": "Underload",
352                  "prom_invoc_rate": 81.981010101010101
353              }
354          ]
355      }
356  ]
357  [
358      {
359          "ram_usage": 0.15063995263767394,
360          "cpu_usage": 0.35009090909090909,
361          "functions": [
362              {
363                  "name": "func_a",
364                  "service_count": 5,
365                  "margin": 72,
366                  "invoc_rate": 78,
367                  "afet": 0.0088467678023952542,
368                  "ram_xfunc": 0.0239822399615873,
369                  "cpu_xfunc": 0.102295180993317502
370                  "max_rate": 150,
371                  "state": "Underload",
372                  "prom_invoc_rate": 81.981010101010101
373              }
374          ]
375      }
376  ]
377  [
378      {
379          "ram_usage": 0.15063995263767394,
380          "cpu_usage": 0.35009090909090909,
381          "functions": [
382              {
383                  "name": "func_a",
384                  "service_count": 5,
385                  "margin": 72,
386                  "invoc_rate": 78,
387                  "afet": 0.0088467678023952542,
388                  "ram_xfunc": 0.0239822399615873,
389                  "cpu_xfunc": 0.102295180993317502
390                  "max_rate": 150,
391                  "state": "Underload",
392                  "prom_invoc_rate": 81.981010101010101
393              }
394          ]
395      }
396  ]
397  [
398      {
399          "ram_usage": 0.15063995263767394,
400          "cpu_usage": 0.35009090909090909,
401          "functions": [
402              {
403                  "name": "func_a",
404                  "service_count": 5,
405                  "margin": 72,
406                  "invoc_rate": 78,
407                  "afet": 0.0088467678023952542,
408                  "ram_xfunc": 0.0239822399615873,
409                  "cpu_xfunc": 0.102295180993317502
410                  "max_rate": 150,
411                  "state": "Underload",
412                  "prom_invoc_rate": 81.981010101010101
413              }
414          ]
415      }
416  ]
417  [
418      {
419          "ram_usage": 0.15063995263767394,
420          "cpu_usage": 0.35009090909090909,
421          "functions": [
422              {
423                  "name": "func_a",
424                  "service_count": 5,
425                  "margin": 72,
426                  "invoc_rate": 78,
427                  "afet": 0.0088467678023952542,
428                  "ram_xfunc": 0.0239822399615873,
429                  "cpu_xfunc": 0.102295180993317502
430                  "max_rate": 150,
431                  "state": "Underload",
432                  "prom_invoc_rate": 81.981010101010101
433              }
434          ]
435      }
436  ]
437  [
438      {
439          "ram_usage": 0.15063995263767394,
440          "cpu_usage": 0.35009090909090909,
441          "functions": [
442              {
443                  "name": "func_a",
444                  "service_count": 5,
445                  "margin": 72,
446                  "invoc_rate": 78,
447                  "afet": 0.0088467678023952542,
448                  "ram_xfunc": 0.0239822399615873,
449                  "cpu_xfunc": 0.102295180993317502
450                  "max_rate": 150,
451                  "state": "Underload",
452                  "prom_invoc_rate": 81.981010101010101
453              }
454          ]
455      }
456  ]
457  [
458      {
459          "ram_usage": 0.15063995263767394,
460          "cpu_usage": 0.35009090909090909,
461          "functions": [
462              {
463                  "name": "func_a",
464                  "service_count": 5,
465                  "margin": 72,
466                  "invoc_rate": 78,
467                  "afet": 0.0088467678023952542,
468                  "ram_xfunc": 0.0239822399615873,
469                  "cpu_xfunc": 0.102295180993317502
470                  "max_rate": 150,
471                  "state": "Underload",
472                  "prom_invoc_rate": 81.981010101010101
473              }
474          ]
475      }
476  ]
477  [
478      {
479          "ram_usage": 0.15063995263767394,
480          "cpu_usage": 0.35009090909090909,
481          "functions": [
482              {
483                  "name": "func_a",
484                  "service_count": 5,
485                  "margin": 72,
486                  "invoc_rate": 78,
487                  "afet": 0.0088467678023952542,
488                  "ram_xfunc": 0.0239822399615873,
489                  "cpu_xfunc": 0.102295180993317502
490                  "max_rate": 150,
491                  "state": "Underload",
492                  "prom_invoc_rate": 81.981010101010101
493              }
494          ]
495      }
496  ]
497  [
498      {
499          "ram_usage": 0.15063995263767394,
500          "cpu_usage": 0.35009090909090909,
501          "functions": [
502              {
503                  "name": "func_a",
504                  "service_count": 5,
505                  "margin": 72,
506                  "invoc_rate": 78,
507                  "afet": 0.0088467678023952542,
508                  "ram_xfunc": 0.0239822399615873,
509                  "cpu_xfunc": 0.102295180993317502
510                  "max_rate": 150,
511                  "state": "Underload",
512                  "prom_invoc_rate": 81.981010101010101
513              }
514          ]
515      }
516  ]
517  [
518      {
519          "ram_usage": 0.15063995263767394,
520          "cpu_usage": 0.35009090909090909,
521          "functions": [
522              {
523                  "name": "func_a",
524                  "service_count": 5,
525                  "margin": 72,
526                  "invoc_rate": 78,
527                  "afet": 0.0088467678023952542,
528                  "ram_xfunc": 0.0239822399615873,
529                  "cpu_xfunc": 0.102295180993317502
530                  "max_rate": 150,
531                  "state": "Underload",
532                  "prom_invoc_rate": 81.981010101010101
533              }
534          ]
535      }
536  ]
537  [
538      {
539          "ram_usage": 0.15063995263767394,
540          "cpu_usage": 0.35009090909090909,
541          "functions": [
542              {
543                  "name": "func_a",
544                  "service_count": 5,
545                  "margin": 72,
546                  "invoc_rate": 78,
547                  "afet": 0.0088467678023952542,
548                  "ram_xfunc": 0.0239822399615873,
549                  "cpu_xfunc": 0.102295180993317502
550                  "max_rate": 150,
551                  "state": "Underload",
552                  "prom_invoc_rate": 81.981010101010101
553              }
554          ]
555      }
556  ]
557  [
558      {
559          "ram_usage": 0.15063995263767394,
560          "cpu_usage": 0.35009090909090909,
561          "functions": [
562              {
563                  "name": "func_a",
564                  "service_count": 5,
565                  "margin": 72,
566                  "invoc_rate": 78,
567                  "afet": 0.0088467678023952542,
568                  "ram_xfunc": 0.0239822399615873,
569                  "cpu_xfunc": 0.102295180993317502
570                  "max_rate": 150,
571                  "state": "Underload",
572                  "prom_invoc_rate": 81.981010101010101
573              }
574          ]
575      }
576  ]
577  [
578      {
579          "ram_usage": 0.15063995263767394,
580          "cpu_usage": 0.35009090909090909,
581          "functions": [
582              {
583                  "name": "func_a",
584                  "service_count": 5,
585                  "margin": 72,
586                  "invoc_rate": 78,
587                  "afet": 0.0088467678023952542,
588                  "ram_xfunc": 0.0239822399615873,
589                  "cpu_xfunc": 0.102295180993317502
590                  "max_rate": 150,
591                  "state": "Underload",
592                  "prom_invoc_rate": 81.981010101010101
593              }
594          ]
595      }
596  ]
597  [
598      {
599          "ram_usage": 0.15063995263767394,
600          "cpu_usage": 0.35009090909090909,
601          "functions": [
602              {
603                  "name": "func_a",
604                  "service_count": 5,
605                  "margin": 72,
606                  "invoc_rate": 78,
607                  "afet": 0.0088467678023952542,
608                  "ram_xfunc": 0.0239822399615873,
609                  "cpu_xfunc": 0.102295180993317502
610                  "max_rate": 150,
611                  "state": "Underload",
612                  "prom_invoc_rate": 81.981010101010101
613              }
614          ]
615      }
616  ]
617  [
618      {
619          "ram_usage": 0.15063995263767394,
620          "cpu_usage": 0.35009090909090909,
621          "functions": [
622              {
623                  "name": "func_a",
624                  "service_count": 5,
625                  "margin": 72,
626                  "invoc_rate": 78,
627                  "afet": 0.0088467678023952542,
628                  "ram_xfunc": 0.0239822399615873,
629                  "cpu_xfunc": 0.102295180993317502
630                  "max_rate": 150,
631                  "state": "Underload",
632                  "prom_invoc_rate": 81.981010101010101
633              }
634          ]
635      }
636  ]
637  [
638      {
639          "ram_usage": 0.15063995263767394,
640          "cpu_usage": 0.35009090909090909,
641          "functions": [
642              {
643                  "name": "func_a",
644                  "service_count": 5,
645                  "margin": 72,
646                  "invoc_rate": 78,
647                  "afet": 0.0088467678023952542,
648                  "ram_xfunc": 0.0239822399615873,
649                  "cpu_xfunc": 0.102295180993317502
650                  "max_rate": 150,
651                  "state": "Underload",
652                  "prom_invoc_rate": 81.981010101010101
653              }
654          ]
655      }
656  ]
657  [
658      {
659          "ram_usage": 0.15063995263767394,
660          "cpu_usage": 0.35009090909090909,
661          "functions": [
662              {
663                  "name": "func_a",
664                  "service_count": 5,
665                  "margin": 72,
666                  "invoc_rate": 78,
667                  "afet": 0.0088467678023952542,
668                  "ram_xfunc": 0.0239822399615873,
669                  "cpu_xfunc": 0.102295180993317502
670                  "max_rate": 150,
671                  "state": "Underload",
672                  "prom_invoc_rate": 81.981010101010101
673              }
674          ]
675      }
676  ]
677  [
678      {
679          "ram_usage": 0.15063995263767394,
680          "cpu_usage": 0.35009090909090909,
681          "functions": [
682              {
683                  "name": "func_a",
684                  "service_count": 5,
685                  "margin": 72,
686                  "invoc_rate": 78,
687                  "afet": 0.0088467678023952542,
688                  "ram_xfunc": 0.0239822399615873,
689                  "cpu_xfunc": 0.102295180993317502
690                  "max_rate": 150,
691                  "state": "Underload",
692                  "prom_invoc_rate": 81.981010101010101
693              }
694          ]
695      }
696  ]
697  [
698      {
699          "ram_usage": 0.15063995263767394,
700          "cpu_usage": 0.35009090909090909,
701          "functions": [
702              {
703                  "name": "func_a",
704                  "service_count": 5,
705                  "margin": 72,
706                  "invoc_rate": 78,
707                  "afet": 0.0088467678023952542,
708                  "ram_xfunc": 0.0239822399615873,
709                  "cpu_xfunc": 0.102295180993317502
710                  "max_rate": 150,
711                  "state": "Underload",
712                  "prom_invoc_rate": 81.981010101010101
713              }
714          ]
715      }
716  ]
717  [
718      {
719          "ram_usage": 0.15063995263767394,
720          "cpu_usage": 0.35009090909090909,
721          "functions": [
722              {
723                  "name": "func_a",
724                  "service_count": 5,
725                  "margin": 72,
726                  "invoc_rate": 78,
727                  "afet": 0.0088467678023952542,
728                  "ram_xfunc": 0.0239822399615873,
729                  "cpu_xfunc": 0.102295180993317502
730                  "max_rate": 150,
731                  "state": "Underload",
732                  "prom_invoc_rate": 81.981010101010101
733              }
734          ]
735      }
736  ]
737  [
738      {
739          "ram_usage": 0.15063995263767394,
740          "cpu_usage": 0.35009090909090909,
741          "functions": [
742              {
743                  "name": "func_a",
744                  "service_count": 5,
745                  "margin": 72,
746                  "invoc_rate": 78,
747                  "afet": 0.0088467678023952542,
748                  "ram_xfunc
```

### 5.1.2 Setup sperimentale per la raccolta dati

Una volta dotato il prototipo di DFaaS degli exporter necessari alla raccolta delle metriche, e consentito all’agente di esportare i dati raccolti, è stato possibile proseguire con l’ideare una metodologia sperimentale di raccolta dati.

Per ottenere dati che fossero utili ad eseguire la simulazione, si è deciso di raccogliere informazioni riguardo lo stato di un nodo con diverse situazioni di deployment, in termini di numero di repliche delle funzioni, e sottoponendolo a diversi quantitativi di carico su di esse.

E’ bene sottolineare che questo lavoro di tesi si colloca in un contesto fortemente dinamico ed eterogeneo come quello dell’**edge computing**, dove i nodi perimetrali della rete possiedono caratteristiche diverse gli uni dagli altri, in termini di risorse computazionali, capacità di archiviazione, banda di comunicazione, ecc. Partendo da questo presupposto si è deciso di eseguire la raccolta dati per tre tipologie di nodi differenti, con diverse disponibilità di risorse. Tutto questo nei limiti imposti dalla macchina su cui il prototipo è stato fatto girare e nelle limitazioni di configurazione imposte da VirtualBox, ovvero il software utilizzato per implementare i nodi del prototipo come macchine virtuali.

Di seguito vengono riassunte in Tabella 5.1 le risorse dei tre nodi utilizzati per la raccolta dati.

Tabella 5.1: Tabella rappresentante le caratteristiche dei nodi in termini di risorse disponibili: CPU e RAM.

	<b>RAM (Gb)</b>	<b># of (virtual) CPUs</b>
<b>node of type 1</b>	2	1
<b>node of type 2</b>	4	2
<b>node of type 3</b>	6	4

Successivamente sono state scelte tre tipologie di funzioni, con caratteristiche e consumi di risorse differenti:

- **funcA**. Funzione molto semplice che ritorna una stringa di “Hello Word”, contenente il nome del nodo che ha processato la richiesta. E’ stata scelta in quanto

già presente tra le funzioni di test utilizzate su DFaaS e permette di simulare l'esecuzione di un task molto semplice.

- **qrcode** [54]. Funzione che utilizza una libreria in Go per generare un QR Code a partire da una stringa (es. un URL). Questa funzione è già disponibile nel **Function Store** [55] di OpenFaaS, che racchiude una serie di funzioni testate dalla comunità e ne facilita il deployment. Questa funzione esegue una computazione più dispendiosa della precedente.
- **ocr** [56]. Funzione di Ocr (Optical Character Recognition) di Tesseract, un engine che esegue questo compito per diversi sistemi operativi. Questa funzione riconosce quindi i caratteri di un'immagine passata in input. Anche questa funzione è disponibile sul Function Store di OpenFaaS. Esegue una computazione più dispendiosa delle precedenti, in particolare in termini di CPU.

Una volta definite le caratteristiche dei nodi è stato necessario studiare con maggior attenzione come organizzare i deployment delle funzioni sopra elencate sui vari nodi per la raccolta dati. Queste considerazioni sono state fatte sulla base dalle caratteristiche dei vari nodi e alle risorse consumate dalle invocazioni delle varie funzioni.

E' stato necessario effettuare uno studio più approfondito del numero di repliche massimo eseguibili sui vari nodi e del numero massimo di invocazioni per ognuna di esse (si parla di invocazioni per replica).

Per eseguire questo studio si sono considerati uno alla volta i nodi di ognuna delle tre tipologie e si sono testate varie combinazioni di repliche e numero di invocazioni su ogni funzione. Per far sì che, poi, gli scenari simulati non causassero problemi, si è studiato sempre il **caso pessimo**; ovvero eseguendo tutte e tre le funzioni con il massimo numero testato di repliche e mandando al nodo il numero massimo di richieste per replica. Questi **stress test** sono stati necessari per consentire la generazione automatica di scenari di test che non compromettessero la raccolta dati, causando problemi nel funzionamento delle macchine virtuali per l'eccessivo numero di richieste o di repliche. In Tabella 5.2 viene riportato per ogni tipologia di nodo e per ogni funzione il numero massimo di repliche, mentre in Tabella 5.3 il numero massimo di richieste per replica.

Tabella 5.2: Tabella rappresentante il numero massimo di repliche eseguibili su ogni tipologia di nodo

	<b>funca</b>	<b>qrcode</b>	<b>ocr</b>
<b>node of type 1</b>	5	2	1
<b>node of type 2</b>	5	2	1
<b>node of type 3</b>	5	2	1

Tabella 5.3: Tabella rappresentante il numero massimo di richieste al secondo (r/s) per replica inoltrabili ad ogni funzione, per ogni tipo di nodo

	<b>funca</b>	<b>qrcode</b>	<b>ocr</b>
<b>node of type 1</b>	15	2	1
<b>node of type 2</b>	20	5	3
<b>node of type 3</b>	30	10	5

Come è possibile vedere in Tabella 5.2 il numero massimo di repliche per funzione è stato mantenuto costante per ogni nodo, in quanto dopo svariati test si è visto che ciò che effettivamente determinava il crash di alcune macchine virtuali era il numero massimo di richieste.

Il **max rate** totale, che viene attribuito come label al momento del deployment di una funzione, è facilmente ottenibile moltiplicando il numero di repliche in esecuzione per quella funzione, per il max rate per ogni replica su quel tipo di nodo, riportato in Tabella 5.3.

Una volta stabiliti questi valori è stato possibile automatizzare la generazione di scenari per la raccolta dati, rispettando il numero massimo di repliche per funzione e di traffico per replica, in base al tipo di nodo. Sono stati generati anche scenari in cui le funzioni sono sovraccaricate; ovvero viene generato un quantitativo di traffico superiore al max rate. Sono state generate circa 60 configurazioni di esperimenti differenti.

Una volta generati gli scenari degli esperimenti, sono stati presi uno ad uno ed eseguiti sul prototipo di DFaaS. Per ognuno di essi è stato impostato il nodo della tipologia su cui doveva essere svolto, sono state eseguite le funzioni in accordo con il numero di repliche generate ed è stato utilizzato Vegeta [57] per invocare le funzioni con il numero di richieste prestabilito. Vegeta [57] è uno strumento che consente di

inviare un tasso costante di richieste http configurabili per una quantità di tempo decisa dall’utente a specifici endpoint.

E’ importante sottolineare che è stato disabilitato l’**autoscaling** di OpenFaaS, in modo che la piattaforma non si prendesse la libertà di cambiare il numero di repliche delle funzioni a fronte di variazioni del quantitativo di traffico.

Ogni esperimento ha la durata di 5 minuti durante i quali viene sottoposto ogni nodo ad un traffico costante, a cui viene sommato un minuto iniziale in cui il traffico sale e un minuto finale in cui scende; si ha quindi una durata totale di 7 minuti per esperimento. L’agente, una volta al minuto, logga le metriche monitorate per ogni funzione, producendo per ogni esperimento un file json come quello mostrato in Figura 5.3. In questo file è possibile vedere che il numero di elementi presenti nella lista associata alla chiave **output** è pari a 7, ovvero alla durata di ogni esperimento. Ognuno di questi 7 elementi conterrà poi le metriche raccolte a livello di nodo e a livello delle singole funzioni presenti sul nodo. All’inizio del file, invece, in corrispondenza della chiave **input**, è presente la configurazione dello scenario generato per l’esperimento, in termini di repliche e traffico per funzione, e la tipologia di nodo su cui è stato eseguito.

Al termine di ogni esperimento i file json di riepilogo sono stati spostati al di sotto di una cartella **data**, presente nella root di progetto, al cui interno sono presenti tre cartelle, una per tipologia di nodo, per dividere gli esperimenti raccolti. Le componenti come l’*instance generator* e il *data loader*, descritti nel Capitolo 4, si avvalgono di questi dati presenti nella directory **data** per generare istanze o per popolare il database degli esperimenti.

## 5.2 Empirical Strategy

Questa strategia rappresenta l’algoritmo che è stato ideato inizialmente e che ha portato alla necessità di creare un simulatore per poterlo testare e validare a confronto con altre strategie di baseline.

La strategia empirica è un algoritmo distribuito che si basa sull’insieme di metriche monitorate sullo stato del nodo e delle funzioni su esso in esecuzione (riportate in

Sezione 5.1.1) per calcolare i pesi di inoltro delle richieste di invocazione delle funzioni. Questa strategia viene eseguita in maniera distribuita su ogni agente e restituisce, per ogni funzione nello stato di sovraccarico (overloaded), i pesi di inoltro verso tutti gli altri nodi. Questi pesi potranno essere diversi da zero solo per i nodi appartenenti al vicinato del nodo di interesse.

Durante la progettazione di questo algoritmo si è utilizzato come ispirazione il comportamento degli agenti di Gru [58] ed i concetti che vengono trattati in questa pubblicazione. In primis è stata ripresa l'idea di strutturare il comportamento dell'agente seguendo il design pattern **MAPE-K** (Monitor-Analyze-Plan-Execute over a shared Knowledge) feedback-loop (vedi Figura 5.4) [58] [59]. Il **MAPE-K** feedback-loop è costituito da quattro elementi architetturali:

- **Monitor.** Collezione i dettagli, le metriche o quanto necessario ai fini dell'analisi, dal sistema gestito (**Managed System**).
- **Analyze.** Esegue un'analisi e un ragionamento sui dati raccolti dal monitoring, tale analisi può essere influenzata dalla **Knowledge Base**, ovvero dai dati memorizzati dall'agente.
- **Plan.** Struttura l'azione, qualora necessaria, per raggiungere un determinato obiettivo.
- **Execute.** Attua il comportamento sul sistema gestito (**Managed System**).

Inoltre, è presente una **Knowledge Base** che è costituita da un insieme di dati condivisi tra le quattro fasi sequenziali descritte precedentemente.

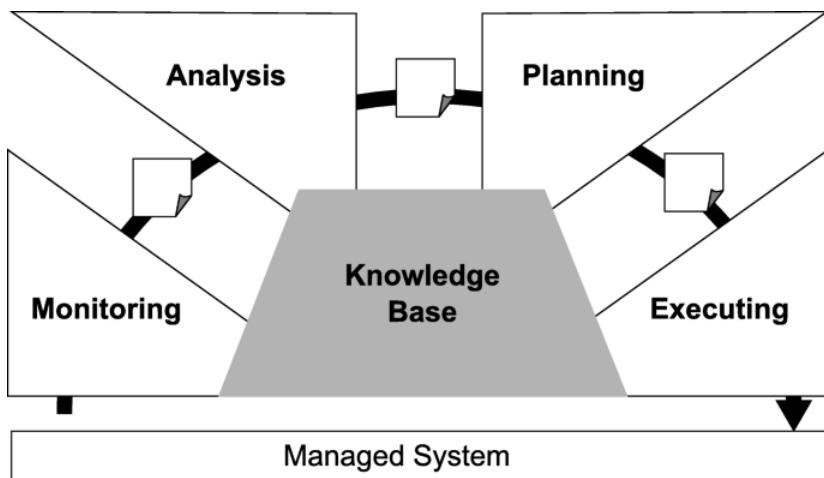


Figura 5.4: Architettura del MAPE-K feedback-loop.

### 5.2.1 Descrizione Algoritmo

L'algoritmo utilizza quindi un approccio empirico, ovvero un approccio sperimentale, basato su evidenze raccolte sul campo. In questo contesto l'agente si baserà quindi sulle metriche che sintetizzano lo stato di salute di un nodo e delle funzioni in esse presenti, per bilanciare il traffico.

Questo algoritmo sfrutta quindi le metriche raccolte dal monitoring del sistema, ottenute dagli agenti vicini, per produrre in output i pesi di inoltro per ogni classe di funzione sovraccaricata, verso gli altri nodi della rete.

L'agente esegue ciclicamente, ad istanti di tempo regolari, le seguenti fasi: **Monitoring, Exchange, Analyze, Plan e Execute**.

#### Monitoring

In questa fase viene interrogato il *gateway* di OpenFaaS per reperire le metriche riportate in Sezione 5.1.1. Inoltre, sfruttando quanto implementato precedentemente nell'agente, vengono interrogate le tabelle di HAProxy per reperire le informazioni sulle richieste che arrivano ad ogni agente e quante effettivamente vengono inoltrate ad esso.

Una volta raccolte queste informazioni viene determinato quello che è lo stato delle funzioni in esecuzione sul nodo. A tal proposito è importante ricordare che ogni funzione in esecuzione su DFaaS deve essere accompagnata da una label (di nome **maxrate**) che indica il max rate per quella funzione. Anche il max rate viene aggiunto tra le metriche raccolte a livello delle singole funzioni. I possibili stati, come nella versione attuale di DFaaS, sono due:

1. **Normal.** Una funzione in questo stato non è sovraccaricata, quindi il numero di richieste di invocazioni effettuate dall'utente è inferiore al max rate.
2. **Overload.** Una funzione in questo stato è sovraccaricata, ovvero il numero di richieste di invocazione supera quello che è il max rate ad essa associato al momento del deployment.

Oltre lo stato della funzione, per ognuna di esse viene anche calcolato il **margine**, riportato anch'esso tra le metriche monitorate. Se il **max rate** è maggiore dell'**invocation rate** tale valore sarà dato dalla differenza tra i due, mentre sarà pari a zero in caso contrario. Il valore di tale metrica rappresenta infatti il numero di richieste ancora servibili da una funzione.

Ciò che viene prodotto in questa fase deve essere un file json che contiene le metriche raccolte, un esempio viene riportato in Figura 5.5.

```

"node_0": {
    "cpu_usage": 0.3381,
    "ram_usage": 0.37555000000000005,
    "functions": [
        {
            "name": "funca",
            "invoc_rate": 39,
            "afet": 0.0011331646532805,
            "cpu_xfunc": 1.575,
            "prom_invoc_rate": 29.993818461142098,
            "ram_xfunc": 2.3425,
            "service_count": 2,
            "margin": 0,
            "state": "Overload",
            "max_rate": 30
        },
        {
            "name": "ocr",
            "invoc_rate": 0,
            "afet": 0.00100000000000002,
            "cpu_xfunc": 2.5,
            "prom_invoc_rate": 0.0,
            "ram_xfunc": 1.1499999999999997,
            "service_count": 1,
            "margin": 1,
            "state": "Normal",
            "max_rate": 1
        },
        {
            "name": "qrcode",
            "invoc_rate": 4,
            "afet": 0.0263997131840909,
            "cpu_xfunc": 8.4625,
            "prom_invoc_rate": 3.9997818333872392,
            "ram_xfunc": 0.7675000000000001,
            "service_count": 2,
            "margin": 0,
            "state": "Overload",
            "max_rate": 4
        }
    ],
}

```

Figura 5.5: Esempio di json prodotto dalla fase di monitoring dell'agente.

## Exchange

In questa fase l'agente invia al proprio vicinato, ovvero agli agenti sui nodi con cui è in grado di comunicare, il json costruito nella fase precedente. Lo scambio avviene attraverso una rete p2p (peer-to-peer). Prendendo spunto da Gru, la comunicazione avviene con un sottoinsieme di peer selezionati, ovvero quell'insieme di nodi che appartengono al vicinato del nodo di interesse, e non in broadcast con tutti gli altri nodi. In questo modo si evita il problema del floating delle richieste, ovvero di rischiare di sovraccaricare la rete con un'inondazione di messaggi. Interessante notare come questa fase non debba avvenire per forza in maniera sincrona, infatti, quando ogni agente eseguirà il proprio loop, si occuperà in questa fase di inviare i dati appena raccolti, ma non eseguirà nessuna operazione bloccante in attesa di messaggi dagli altri nodi. Ogni agente durante il ricalcolo agirà in maniera totalmente **asincrona**, calcolando i pesi di inoltro, basandosi sulle ultime informazioni ottenute dal proprio vicinato.

Una volta ricevute le informazioni, ogni nodo avrà conoscenza dello stato dei propri vicini, in un unico file di configurazione (come mostrato in Figura 5.6).

```

"node_2": {
    "functions": [
        {
            "name": "funca",
            "invoc_rate": 9,
            "afet": "0.010080972089656779,
            "cpu_xfunc": 0.2674999999999996,
            "prom_invoc_rate": 10.000272737190532,
            "ram_xfunc": 1.4525000000000001,
            "service_count": 4,
            "margin": 111,
            "state": "Normal",
            "max_rate": 120
        },
        {
            "name": "qrcode",
            "invoc_rate": 31,
            "afet": "0.06753094725301971,
            "cpu_xfunc": 12.697500000000002,
            "prom_invoc_rate": 20.036910433118862,
            "ram_xfunc": 0.315,
            "service_count": 2,
            "margin": 0,
            "state": "Overload",
            "max_rate": 20
        },
        {
            "name": "ocr",
            "invoc_rate": 0,
            "afet": "0.0",
            "cpu_xfunc": 0.5549999999999999,
            "prom_invoc_rate": 0.0,
            "ram_xfunc": 0.0525,
            "service_count": 1,
            "margin": 5,
            "state": "Normal",
            "max_rate": 5
        }
    ],
    "cpu_usage": 0.230775,
    "ram_usage": 0.15130000000000002
},
1,
"node_4": {
    "functions": [
        {
            "name": "funca",
            "invoc_rate": 158,
            "afet": "0.0013508853704859833,
            "cpu_xfunc": 3.844999999999998,
            "prom_invoc_rate": 100.00636320659865,
            "ram_xfunc": 2.1550000000000002,
            "service_count": 5,
            "margin": 0,
            "state": "Overload",
            "max_rate": 100
        },
        {
            "name": "ocr",
            "invoc_rate": 0,
            "afet": "0.0",
            "cpu_xfunc": 1.9,
            "prom_invoc_rate": 0.0,
            "ram_xfunc": 0.0625,
            "service_count": 1,
            "margin": 3,
            "state": "Normal",
            "max_rate": 3
        },
        {
            "name": "qrcode",
            "invoc_rate": 0,
            "afet": "0.0",
            "cpu_xfunc": 0.0,
            "prom_invoc_rate": 0.0,
            "ram_xfunc": 0.0899999999999998,
            "service_count": 2,
            "margin": 10,
            "state": "Normal",
            "max_rate": 10
        }
    ],
    "cpu_usage": 0.31405,
    "ram_usage": 0.204925
},
211

```

Figura 5.6: Esempio di json ottenuto dopo la fase di comunicazione, supponendo che il nodo con id 0 abbia come vicini i nodi 2 e 4.

## Analyze

Questa rappresenta la fase più importante e centrale dell'algoritmo in cui l'agente, sfruttando le informazioni scambiate con gli altri nodi, calcola i pesi di inoltro verso di essi.

Avendo a disposizione le metriche ottenute dagli altri nodi, per ogni funzione nello stato **overload**, è necessario attuare il bilanciamento del carico. Il traffico verrà bilanciato per queste funzioni verso tutti gli altri nodi appartenenti al vicinato, che possiedono la medesima funzione ma in uno stato **normal**, ovvero non in sovraccarico (con un **margine** maggiore di zero). La fase di analisi e calcolo dei pesi, per ogni funzione sovraccaricata, si articola in due sotto-fasi:

1. **Calcolo dei pesi a livello delle singole metriche** (vedi Algoritmo 1). Questa fase è costituita dalle due funzioni riportate nell’Algoritmo 1, si suppone che la computazione parta dalla prima funzione rappresentata, che a sua volta utilizzerà internamente la seconda. In questa fase, per ogni funzione in stato di **overload** (Linea 3), è necessario calcolare quelli che sono i pesi di inoltro relativi ad ogni metrica (afet, invoc rate, max rate, ecc.). Si considera quindi singolarmente ogni metrica e, per ognuna di esse, il valore che assume in ognuno dei nodi del vicinato che hanno la funzione in esame, ma in uno stato di **normal** (Linea 6). In questo modo è possibile attribuire un peso ad ogni metrica per ogni nodo del vicinato (tramite la funzione a Linea 11). E’ necessario tenere in considerazione il fatto che le metriche possono essere di due tipologie:

- **Proporzionalmente dirette.** Maggiore è il valore della metrica per un determinato nodo, maggiore deve essere il valore del peso attribuito a quel nodo. Metriche dirette sono il **service count** (numero di repliche per ogni funzione) e il **margine**.

La formula utilizzata per il calcolo del peso di queste metriche è la seguente (applicata a Linea 21):

$$w_{metric\_name(node_id)} = \frac{metric\_value(node_id)}{\sum_{x \forall x \in neighbours\_ids} metric\_value(node_x)} \quad (5.1)$$

Prendendo ad esempio **service count** per la funzione `funca`, per calcolare il peso di questa metrica verso il nodo 1 ( $w_{service\_count(node_1)}$ ) si prende il valore di questa metrica sul nodo 1 e si divide per la somma dei valori delle metriche negli altri nodi del vicinato che hanno la funca nello stato di **normal**, calcolata dal ciclo di Linea 15.

E’ bene notare che eseguendo il calcolo in questo modo quella che si ottiene è una **distribuzione di probabilità**. Ovvero la somma dei pesi della metrica

**service count** per una determinata funzione, verso tutti i nodi del vicinato che hanno la medesima funzione nello stato di non sovraccarico, sarà 1.

- **Proporzionalmente inverse.** Maggiore è il valore della metrica per un determinato nodo, minore deve essere il valore del peso attribuito a quel nodo. Le metriche di questo tipo sono tutte le rimanenti tra cui: invocation rate, max rate, utilizzo di RAM e CPU del nodo, utilizzo di RAM e CPU a livello di funzione e AFET.
  - **Nota.** In questo calcolo l'invocation rate e il max rate sono utilizzati in maniera combinata in quanto l'invocation rate preso singolarmente è poco significativo. Per tale ragione l'invocation rate viene diviso per il max rate.

Per le metriche di questo tipo è necessario eseguire due formule per ottenere dei pesi inversamente proporzionali al valore delle metriche. Per prima cosa viene applicata per ogni funzione e per ogni metrica la seguente formula (vedi Linea 25):

$$w_{metric\_name(node_id)} = \frac{1}{metric\_value(node_id)} \quad (5.2)$$

Successivamente, sui valori ottenuti applicando Equazione (5.2), si riapplica l'Equazione (5.1) (vedi Linea 28) per ottenere una distribuzione di probabilità, utilizzando i pesi ottenuti al posto dei valori delle metriche.

---

**Algorithm 1:** Algorithm used to calculate weights for each metric related to each **overloaded** function

---

**Data:** Json file containing all metrics exchanged by agents, gathered during monitor phase

**Result:** Weights for each metric, calculated for each **overloaded** function, towards neaby nodes

```
1 Function MetricWeights(overloaded_funcs, neighbourhood):
2     weights = {}
3     for func in overloaded_funcs do
4         helpers = {}
5         for neighbour in neighbourhood do
6             if func deployed in neighbour & func is normal then
7                 help_func = get_func_metrics_in_neigh(func, neighbour)
8                 helpers[neighbour] = help_func
9                 weights[func] = ComputeWeight(helpers)
10    return weights
11 Function ComputeWeight(helpers):
12     weights = {}
13     for metric in METRIC_SET do
14         sum = 0
15         for neighbour in helpers do
16             value = get_metric_from_neigh(metric, neighbour)
17             sum += value
18             w_metric[neighbour] = value
19         if metric is DIRECT then
20             for neighbour, metric_value in w_metric do
21                 w_metric[neighbour] = metric_value / sum
22         else
23             sum = 0
24             for neighbour, metric_value in w_metric do
25                 w_metric[neighbour] = 1 / metric_value
26                 sum += 1 / metric_value
27             for neighbour, metric_value in w_metric do
28                 w_metric[neighbour] = metric_value / sum
29             for neighbour, w in w_metric do
30                 weights[neighbour][metric] = w
31 return weights
```

---

2. **Aggregazione dei pesi per ogni funzione** (vedi Algoritmo 2). Una volta ottenuti i pesi per ogni metrica, verso gli altri nodi del vicinato per la funzione in esame, è possibile aggregarli per ottenere il peso complessivo di inoltro verso uno specifico nodo. Questi pesi sono quelli calcolati dalla fase precedente e rappresentati nell’Algoritmo 2, attraverso il parametro di input *weights\_for\_metrics* a Linea 1. Tale parametro contiene un dizionario a tre livelli che viene iterato per ogni funzione (vedi Linea 3), per ogni nodo vicino (vedi Linea 5) e per ogni metrica (vedi Linea 7). Ad ogni tupla, rappresentata dalla tripletta (funzione, nodo, metrica), è associato l’effettivo peso calcolato nella fase precedente.

Per prima cosa viene applicata la seguente formula per aggregare i pesi delle singole metriche attribuiti ad uno specifico nodo (applicata a Linea 8):

$$w_{func\_name \rightarrow node_id} = \sum_{w \forall w_{metric\_name(node_id)}} w \quad (5.3)$$

Anche in questo caso, una volta fatta questa operazione, è necessario ottenere una **distribuzione di probabilità** (applicata a Linea 10), difatti utilizzando i valori ottenuti da Equazione (5.3) al posto dei valori delle singole metriche, viene applicata l’Equazione (5.1).

In questa seconda fase si potrebbero aggiungere dei moltiplicatori per le singole metriche, in modo da pesarle diversamente a seconda del tipo di bilanciamento che si vuole effettuare. Questi moltiplicatori potrebbero essere inseriti nell’Equazione (5.3), affiancati ad ogni  $w$ , mentre nello pseudocodice sono quelli ottenuti dal dizionario *METRIC\_WEIGHTS* a Linea 8. In questo lavoro i pesi sono stati lasciati tutti uguali, ma è stato predisposto il calcolo perché possano essere cambiati in un qualsiasi momento qualora si volesse ottenere un bilanciamento atto ad ottimizzare ad esempio l’utilizzo di RAM, di CPU ecc. Questi pesi potrebbero essere altrimenti calcolati sfruttando un modello di Machine Learning.

---

**Algorithm 2:** Algorithm used to calculate final weights for each **overloaded** function toward nearby nodes

---

**Data:** *weights*: A dictionary containing weights, for each **overloaded function**, for each metric toward nearby nodes

**Result:** Weights towards other nodes, calculated for each **overloaded function**

```
1 Function WeightsAggregation(weights_for_metrics):
2     weights = {}
3     for func, w_for_nodes in weights_for_metrics do
4         weights[func] = {}
5         for node, w_for_metrics in w_for_nodes do
6             sum = 0
7             for metric, w in w_for_metrics do
8                 sum += w * METRIC_WEIGHTS[metric]
9             weights[func][node] = sum
10    return recalc_distribution(weights)
```

---

Di seguito viene riportato un esempio del calcolo eseguito dalle due fasi appena elencate. Si suppone di avere uno scenario come quello riportato in Figura 5.7 in cui si hanno 4 nodi e si vuole bilanciare il traffico prodotto da Vegeta sulla funzione **funca** del nodo\_0. Questa funzione si troverà in uno stato di **overloaded**, di conseguenza l'agente sul nodo\_0 dovrà bilanciare il traffico verso i vicini che possiedono la medesima funzione nello stato di **normal**. Si suppone di avere a disposizione solo un sottoinsieme delle metriche totali per facilitare la fase di calcolo nell'esempio. Per mostrare un calcolo di bilanciamento si utilizza una metrica direttamente proporzionale (**service count**) e una inversamente proporzionale (rapporto tra **invocation rate** e **max rate**).

Vengono di seguito applicati quelli che sono gli step della fase di **Analyze** alla funzione **funca** sul nodo\_0, in cui l'**invocation rate** è maggiore del **max rate** e quindi la funzione si trova in uno stato di **overloaded**.

Per quanto riguarda lo step 1, ovvero il **calcolo dei pesi a livello delle singole metriche**, si predono in esame una dopo l'altra le metriche a disposizione.

Nel caso del **service count** questa rappresenta una metrica **direttamente proporzionale**, quindi viene applicata l'Equazione (5.1) per ogni nodo. Viene calcolato il peso di questa metrica per il nodo\_1 (Equazione (5.4)), per il nodo\_2 (Equazio-

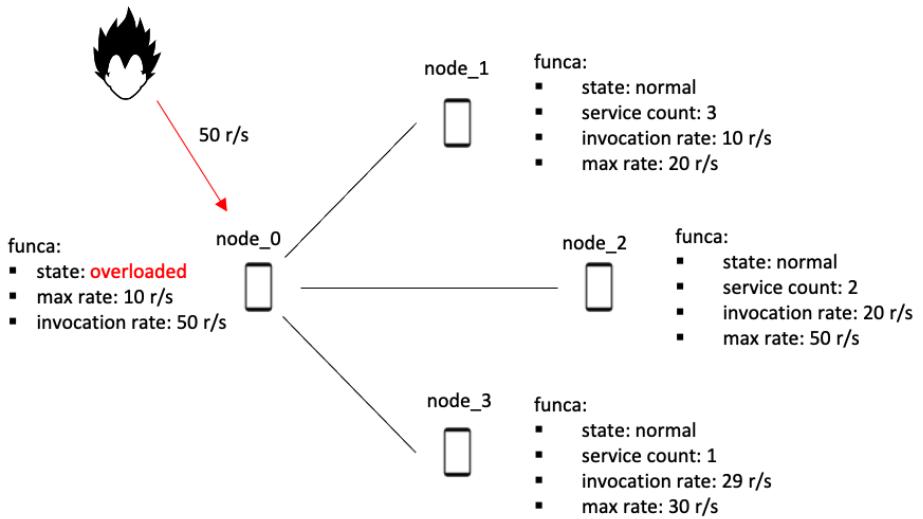


Figura 5.7: Esempio di situazione in cui bilanciare il traffico.

ne (5.5)) ed infine per il nodo\_3 (Equazione (5.6)). Notare come la somma dei 3 dia uno, ovvero formi una distribuzione di probabilità.

$$w_{service\_count(node_1)} = \frac{3}{3+2+1} = 0.5 \quad (5.4)$$

$$w_{service\_count(node_2)} = \frac{2}{3+2+1} = 0.333 \quad (5.5)$$

$$w_{service\_count(node_3)} = \frac{1}{3+2+1} = 0.1667 \quad (5.6)$$

Successivamente si considera l'**invocation rate** che viene considerato in rapporto al **max rate**. Anche in questo caso viene eseguita la prima fase per tutti e tre i nodi.

$$w_{invocation\_rate(node_1)} = \frac{1}{10/20} = 2 \quad (5.7)$$

$$w_{invocation\_rate(node_2)} = \frac{1}{20/50} = 2.5 \quad (5.8)$$

$$w_{invocation\_rate(node_3)} = \frac{1}{29/30} = 0.9667 \quad (5.9)$$

Essendo una metrica **proporzionalmente inversa**, subito dopo aver applicato l’Equazione (5.2), si riapplica l’Equazione (5.1) utilizzando i valori appena calcolati al posto di quelli delle metriche. Questo viene fatto per ottenere una distribuzione di probabilità.

$$w_{invocation\_rate(node_1)} = \frac{2}{2 + 2.5 + 0.9667} = 0.36 \quad (5.10)$$

$$w_{invocation\_rate(node_2)} = \frac{2.5}{2 + 2.5 + 0.9667} = 0.46 \quad (5.11)$$

$$w_{invocation\_rate(node_3)} = \frac{1}{2 + 2.5 + 0.9667} = 0.18 \quad (5.12)$$

Una volta ottenuta per ogni metrica la distribuzione di probabilità verso gli altri nodi della rete, che possiedono la funzione **funca** in stato di **normal**, si procede con la fase 2, ovvero **l’aggregazione dei pesi per ogni funzione**.

Per prima cosa viene eseguita la funzione di aggregazione scelta, in questo caso la somma riportata nell’Equazione (5.3), per ogni nodo in esame e si ottengono i seguenti risultati. In questo esempio sono considerate solo le due metriche riportate, ovvero il **service count** e l’**invocation rate** (considerato in rapporto con il **max rate**).

$$w_{funca \rightarrow node_1} = w_{service\_count(node_1)} + w_{invocation\_rate(node_1)} = 0.5 + 0.36 = 0.86 \quad (5.13)$$

$$w_{funca \rightarrow node_2} = w_{service\_count(node_2)} + w_{invocation\_rate(node_2)} = 0.333 + 0.46 = 0.793 \quad (5.14)$$

$$w_{funca \rightarrow node_3} = w_{service\_count(node_3)} + w_{invocation\_rate(node_3)} = 0.1667 + 0.18 = 0.3467 \quad (5.15)$$

Una volta fatto questo viene riapplicata nuovamente l'Equazione (5.1) sui valori appena ottenuti, per ottenere una distribuzione di probabilità.

$$w_{funca \rightarrow node_1} = \frac{0.86}{0.86 + 0.793 + 0.3467} = 0.43 \quad (5.16)$$

$$w_{funca \rightarrow node_2} = \frac{0.793}{0.86 + 0.793 + 0.3467} = 0.40 \quad (5.17)$$

$$w_{funca \rightarrow node_3} = \frac{0.3467}{0.86 + 0.793 + 0.3467} = 0.17 \quad (5.18)$$

Al termine del procedimento quelli ottenuti sono i pesi di inoltro calcolati dall'agente sul nodo\_0 per il nodo\_1 (Equazione (5.16)), per il nodo\_2 (Equazione (5.17)) e per il nodo\_3 (Equazione (5.18)), relativi alla funzione **funca**.

Le 40 richieste eccedenti sul nodo\_0 verranno quindi distribuite sugli altri 3 nodi con le probabilità appena indicate. Indicativamente verranno così distribuite:

- **Numero di richieste verso il nodo\_1** =  $40 * 0.43 = 17$
- **Numero di richieste verso il nodo\_2** =  $40 * 0.40 = 16$
- **Numero di richieste verso il nodo\_3** =  $40 * 0.17 = 7$

E' importante notare che i valori delle metriche utilizzati in questo esempio sono stati ipotizzati solo al fine di semplificare il calcolo. Inoltre, si sono utilizzate solamente due metriche, non considerando quelle relative allo stato del nodo (uso CPU e RAM) o relative al consumo delle singole funzioni. I pesi calcolati potrebbero quindi non rispecchiare quanto ci si poteva aspettare dalla situazione di esempio.

## Plan

Questa fase prende in input i pesi prodotti dall'**Analyze** e attua una semplice *pianificazione* probabilistica su di essi.

In reti piccole, in cui tutti gli agenti si possono vedere a vicenda, o in piccoli cluster di una rete in cui i nodi hanno vicini simili, potrebbe verificarsi una situazione in

cui l'algoritmo distribuito fin ora ideato possa convergere e calcolare gli stessi pesi di inoltro su tutti i nodi. Questa convergenza porterebbe ad uno stato in cui tutti i nodi bilanciano il traffico principalmente verso lo stesso nodo, che potrebbe essere quello maggiormente scarico o quello con più risorse computazionali a disposizione. Seppur possa sembrare un'assunzione sensata bisogna considerare il fatto che il nodo nella finestra temporale successiva, prima del prossimo ricalcolo dell'algoritmo, si possa trovare inondato di richieste da tutti gli altri, più eventuali provenienti dall'utente.

Per cercare di limitare, seppur in piccola parte, questo problema, nella fase di **Plan** viene aggiunto, con una certa probabilità, un rumore probabilistico ai pesi calcolati dal nodo, e poi ricalcolata la distribuzione di essi. In questo caso ogni nodo aggiunge del rumore ai pesi calcolati, provocando una maggior incertezza e diminuendo la probabilità che vengano calcolati pesi molto simili da agenti su nodi differenti.

La strategia applicata è molto semplice, ed è quella mostrata dall'Algoritmo 3.

---

**Algorithm 3:** Algorithm used to add probabilistic noise to calculated weights for each **overloaded** function

---

**Data:** Weights calculated by agent during **Analyze** phase for a specific function ( $[w_1, \dots, w_n]$ )  
**Result:** New weights that forms a probability distribution

```

1 for  $w_i$  in  $[w_1, \dots, w_n]$  do
2   | rnd = random(0, 100)
3   | if rnd > 60 then
4     |   | return
5     |   rumor = random(0, 40) *  $w_i$  / 100
6     |   rnd = random(0, 100)
7     |   if rnd > 50 then
8       |     |  $w_i$  += rumor
9     |   else
10    |     |  $w_i$  -= rumor
11  | [w_1, ..., w_n] = recalc_distribution([w_1, ..., w_n])
12 return [w_1, ..., w_n]

```

---

Come mostrato nell'Algoritmo 3, per ogni funzione nello stato di **overloaded**, vengono presi in input i pesi calcolati nella fase di **Analyze** e ad essi viene aggiunto una certa quantità di rumore, con una probabilità del 60%. Questo algoritmo viene

infatti eseguito per ogni funzione nello stato di sovraccarico, e per ognuna di esse prende in input il vettore dei pesi di inoltro verso tutti gli altri nodi della rete.

Successivamente viene calcolato il rumore che può variare tra lo 0% e il 40% del peso  $w_i$  in esame. Con una probabilità del 50% questo rumore viene sommato al peso, altrimenti sottratto, per poi passare al peso successivo.

Una volta terminati tutti i pesi del vettore di input, per la funzione in esame, si ricalcola la distribuzione di probabilità per i pesi appena calcolati.

## Execute

La fase di **Execute** si occupa di andare a mettere in pratica quanto calcolato dalle fasi precedenti. Vengono infatti applicati quelli che sono i pesi ottenuti per ogni funzione in stato di sovraccarico, al fine di bilanciare il carico delle richieste.

In DFaaS la componente che si interpone tra l'arrivo delle richieste http di invocazione di una funzione e il cluster OpenFaaS che le esegue è HAProxy [32]. Proprio per questo l'agente, nella fase di **Execute**, si occuperà di andare ad aggiornare il file di configurazione del proxy, istruendolo sulle probabilità di inoltro da utilizzare.

### 5.2.2 Esperimento

Una volta definito a livello teorico l'algoritmo, questo è stato implementato con i necessari adeguamenti sul simulatore. In questa sezione viene mostrato come è stato validato il seguente algoritmo in un'istanza di esempio. Tale istanza è stata poi anche utilizzata nell'esperimento complessivo riguardante tutti gli altri algoritmi, per poi metterli a confronto tutti insieme.

I valori dei parametri utilizzati nell'esperimento associato ad ogni algoritmo per ottenere la medesima istanza su cui eseguire il test sono i seguenti:

- **Seed.** 701.
- **Nodes Number.** 10.
- **Edge probability.** 0.3.

Tali valori possono essere passati come argomenti a linea di comando al *simulation controller* per riprodurre l'esperimento con la medesima istanza; in alternativa è possibile passargli direttamente l'intera rappresentazione dell'istanza in formato json.

La topologia di rete dell'istanza simulata è quella riportata in Figura 5.8.

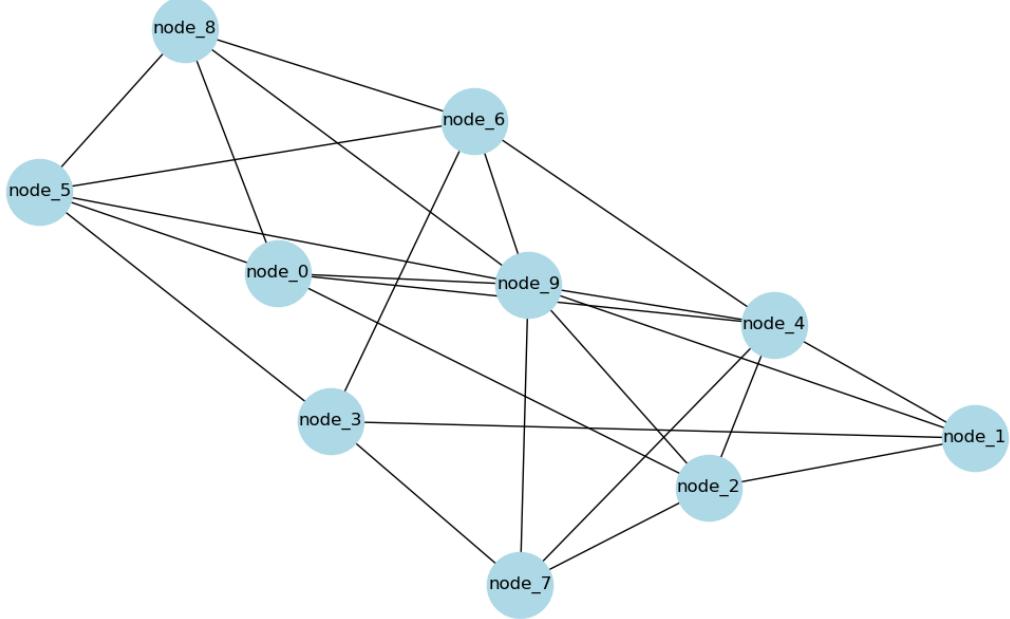


Figura 5.8: Esempio istanza generata ai fini della comparazione dei vari algoritmi implementati.

Di seguito, in Figura 5.9, vengono mostrati i report prodotti dal *simulator* durante la simulazione della strategia **empirica** su questa istanza.

Prendendo come esempio il nodo\_1, il suo max rate per quanto riguarda la funzione **funca** è pari a 100 r/s, ma questa è sottoposta ad un totale di 177 r/s. Questo significa che la funzione **funca** sarà sovraccaricata e l'agente dovrà eseguire la strategia **empirica** per la distribuzione del carico.

Il vicinato del nodo\_1 è costituito dai nodi 2, 3, 4 e 9 come è possibile evincere dalla topologia di rete in Figura 5.8.

I pesi calcolati al termine dell'esecuzione della strategia empirica sono reperibili dal log dell'agente, in particolare quelle rappresentate in Figura 5.9 sono le tabelle relative alla terza iterazione della simulazione. I pesi si possono quindi vedere in Figura 5.10, in cui viene raffigurato il file di log dell'agente che viene eseguito sul nodo\_1.

Tabella richieste inoltrate - funca

	<b>node_0</b>	<b>node_1</b>	<b>node_2</b>	<b>node_3</b>	<b>node_4</b>	<b>node_5</b>	<b>node_6</b>	<b>node_7</b>	<b>node_8</b>	<b>node_9</b>
<b>node_0</b>	30	0	5	0	0	0	0	0	4	0
<b>node_1</b>	0	100	59	18	0	0	0	0	0	0
<b>node_2</b>	0	0	9	0	0	0	0	0	0	0
<b>node_3</b>	0	0	0	19	0	0	0	0	0	0
<b>node_4</b>	0	0	29	0	100	0	0	29	0	0
<b>node_5</b>	0	0	0	4	0	30	0	0	5	0
<b>node_6</b>	0	0	0	21	0	0	100	0	56	0
<b>node_7</b>	0	0	0	0	0	0	0	9	0	0
<b>node_8</b>	0	0	0	0	0	0	0	0	16	0
<b>node_9</b>	0	0	0	0	0	0	0	0	0	0

Tabella invocation rate

	<b>funca</b>	<b>qrcode</b>	<b>ocr</b>
<b>node_0</b>	39	4	0
<b>node_1</b>	177	0	9
<b>node_2</b>	9	31	0
<b>node_3</b>	19	0	7
<b>node_4</b>	158	0	0
<b>node_5</b>	39	4	0
<b>node_6</b>	177	0	9
<b>node_7</b>	9	31	0
<b>node_8</b>	16	0	1
<b>node_9</b>	0	6	0

Tabella max\_rate

	<b>funca</b>	<b>qrcode</b>	<b>ocr</b>
<b>node_0</b>	30	4	1
<b>node_1</b>	100	10	3
<b>node_2</b>	120	20	5
<b>node_3</b>	30	4	1
<b>node_4</b>	100	10	3
<b>node_5</b>	30	4	1
<b>node_6</b>	100	10	3
<b>node_7</b>	120	20	5
<b>node_8</b>	60	0	3
<b>node_9</b>	0	10	0

Figura 5.9: Esempio di distribuzione del carico applicata sulla **funca** dalla strategia empirica al termine di una iterazione della simulazione. Vengono riportate la tabella dei max rate, degli invocation rate e delle richieste inoltrate relativamente alla **funca**.

```

19   ----- MINUTE 3 -----
20   | > STRATEGY: empirical_strategy <
21   Weights normalized for func ocr: {'node_2': 58.046573749238526, 'node_4': 41.953426250761474, 'node_3': 0, 'node_9': 0}
22   Weights normalized for func funca: {'node_3': 26.19094871716035, 'node_2': 73.80905128283965, 'node_4': 0, 'node_9': 0}
23

```

Figura 5.10: Pesi calcolati dall'agente del nodo\_1 per la funzione **funca**.

Utilizzando questi pesi sono quindi inoltrate le richieste come si può vedere dalla seconda riga della tabella delle richieste per la **funca**, riportata in alto in Figura 5.9. Il traffico, rappresentato dalle 77 r/s eccedenti in arrivo alla **funca** del nodo\_1, viene

inoltrato verso il nodo 2 e il nodo 3, e così suddiviso:

- 59 r/s inoltrate al nodo\_2; queste rappresentano circa il 77% delle 77 r/s totali eccedenti (il peso di inoltro verso il nodo\_2 è del 74%).
- 18 r/s inoltrate al nodo\_3; queste rappresentano circa il 23% delle 77 r/s totali eccedenti (il peso di inoltro verso il nodo\_3 è del 26%).

Non viene distribuito sui nodi 4 e 9 in quanto: il nodo\_4 ha la funzione **funca** ma in uno stato di sovraccarico, mentre il nodo\_9 non possiede nessuna replica della **funca**. Queste considerazioni sono sempre state fatte osservando le tabelle degli invocation rate e dei max rate in Figura 5.9.

Una volta fatte queste considerazioni, ed aver dimostrato che il quantitativo delle richieste inoltrate verso i nodi del vicinato corrisponde ai pesi calcolati, viene riportato il grafico prodotto dall'*analyzer*, che riporta l'andamento dell'indice **success rate** durante gli step di simulazione (vedi Figura 5.11).

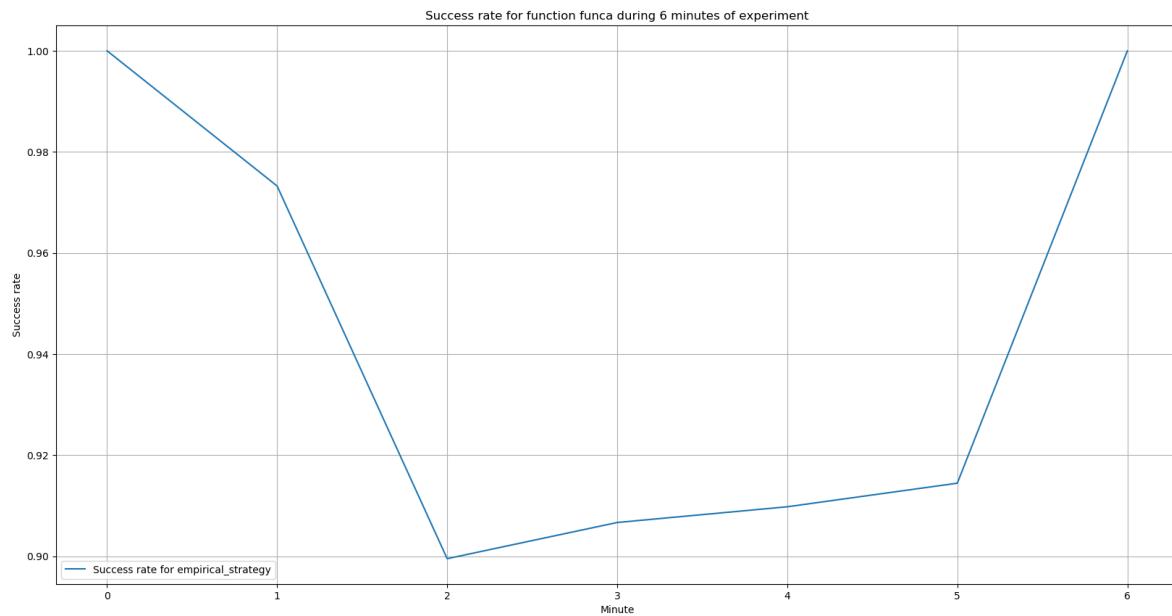


Figura 5.11: Andamento del **success rate** durante la simulazione per la funzione **funca**, applicando la strategia **empirica**.

## 5.3 Base Strategy

In questa sezione viene riportata la strategia di **baseline** più semplice utilizzata, ovvero la **Base** strategy.

### 5.3.1 Descrizione Algoritmo

La **Base** strategy rappresenta un algoritmo che non attua un vero e proprio comportamento, infatti questa strategia considera ogni agente come isolato, non interessandosi del proprio vicinato o della comunicazione con esso. Questo algoritmo rappresenta la situazione in cui ogni agente non comunica con gli altri e deve eseguire le richieste di funzioni basandosi solo sulle proprie capacità. Non è quindi presente nessuna logica di esecuzione distribuita, ma viene semplicemente ritornato, per tutte le funzioni nello stato di **overloaded**, un'insieme di pesi tutti uguali a zero. Questo significa che non viene attuata nessuna logica di distribuzione del carico e quindi, tutte le richieste eccedenti il max rate delle funzioni, verranno scartate.

### 5.3.2 Esperimento

Anche questa strategia è stata testata, in modo da favorire il confronto, sulla medesima istanza utilizzata per l'**Empirical** strategy, descritta nella sezione precedente.

Le tabelle dei max rate e degli invocation rate rimangono valide quelle di Figura 5.9, mentre la tabella di inoltro, in questo esperimento, è quella riportata in Figura 5.12. Come è possibile vedere in questa tabella i valori delle richieste inoltrate sono tutti pari a zero. Solo lungo la diagonale è possibile vedere valori diversi da zero, in quanto rappresentano le richieste gestite da ogni singolo nodo.

Un esempio di output dei pesi calcolati da un agente che esegue questo tipo di strategia è riportato in Figura 5.13. Questi sono i pesi per la funzione **funca**, che si trova in uno stato di **overloaded**, e sono gli stessi calcolati sempre dall'agente in esecuzione sul nodo\_1 di Figura 5.10. E' interessante notare che anche per la funzione **ocr** i pesi sono tutti uguali a zero.

Tabella richieste inoltrate - funca

	node_0	node_1	node_2	node_3	node_4	node_5	node_6	node_7	node_8	node_9
node_0	30	0	0	0	0	0	0	0	0	0
node_1	0	100	0	0	0	0	0	0	0	0
node_2	0	0	9	0	0	0	0	0	0	0
node_3	0	0	0	19	0	0	0	0	0	0
node_4	0	0	0	0	100	0	0	0	0	0
node_5	0	0	0	0	0	30	0	0	0	0
node_6	0	0	0	0	0	0	100	0	0	0
node_7	0	0	0	0	0	0	0	9	0	0
node_8	0	0	0	0	0	0	0	0	16	0
node_9	0	0	0	0	0	0	0	0	0	0

Figura 5.12: Tabella delle richieste inoltrate sfruttando la **base** strategy, relativa alla funzione **funca**.

```

3   ----- MINUTE 3 -----
4   | > STRATEGY: base_strategy <
5   | Weights normalized for func ocr: {'node_3': 0, 'node_2': 0, 'node_4': 0, 'node_9': 0}
6   | Weights normalized for func funca: {'node_3': 0, 'node_2': 0, 'node_4': 0, 'node_9': 0}

```

Figura 5.13: Pesi calcolati dall'agente del nodo\_1 per la funzione funca.

Infine, anche in questo caso, viene mostrato il grafico prodotto dall'*analyzer*, che riporta l'andamento dell'indice **success rate** durante gli step di simulazione (vedi Figura 5.14).

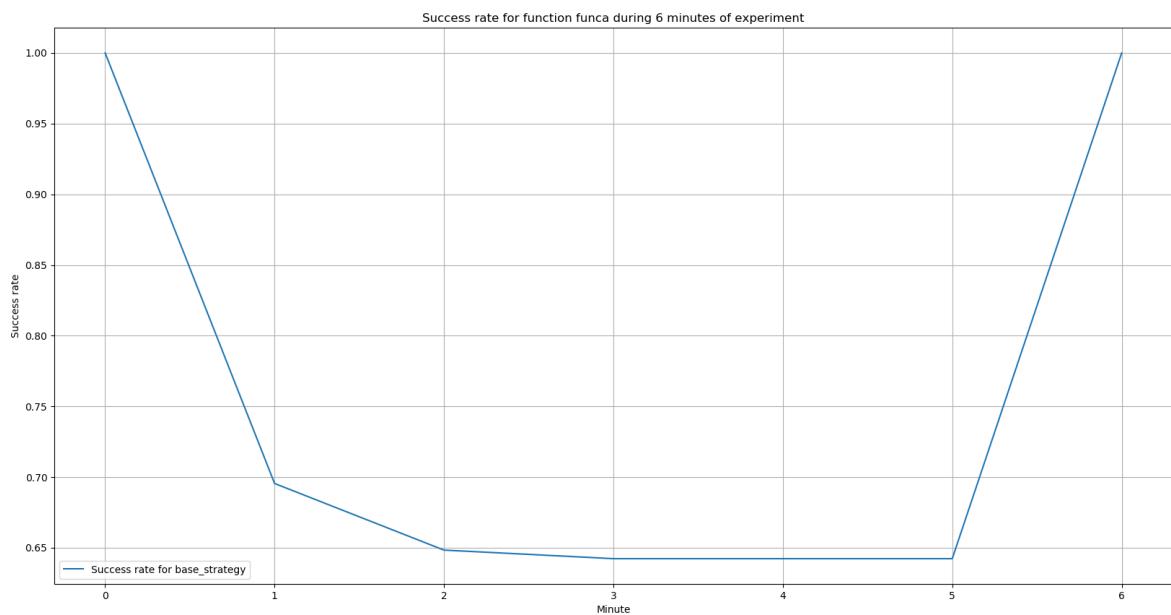


Figura 5.14: Andamento del **success rate** durante la simulazione per la funzione **funca**, applicando la **base** strategy.

## 5.4 Random Strategy

In questa sezione viene descritta una seconda strategia di **baseline** introdotta ai fini di confronto con la strategia **empirica**.

### 5.4.1 Descrizione Algoritmo

La **Random** strategy implementa un algoritmo banale, che serve da confronto immediato con la strategia empirica.

Anche in questo caso, come nel precedente, non viene applicata nessuna logica di computazione distribuita e non vengono sfruttate le informazioni del vicinato. Ogni nodo però non agisce come isolato, ma bilancia il traffico verso i nodi vicini in maniera casuale.

Ogni agente calcola quindi i pesi di inoltro verso i nodi vicini non tenendo conto delle metriche da essi raccolte, del fatto che essi possiedano o meno la funzione per cui

si sta cercando di bilanciare il carico o se questa funzione sia già sovraccaricata, ma generano pesi casuali.

### 5.4.2 Esperimento

Anche in questo caso, al fine di comparare i risultati ottenuti, viene eseguito l'esperimento sfruttando la medesima istanza di test utilizzata per la strategia **empirica**.

La tabella di inoltro delle richieste viene mostrata in Figura 5.15. Anche in questo caso, per quanto riguarda le tabelle dei max rate e degli invocation rate, rimangono valide quelle di Figura 5.9.

Come è possibile vedere in Figura 5.15, il nodo\_1 sfruttando questo algoritmo inoltra non solo ai nodi 2 e 3, ma anche al nodo\_9. Quest'ultimo nodo però, in accordo con la tabella dei max rate di Figura 5.9, non ha su di esso in esecuzione repliche della funzione **funca**. Questo dimostra che la **random** strategy inoltra le richieste in maniera casuale, senza tener conto dello stato degli altri nodi vicini.

Tabella richieste inoltrate - funca										
	node_0	node_1	node_2	node_3	node_4	node_5	node_6	node_7	node_8	node_9
node_0	30	0	2	0	1	3	0	0	3	0
node_1	0	100	23	18	0	0	0	0	0	36
node_2	0	0	9	0	0	0	0	0	0	0
node_3	0	0	0	19	0	0	0	0	0	0
node_4	9	2	14	0	100	0	10	10	0	13
node_5	0	0	0	1	0	30	3	0	4	1
node_6	0	0	0	3	24	7	100	0	21	22
node_7	0	0	0	0	0	0	0	9	0	0
node_8	0	0	0	0	0	0	0	0	16	0
node_9	0	0	0	0	0	0	0	0	0	0

Figura 5.15: Tabella delle richieste inoltrate sfruttando la **random** strategy, relativa alla funzione **funca**.

Un esempio di output dei pesi calcolati da un agente che esegue questo tipo di strategia è riportato in Figura 5.16.

Questi sono i pesi per la funzione **funca**, che si trova in uno stato di **overloaded**, calcolati dall'agente che è in esecuzione sul nodo\_1. In accordo con il traffico distribuito

in Figura 5.15 il peso verso il nodo\_9 è maggiore di zero. Inoltre, anche il peso di inoltro verso il nodo\_4 è maggiore di zero, seppur questo nodo possieda la **funca** ma in uno stato di sovraccarico.

```
3 ━━━━━ MINUTE 3 ━━━━━
4 | > STRATEGY: random_strategy <
5 | Weights normalized for func ocr: {'node_3': 18.592964824120603, 'node_2': 11.557788944723619, 'node_4': 48.24120603015075, 'node_9': 21.608040201005025}
6 | Weights normalized for func funca: {'node_3': 5.154639175257731, 'node_2': 19.072164948453608, 'node_4': 40.72164948453608, 'node_9': 35.051546391752574}
```

Figura 5.16: Pesi calcolati dall'agente del nodo\_1 per la funzione funca.

Infine, anche in questo caso, viene mostrato il grafico prodotto dall'*analyzer*, che riporta l'andamento dell'indice **success rate** durante gli step di simulazione (vedi Figura 5.17).

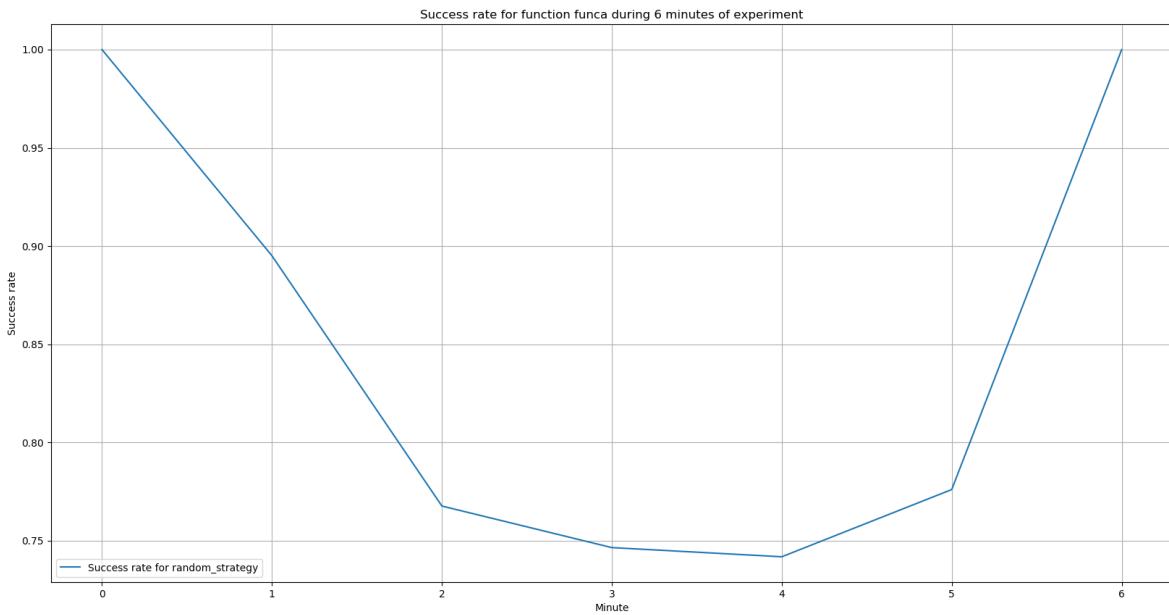


Figura 5.17: Andamento del **success rate** durante la simulazione per la funzione **funca**, applicando la **random** strategy.

## 5.5 DFaaS Static Strategy

Per ultima è stata implementata la semplice strategia statica di distribuzione del carico attuata sul prototipo di DFaaS.

Il funzionamento dettagliato di questo algoritmo è disponibile in [1], di Ciavotta et. al.

### 5.5.1 Descrizione Algoritmo

La **DFaaS Static** strategy è un semplice algoritmo distribuito di bilanciamento del carico implementato in DFaaS. Questo algoritmo, a differenza delle ultime due strategie presentate, sfrutta le informazioni ottenute dal vicinato per il calcolo dei pesi.

L'algoritmo si articola in due fasi principali, tra le quali si interpone una fase di comunicazione [1]:

1. **Fase 1.** In questa prima fase vengono raccolte informazioni dal cluster di OpenFaaS e da HAProxy, tra cui, per ogni funzione in esecuzione, il max rate, l'in-

vocation rate, ecc. Successivamente per ogni funzione ne viene determinato lo stato; sarà in stato di **overloaded**, altrimenti in stato di **underloaded**. Per ogni funzione nello stato di **underloaded** ogni agente calcola quelli che sono i **limitIn**. Questi limiti corrispondono al possibile numero di richieste ammissibili per ogni funzione da ogni altro nodo della rete p2p.

Questa prima fase dell'algoritmo viene quindi eseguita solo per le funzioni nello stato di **underloaded**.

2. **Comunicazione.** Una volta terminata questa fase l'agente invia a tutti i nodi della rete p2p quelli che sono i limiti appena calcolati (**comunicazione globale**). Ogni agente, alla ricezione delle informazioni da parte degli altri nodi, memorizza i **limitIn** ricevuti nella variabile **limitOut** associata a quel determinato nodo. I **limitOut** rappresentano il massimo numero di richieste inviabili al nodo da cui si è ricevuta l'informazione. I **limitOut** hanno quindi lo stesso contenuto informativo dei **limitIn**, ma visti dal punto di vista di chi deve inviare le richieste al posto di chi deve riceverle. Sia i **limitIn** che i **limitOut** sono espressi in termini di richieste al secondo (r/s).
3. **Fase 2.** Per ogni funzione nello stato di **overloaded** vengono calcolati quelli che sono i pesi di inoltro delle richieste verso gli altri nodi della rete, sulla base dei **limitOut** ricevuti. Vengono infine scritti i pesi, che formano una distribuzione di probabilità, nel file di configurazione di HAProxy.

Questa seconda fase, invece, viene eseguita solo per le funzioni nello stato di **overlaoded**.

Le fasi appena descritte sono state riassunte, se si vuole entrare nel dettaglio di come vengono effettuati i calcoli, avendo a disposizione anche un'esempio, è possibile fare riferimento a [1].

Altra caratteristica di questo algoritmo è quella che è stato implementato in modo tale che gli agenti lo eseguissero in maniera **sincrona**, ovvero che eseguissero le fasi

tutti contemporaneamente, sfruttando un **timestamp** di sistema. Ogni agente nel prototipo di DFaaS riesegue l'algoritmo una volta al minuto.

Anche in questo caso, per essere implementato nel simulatore, è stato necessario eseguire qualche adeguamento. Primo cambiamento attuato è relativo alla comunicazione globale; ogni nodo, infatti, è stato dotato del concetto di vicinato, come nella strategia empirica, e comunica solo con i nodi appartenenti ad esso.

### 5.5.2 Esperimento

Anche in questo caso, al fine di comparare i risultati ottenuti, viene eseguito l'esperimento sfruttando la medesima istanza di test utilizzata per la strategia **empirica**.

La tabella di inoltro delle richieste viene mostrata in Figura 5.18. Anche in questo caso, per quanto riguarda le tabelle dei max rate e degli invocation rate, rimangono valide quelle di Figura 5.9.

Tabella richieste inoltrate - funca

	<b>node_0</b>	<b>node_1</b>	<b>node_2</b>	<b>node_3</b>	<b>node_4</b>	<b>node_5</b>	<b>node_6</b>	<b>node_7</b>	<b>node_8</b>	<b>node_9</b>
<b>node_0</b>	30	0	5	0	0	0	0	0	4	0
<b>node_1</b>	0	100	70	7	0	0	0	0	0	0
<b>node_2</b>	0	0	9	0	0	0	0	0	0	0
<b>node_3</b>	0	0	0	19	0	0	0	0	0	0
<b>node_4</b>	0	0	30	0	100	0	0	28	0	0
<b>node_5</b>	0	0	0	2	0	30	0	0	7	0
<b>node_6</b>	0	0	0	15	0	0	100	0	62	0
<b>node_7</b>	0	0	0	0	0	0	0	9	0	0
<b>node_8</b>	0	0	0	0	0	0	0	0	16	0
<b>node_9</b>	0	0	0	0	0	0	0	0	0	0

Figura 5.18: Tabella delle richieste inoltrate sfruttando la **DFaaS Static** strategy, relativa alla funzione **funca**.

Come è possibile vedere in Figura 5.18, in questo caso l'agente in esecuzione sul nodo `_1`, a differenza delle due strategie di **baseline**, considera quelle che sono le informazioni ottenute dal vicinato. Difatti, inoltra le richieste in eccesso solamente ai nodi 2 e 3, e non ai nodi 4 (la funzione **funca** si trova in uno stato di sovraccarico) e 9 (non possiede repliche della funzione **funca**).

Un esempio di output dei pesi calcolati da un agente che esegue questo tipo di strategia è riportato in Figura 5.19. Questi sono i pesi per la funzione **funca**, che si trova in uno stato di **overloaded**, relativi all'agente in esecuzione sul nodo\_1.

```

3   ----- MINUTE 3 -----
4   | > STRATEGY: dfaas_static_strategy <
5   Weights normalized for func ocr: {'node_3': 0.0, 'node_2': 62.5, 'node_4': 37.49999999999999, 'node_9': 0.0}
6   Weights normalized for func funca: {'node_3': 9.01639344262295, 'node_2': 90.98360655737704, 'node_4': 0.0, 'node_9': 0.0}

```

Figura 5.19: Pesi calcolati dall'agente del nodo\_1 per la funzione **funca**.

Utilizzando questi pesi sono state inoltrate le richieste di invocazione della funzione **funca** dal nodo1\_1 verso i nodi 2 e 3, come mostrato in Figura 5.18. Il traffico, rappresentato dalle 77 r/s eccedenti in arrivo alla **funca** del nodo\_1, viene inoltrato verso il nodo 2 e il nodo 3, e così suddiviso:

- 70 r/s inoltrate al nodo\_2; queste rappresentano circa il 91% delle 77 r/s totali eccedenti (il peso di inoltro verso il nodo\_2 è del 91%).
- 7 r/s inoltrate al nodo\_3; queste rappresentano circa il 9% delle 77 r/s totali eccedenti (il peso di inoltro verso il nodo\_3 è del 9%).

La distribuzione del carico rispetta quindi quelli che sono i pesi calcolati dall'algoritmo.

Infine, anche in questo caso, viene mostrato il grafico prodotto dall'*analyzer*, che riporta l'andamento dell'indice **success rate** durante gli step di simulazione (vedi Figura 5.20).

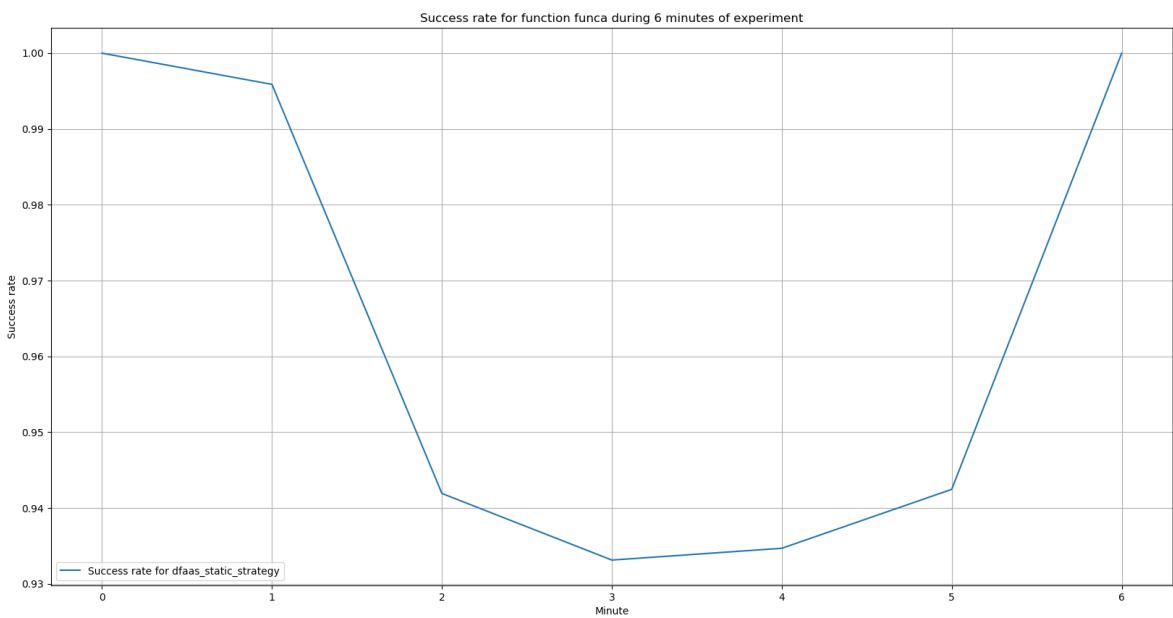


Figura 5.20: Andamento del **success rate** durante la simulazione per la funzione **funca**, applicando la **DFaaS Static strategy**.

## 5.6 Discussione e Confronto

La strategia empirica è stata messa a confronto con le altre tre implementate in diversi esperimenti e diverse configurazioni, in modo da validarne oltre che il funzionamento anche le performance.

Di seguito vengono riportati sullo stesso grafico quelli che sono i risultati ottenuti a partire dalla stessa istanza, la cui topologia viene raffigurata in Figura 5.8, utilizzando le diverse strategie descritte in questo capitolo. L’istanza è quella utilizzata negli esperimenti delle singole strategie per verificarne la correttezza e comprende la simulazioni di 10 agenti, ognuno dei quali, per ogni istante di simulazione eseguirà tutte le strategie, per metterne a confronto i risultati.

In Figura 5.21 viene riportato quello che è il confronto tra i **success rate** durante gli step di simulazione. Sullo stesso grafico, sono infatti mostrate le curve riportate in Figura 5.11, Figura 5.14, Figura 5.17 e Figura 5.20.

Come è possibile vedere in Figura 5.21 la strategia **empirica** e la **DFaaS static strategy**, si distaccano dalle due di **baseline** in termini di success rate. Queste due sono

inoltre molto vicine in termini di risultati, avendo una differenza, in tutte le simulazioni effettuate, di uno o due punti percentuali.

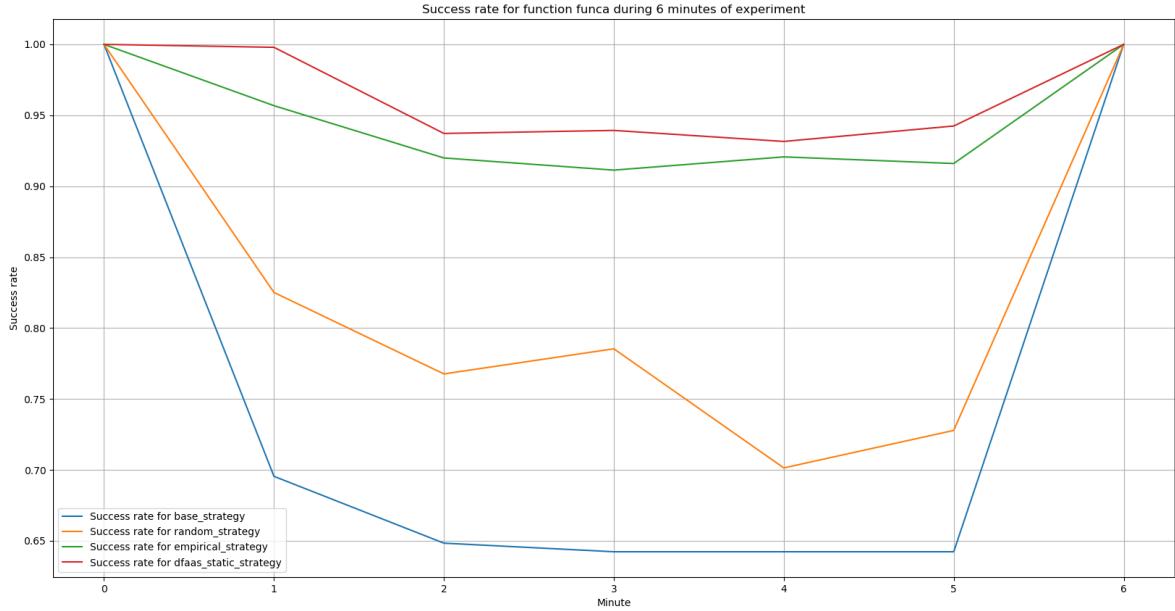


Figura 5.21: Andamento del **success rate** durante la simulazione per la funzione **funca**, applicando tutte le strategie.

Nella stessa istanza su cui è stata fatta la comparazione, è possibile confrontare i risultati in termini di **success rate** anche per quanto riguarda la distribuzione del carico sulle funzioni **qrcode** (vedi Figura 5.22) e **ocr** (vedi Figura 5.23).

In questi ultimi due grafici, ed in particolare in quello riguardante la funzione **qrcode**, è possibile vedere come l'algoritmo empirico e quello presente sul prototipo di DFaaS si equivalgano in termini di **success rate**.

Le strategie sono state comparate sfruttando gli indici prodotti dall'*analyzer* e le informazioni riassuntive prodotte dal *simulation controller*.

In Tabella 5.4 vengono riportate le informazioni riassuntive, riportando altri indici di confronto, mediati su tutte le funzioni, tra le diverse strategie. Come si può vedere la strategia empirica ha un success rate medio pari circa al 93% (durante il periodo di stress pari a 90%), mentre le strategie statiche di DFaaS pari al 94% (durante il periodo di stress pari a 92%). Le differenze sono quindi davvero minime.

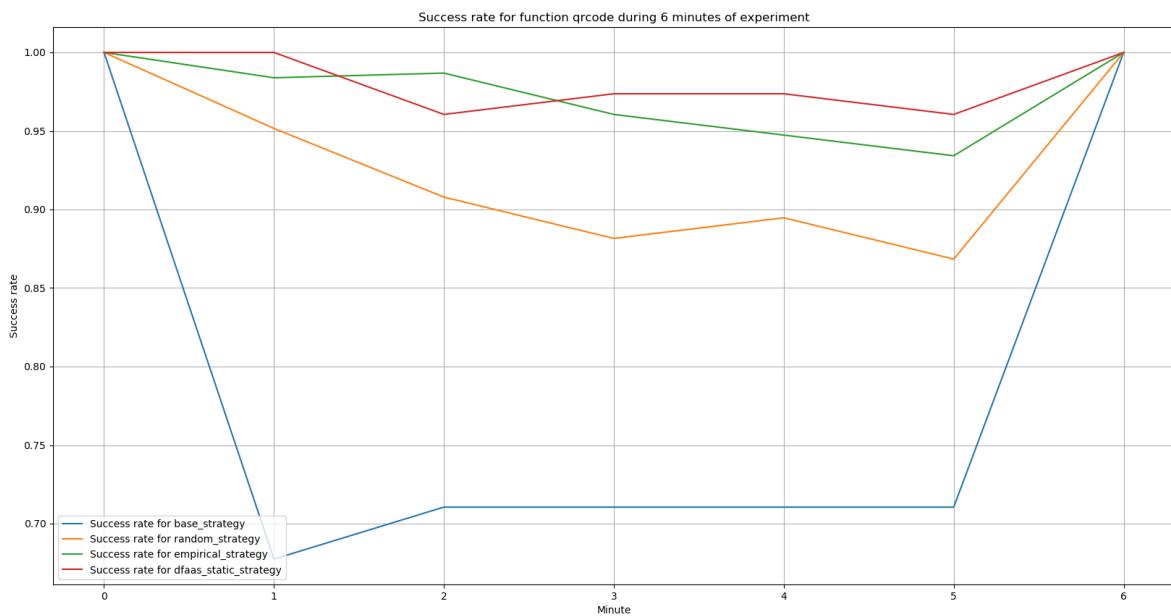


Figura 5.22: Andamento del **success rate** durante la simulazione per la funzione **qrcode**, applicando tutte le strategie.

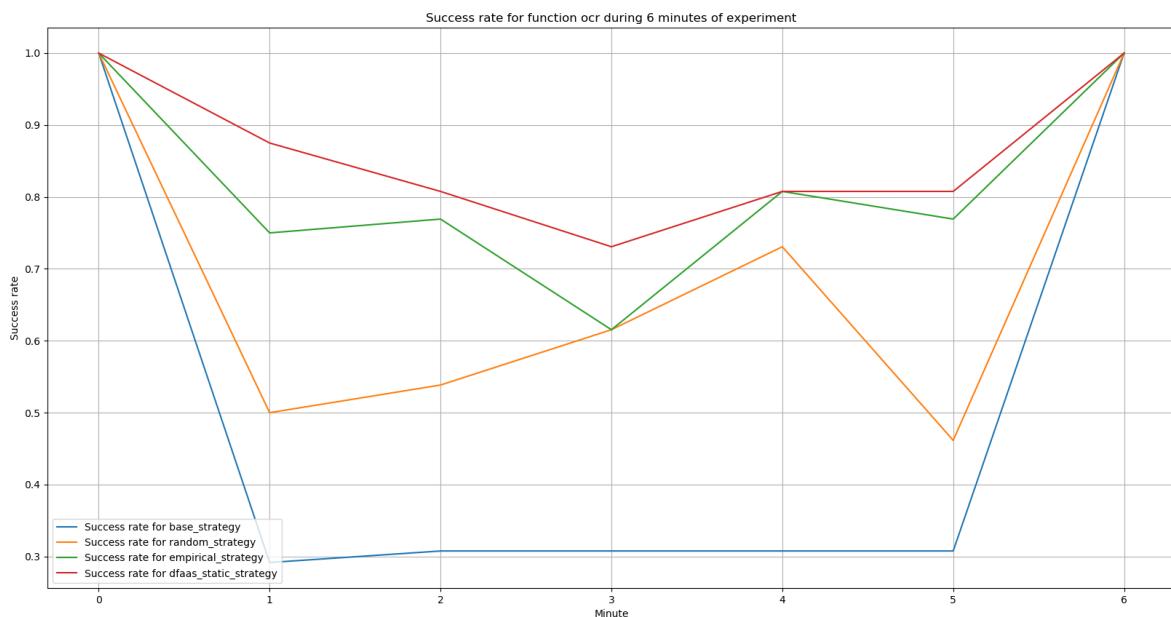


Figura 5.23: Andamento del **success rate** durante la simulazione per la funzione **ocr**, applicando tutte le strategie.

Questi punteggi si distaccano invece da quelle che sono le strategie di **baseline**; la base strategy raggiunge infatti un success rate, nel periodo di stress, pari al 55%, mentre quella casuale pari al 75%.

Per quanto riguarda le differenze in termini di numero di rigetti, anche in questo

Tabella 5.4: Comparazione delle diverse strategie sulla base delle informazioni riassuntive prodotte dal *simulation controller*, mediate su tutte le funzioni. SR rappresenta il **success rate**.

Strategies	Base	Random	Empirical	DFaaS static
<b>Mean SR</b>	68.18%	82.22%	92.63%	94.10%
<b>SR variance</b>	632.78	258.93	72.67	51.90
<b>Mean SR (stress period)</b>	55.42%	75.11%	89.69%	91.74%
<b>SR variance (stress period)</b>	318.02	184.97	70.99	52.96
<b>Tot reject number</b>	75540	46956	16560	11844

caso la strategia empirica (16560 rigetti) e quella di DFaaS (11844 rigetti) si distaccano di molto dalla base strategy (75540 rigetti) e dalla random strategy (46956 rigetti).

Anche in termini del percentile al 90% del numero di richieste rigettate si può vedere come la strategia empirica, con un valore di 3060 richieste, sia molto vicina alla DFaaS static strategy, con un valore di 2340 richieste. Entrambi di gran lunga inferiori a quelli delle base e random strategy, rispettivamente pari a 13800 e 8880 richieste.

Successivamente sono stati eseguite tante altre simulazioni variando numero di agenti e topologia di rete, ma i risultati ottenuti sono stati, in situazioni di congestione, molto simili a quelli sopra riportati.

Di seguito viene riportato un altro confronto, in termini di **success rate** della funzione **funca** (vedi Figura 5.24), eseguito su una topologia di rete con 25 agenti.

Da questo esperimento è possibile vedere ancora meglio come la strategia empirica e la strategia statica implementata su DFaaS siano simili in termini di performance. Difatti, la strategia empirica ha un success rate, mediato su tutte le funzioni, pari al 89.6%, mentre la strategia di DFaaS pari al 90.5%, con una differenza inferiore al punto percentuale. Questo implica che anche lo scarto in termini di richieste rigettate si è assottigliato.

La strategia empirica esegue la distribuzione del carico basandosi su diverse metriche, mentre la strategia implementata sul prototipo di DFaaS calcola i pesi basandosi solamente sui limiti calcolati, e quindi indirettamente sul margine di richieste che ogni funzione può ancora servire. Per mostrare la flessibilità della strategia empirica si è deciso di rieseguire il medesimo esperimento con 25 agenti ma cambiando i pesi delle

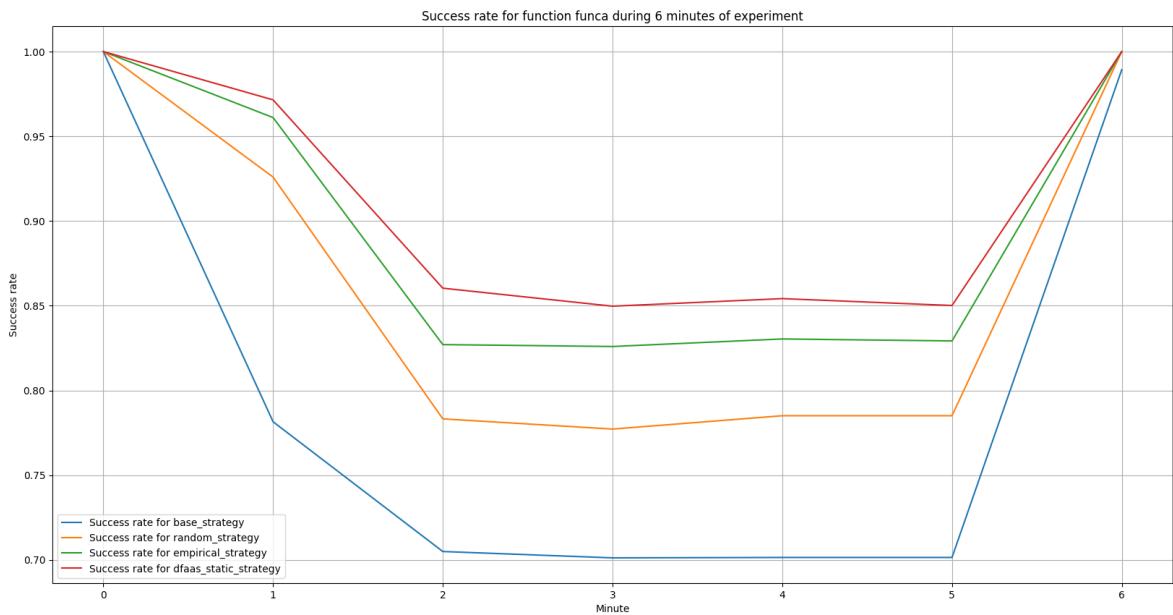


Figura 5.24: Andamento del **success rate** durante la simulazione per la funzione **funca**, applicando tutte le strategie.

metriche considerate da questo algoritmo. E' stato infatti modificato il dizionario dei pesi della strategia empirica, facendogli considerare solamente la metrica del **margine**, lasciando il corrispondente peso a uno, e mettendo tutti gli altri a zero.

Il risultato ottenuto è quello che ci si poteva aspettare, ovvero che la strategia empirica si comportasse in maniera ancora più simile a quella statica implementata su DFaaS. Riproducendo l'esempio con i 25 agenti il grafico ottenuto per quanto riguarda la funzione **funca** è quello riportato in Figura 5.25. Come si può notare, la strategia empirica eguaglia quella di DFaaS, con un success rate medio pari a 94.5% contro il 94.3%.

Questo ulteriore esperimento ha consentito di mostrare la flessibilità della strategia **empirica** che tiene conto di una moltitudine di metriche, ad ognuna delle quali attribuisce un peso. Questi pesi si prestano per essere ottimizzabili a seconda del tipo di bilanciamento del carico si vuole ottenere, come ad esempio una distribuzione delle richieste per ottimizzare l'utilizzo di CPU o di RAM e così via. Seppur in questo lavoro di tesi i pesi sono stati tutti lasciati uguali (tutti pari ad uno), e quindi le metriche pesate allo stesso modo, questi potrebbero anche essere calcolati dinamicamente da un

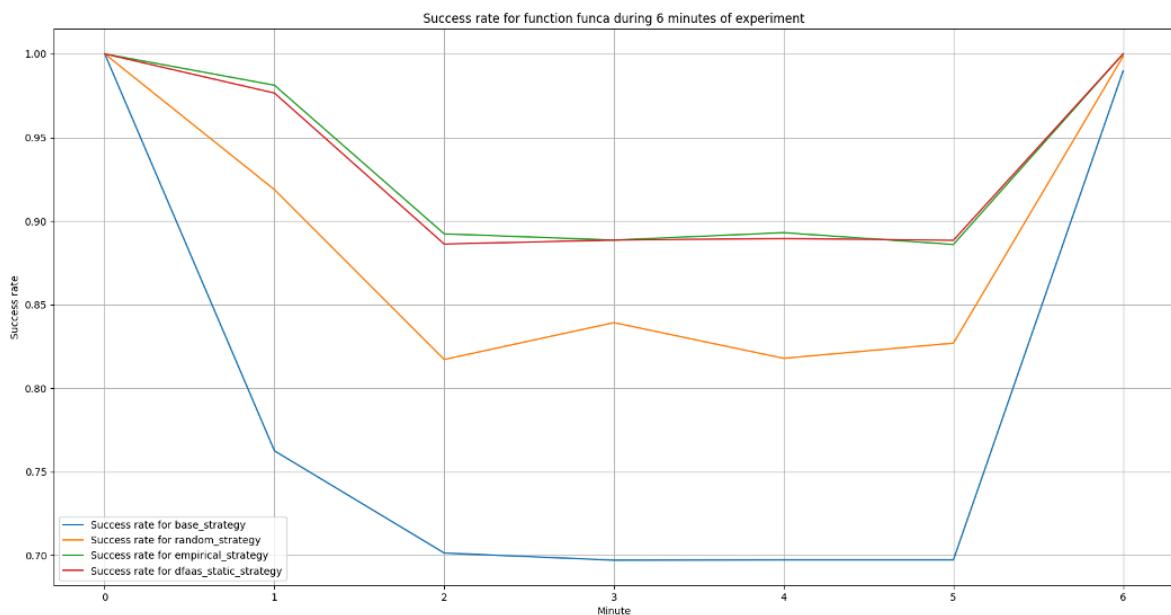


Figura 5.25: Andamento del **success rate** durante la simulazione per la funzione **funca**, applicando tutte le strategie, dopo aver cambiato i pesi associati alle metriche per la strategia **empirica**.

modello di Machine Learnig.

La **DFaaS Static** strategy potrebbe sembrare che in termini di success rate risulti leggermente migliore in alcuni dei grafici riportati, ma questo è dovuto semplicemente al fatto che gli indici utilizzati principalmente ai fini della valutazione sono quelle relativi al success rate o al numero di rigetti effettuati. La strategia di DFaaS, infatti, per eseguire la distribuzione del carico tiene conto solo di rispettare i max rate imposti dalle varie funzioni, non considerando in nessun modo lo stato del nodo su cui l'agente è in esecuzione. Oltretutto non vengono utilizzati indici che riguardino l'utilizzo delle risorse sul nodo, che invece, potrebbero favorire la strategia **empirica**.

Infine, è possibile concludere, che la strategia **empirica** risulti essere migliore rispetto gli algoritmi di baseline ed inoltre che superi in termini di flessibilità la DFaaS static strategy, seppur i risultati siano pressoché identici.

# Capitolo 6

## Conclusioni

Il problema trattato in questo lavoro di tesi è quello del bilanciamento del carico di richieste di invocazioni di funzioni a piattaforme di FaaS, in un ambiente fortemente dinamico ed eterogeneo come quello dell'Edge Computing.

In un ambiente complesso e imprevedibile come questo è stato progettato un algoritmo distribuito di bilanciamento del carico, che persegue questo obiettivo basandosi sulle informazioni scambiate con gli altri agenti in esecuzione sui nodi appartenenti al sistema. L'algoritmo prende il nome di **empirical strategy** e viene eseguito periodicamente da ogni agente, calcolando i pesi di inoltro per ogni funzione nello stato di sovraccarico, verso tutti i nodi raggiungibili in grado di fornire un aiuto nell'esecuzione delle richieste per quella determinata funzione. Gli agenti si scambiano informazioni riguardanti lo stato del nodo, sintetizzato per mezzo di metriche raccolte ciclicamente durante una fase di monitoraggio dello stato del sistema.

L'algoritmo progettato, e descritto in Sezione 5.2, ha raggiunto i risultati desiderati, superando di gran lunga le performance degli algoritmi di baseline utilizzati ai fini del confronto, ed eguagliando quelle dell'algoritmo precedentemente implementato su DFaaS [1], ma consentendo una maggiore flessibilità ed espandibilità. DFaaS [1] rappresenta un'architettura decentralizzata, presente in letteratura, basata su FaaS e progettata per bilanciare il carico di lavoro tra nodi edge federati. Tale sistema è stato utilizzato come base per la raccolta di dati reali, con i quali simulare poi gli

algoritmi implementati, descritti nel Capitolo 5, e ne è stata utilizzata la strategia di distribuzione in esso implementata come metodo di confronto con l'algoritmo ideato.

Prima di riportare l'algoritmo ideato sul sistema di DFaaS [1], la correttezza della **empirical strategy** è stata testata realizzando un simulatore, ovvero un componente software in grado di astrarre dalle specifiche a basso livello, come ad esempio la comunicazione tra gli agenti, per concentrarsi sulla validità dei calcoli eseguiti dall'algoritmo. Questo simulatore è stato scritto in maniera modulare, generica ed estendibile, consentendone il riutilizzo per l'implementazione di altri algoritmi ed il confronto con quelli esistenti. Punto di forza di questa trattazione è proprio la generalità del simulatore che ha consentito, in fase di sviluppo, di aggiungere algoritmi di distribuzione del carico in maniera molto semplice.

Il simulatore è inoltre dotato di un database che contiene i dati sperimentalmente raccolti da situazioni reali sul sistema DFaaS in essere, secondo la metodologia descritta in Sezione 5.1.2. Il simulatore utilizza il database per reperire dinamicamente le configurazioni da simulare, anch'esse generate da un'apposita componente.

Sulla base dei risultati ottenuti dal simulatore, ed esposti nel Capitolo 5, è stato possibile decretare raggiunto l'obiettivo inizialmente prefissato, ovvero quello di ideare un algoritmo distribuito che basasse il bilanciamento del carico su una serie di evidenze raccolte dal nodo, e che sintetizzassero il suo stato di salute.

Sulla base di queste considerazioni si è quindi implementata una prima versione semplice e perfettibile dell'algoritmo sul prototipo di DFaaS, che ne dimostra l'effettivo funzionamento, nonostante si presti ad essere migliorata ed ottimizzata.

Per quanto riguarda possibili sviluppi futuri, una prima idea potrebbe essere quella di indagare ulteriormente la flessibilità della strategia empirica ideata. Questo potrebbe essere fatto pesando diversamente le metriche utilizzate per il calcolo dei pesi di inoltro a seconda dei fini che si vogliono perseguire. Sarebbe ad esempio possibile ricercare la combinazione ideale di pesi per le metriche al fine di ottenere un bilanciamento orientato all'ottimizzazione dell'utilizzo di RAM o di CPU; oppure sarebbe possibile variare questi pesi dinamicamente, facendoli calcolare da un algoritmo di Machine Learning.

Inoltre, si potrebbe espandere il database contenente le informazioni degli esperimenti eseguiti, qualora si decidesse di eseguire un'altra campagna di raccolta dati sul prototipo di DFaaS.

Infine, ultimo spunto per possibili futuri lavori, è la possibilità di implementare ulteriori algoritmi di distribuzione del carico, al fine di testarli anch'essi in un ambiente isolato, e validarne il funzionamento a confronto con quelli già esistenti.

# Bibliografia

- [1] M. Ciavotta, D. Motterlini, M. Savi, and A. Tundo, “Dfaas: Decentralized function-as-a-service for federated edge computing,” in *2021 IEEE 10th International Conference on Cloud Networking (CloudNet)*, pp. 1–4, 2021.
- [2] C. Cicconetti, M. Conti, and A. Passarella, “A decentralized framework for serverless edge computing in the internet of things,” *IEEE Transactions on Network and Service Management*, vol. 18, no. 2, pp. 2166–2180, 2021.
- [3] C. Cho, S. Shin, H. Jeon, and S. Yoon, “Qos-aware workload distribution in hierarchical edge clouds: A reinforcement learning approach,” *IEEE Access*, vol. 8, pp. 193297–193313, 2020.
- [4] UNIMIBInside, “Dfaas: Decentralized function-as-a-service for federated edge.” <https://github.com/UNIMIBInside/dfaas>. Accessed on 14-02-2022.
- [5] D. c. Saraswathi AT a, Kalaashri.Y.RA b, “Dynamic resource allocation scheme in cloud computing,” in *Procedia Computer Science*, vol. 47, pp. 30–36, 2015.
- [6] Gartner, “Public cloud computing.” <https://www.gartner.com/en/information-technology/glossary/public-cloud-computing>. Accessed on 29-12-2021.
- [7] AWS, “Cos’è il cloud computing?” <https://aws.amazon.com/it/what-is-cloud-computing/>. Accessed on 29-12-2021.
- [8] RedHat, “Cos’è il serverless computing?” <https://www.redhat.com/it/topics/cloud-native-apps/what-is-serverless>. Accessed on 29-12-2021.

- [9] Cloudflare, “Cos’è il computing serverless?.” <https://www.cloudflare.com/it-it/learning/serverless/what-is-serverless/>. Accessed on 29-12-2021.
- [10] Mike Roberts, “Serverless Architectures.” <https://martinfowler.com/articles/serverless.html>. Accessed on 29-12-2021.
- [11] Google Firebase, “Firebase helps you build and run successful apps.” <https://firebase.google.com/>. Accessed on 29-01-2021.
- [12] IONOS, “Sviluppare un back end personalizzato con il backend as a service.” <https://www.ionos.it/digitalguide/server/know-how/backend-as-a-service-baas/>. Accessed on 29-01-2021.
- [13] RedHat, “Cos’è il FaaS (Function-as-a-Service) computing?.” <https://www.redhat.com/it/topics/cloud-native-apps/what-is-faas>. Accessed on 03-01-2022.
- [14] RedHat, “Cos’è la containerizzazione?.” <https://www.redhat.com/it/topics/cloud-native-apps/what-is-containerization>. Accessed on 01-01-2022.
- [15] Docker, “Docker.” <https://www.docker.com/>. Accessed on 02-01-2022.
- [16] Amazon, “AWS Lambda.” <https://aws.amazon.com/it/lambda/>. Accessed on 01-01-2022.
- [17] Google, “Cloud Functions.” <https://cloud.google.com/functions>. Accessed on 01-01-2022.
- [18] OpenFaaS, “Serverless Functions, Made Simple.” <https://www.openfaas.com/>. Accessed on 01-01-2022.
- [19] Kubeless, “Kubeless..” <https://kubeless.io/>. Accessed on 01-01-2022.
- [20] Microsoft Azure, “Funzioni di Azure.” <https://azure.microsoft.com/it-it/services/functions/#overview>. Accessed on 03-01-2022.

- [21] IBM, “What is edge computing?.” <https://www.ibm.com/cloud/what-is-edge-computing>. Accessed on 07-01-2022.
- [22] Maurizio Di Paolo Emilio, “Fog o Edge Computing? L’evoluzione del Cloud per l’IoT.” <https://www.innovationpost.it/2018/03/08/fog-edge-computing-levoluzione-del-cloud-per-liot/>. Accessed on 03-01-2022.
- [23] Cisco, “Che cos’è l’edge computing?.” [https://www.cisco.com/c/it\\_it/solutions/computing/what-is-edge-computing.html#~tipi-di-edge-computing](https://www.cisco.com/c/it_it/solutions/computing/what-is-edge-computing.html#~tipi-di-edge-computing). Accessed on 07-01-2022.
- [24] Adriano Di Stasi, “Il fog computing nell’era del cloud e dell’Internet of Things.” <https://www.industry4business.it/cloud/il-fog-computing-nellera-del-cloud-e-dellinternet-of-things/>. Accessed on 07-01-2022.
- [25] Hewlett Packard Enterprise, “COS’È L’EDGE COMPUTING?.” <https://www.hpe.com/it/it/what-is/edge-computing.html>. Accessed on 07-01-2022.
- [26] RedHat, “Cos’è l’architettura edge?.” <https://www.redhat.com/it/topics/edge-computing/what-is-edge-architecture>. Accessed on 10-02-2022.
- [27] Gartner, “What Edge Computing Means for Infrastructure and Operations Leaders.” <https://www.gartner.com/smarterwithgartner/what-edge-computing-means-for-infrastructure-and-operations-leaders>. Accessed on 07-01-2022.
- [28] RedHat, “I vantaggi dell’edge computing.” <https://www.redhat.com/it/topics/edge-computing/what-is-edge-computing#edge-analisi-dei-dati-e-ia/ml>. Accessed on 07-01-2022.
- [29] Cabe Atwell, “What is Edge Machine Learning?.” <https://www.fierceelectronics.com/electronics/what-edge-machine-learning>. Accessed on 07-01-2022.

- [30] Microsoft, “Resource-efficient ML for Edge and Endpoint IoT Devices.” <https://www.microsoft.com/en-us/research/project/resource-efficient-ml-for-the-edge-and-endpoint-iot-devices/>. Accessed on 07-01-2022.
- [31] Andrew Overheid, “Understanding edge computing vs. fog computing.” <https://www.onlogic.com/company/io-hub/edge-computing-vs-fog-computing/>. Accessed on 10-02-2022.
- [32] HAProxy, “HAProxy: The Reliable, High Performance TCP/HTTP Load Balancer.” <http://www.haproxy.org/>. Accessed on 07-01-2022.
- [33] T. Pfandzelter and D. Bermbach, “tinyfaas: A lightweight faas platform for edge environments,” in *2020 IEEE International Conference on Fog Computing (ICFC)*, pp. 17–24, 2020.
- [34] Internet Engineering Task Force (IETF), “Rfc 7252 - the constrained application protocol (coap).” <https://datatracker.ietf.org/doc/html/rfc7252>. Accessed on 05-02-2022.
- [35] H. Jeon, C. Cho, S. Shin, and S. Yoon, “A cloudsim-extension for simulating distributed functions-as-a-service,” in *2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 386–391, 2019.
- [36] Cloud Computing and Distributed Systems (CLOUDS) Laboratory, School of Computing and Information Systems, The University of Melbourne, Australia, “Cloudsim: A framework for modeling and simulation of cloud computing infrastructures and services.” <http://www.cloudbus.org/cloudsim/>. Accessed on 07-02-2022.
- [37] Python, “Python.” <https://www.python.org/>. Accessed on 19-01-2022.
- [38] SQLite, “SQLite.” <https://www.sqlite.org/index.html>. Accessed on 14-01-2022.

- [39] Refactoring.Guru, “Factory Method.” <https://refactoring.guru/design-patterns/factory-method>. Accessed on 27-01-2022.
- [40] Refactoring.Guru, “Strategy.” <https://refactoring.guru/design-patterns/strategy>. Accessed on 27-01-2022.
- [41] L. A. Borgida A., Casanova M.A., “Logical database design: from conceptual to logical schema,” in *LIU L., ÖZSU M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA*, 2009.
- [42] Refactoring.Guru, “Singleton.” <https://refactoring.guru/design-patterns/singleton>. Accessed on 27-01-2022.
- [43] OpenFaaS, “Monitoring Functions.” <https://docs.openfaas.com/architecture/metrics/>. Accessed on 20-01-2022.
- [44] OpenFaaS, “OpenFaaS stack.” <https://docs.openfaas.com/architecture/stack/>. Accessed on 20-01-2022.
- [45] Prometheus, “Exporters and Integrations.” <https://prometheus.io/docs/instrumenting/exporters/>. Accessed on 20-01-2022.
- [46] Prometheus, “Node exporter.” [https://github.com/prometheus/node\\_exporter](https://github.com/prometheus/node_exporter). Accessed on 20-01-2022.
- [47] cAdvisor, “Google.” <https://github.com/google/cadvisor>. Accessed on 20-01-2022.
- [48] Docker, “Overview of Docker Compose.” <https://docs.docker.com/compose/>. Accessed on 20-01-2022.
- [49] Docker, “Swarm mode overview.” <https://docs.docker.com/engine/swarm/>. Accessed on 20-01-2022.
- [50] RedHat, “Red Hat Ansible Automation Platform.” <https://www.ansible.com/>. Accessed on 20-01-2022.

- [51] RedHat, “I concetti base di Ansible.” <https://www.redhat.com/it/topics/automation/learning-ansible-tutorial>. Accessed on 20-01-2022.
- [52] Prometheus, “HTTP API.” <https://prometheus.io/docs/prometheus/latest/querying/api/>. Accessed on 21-01-2022.
- [53] Google, “Go: Build fast, reliable, and efficient software at scale.” <https://go.dev/>. Accessed on 21-01-2022.
- [54] faas-and-furious, “FaaS qrcode.” <https://github.com/faas-and-furious/qrcode>. Accessed on 21-01-2022.
- [55] OpenFaaS, “OpenFaaS Function Store.” <https://github.com/openfaas/store>. Accessed on 21-01-2022.
- [56] viveksyng, “OpenFaaS ocr.” <https://github.com/viveksyng/openfaas-ocr>. Accessed on 21-01-2022.
- [57] Tomás Senart, “Vegeta.” <https://github.com/tsenart/vegeta>. Accessed on 21-01-2022.
- [58] E. Nitto, L. Florio, and D. Tamburri, *Autonomic Decentralized Microservices: The Gru Approach and Its Evaluation*, pp. 209–248. 12 2019.
- [59] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, Mary Shaw, *Engineering Self-Adaptive Systems through Feedback Loops*. Springer, Berlin, Heidelberg. Accessed on 12-01-2022.