Università di Pisa

Dipartimento di Ingegneria dell'Informazione

Laurea Triennale in Ingegneria Informatica

Realizzazione di un framework Function as a Service in Python per il test rapido di algoritmi di scheduling

Candidato
Tommaso Molesti

Relatori
Carlo Vallati
Francesca Righetti
Giuseppe Anastasi

9 Ottobre 2025

Contents

1 Abstract

L'evoluzione del Cloud Computing ha consolidato il paradigma Serverless e il modello Function as a Service (FaaS in seguito) come approcci dominanti per lo sviluppo di applicazioni scalabili. Tuttavia, le performance di questi sistemi sono legate a due sfide: la latenza introdotta dal fenomeno del "cold start" e l'efficienza degli algoritmi di scheduling nel distribuire il carico di lavoro. La valutazione di diverse strategie per la gestione dei container e delle risorse in ambienti controllati e riproducibili è fondamentale per l'ottimizzazione di tali piattaforme.
Questa tesi affronta questa esigenza attraverso la progettazione e l'implementazione di un framework FaaS in Python, concepito come un banco di prova (testbed) per l'analisi comparativa di algoritmi di scheduling. L'architettura del sistema si basa su un Gateway API centrale, sviluppato con FastAPI, che orchestra un cluster di nodi per l'esecuzione di funzioni, con i quali comunica tramite SSH. Le funzioni utente vengono eseguite come container Docker, permettendo di simulare scenari realistici. Un elemento chiave è la modularità, che consente di integrare e confrontare dinamicamente diverse policy di scheduling e strategie di gestione del ciclo di vita dei container, implementando e analizzando le modalità di invocazione Cold, Pre-warmed e Warmed.
L'analisi sperimentale condotta per testare il sistema sviluppato ha permesso di quantificare l'impatto di ciascuna strategia sulla latenza di esecuzione e sull'utilizzo delle risorse (CPU e RAM). I risultati dimostrano come le tecniche di warming possano abbattere drasticamente i tempi di risposta e come la scelta di un algoritmo di scheduling consapevole dello stato dei nodi influenzi il bilanciamento del carico. Il framework sviluppato si è dimostrato uno strumento flessibile e riproducibile, che può facilitare lo studio e l'ottimizzazione delle performance nei sistemi FaaS.

2 Introduzione

2.1 L'evoluzione del Cloud Computing e il paradigma Serverless

Il Cloud Computing ha rivoluzionato il mondo dell'informatica, offrendo un'alternativa flessibile all'approccio tradizionale. In passato, per creare un'applicazione, un'azienda doveva acquistare e mantenere fisicamente i propri server, un processo costoso e rigido noto come infrastruttura on-premise. Il cloud ha introdotto un nuovo modello: le risorse di calcolo, come server, spazio di archiviazione e reti, sono diventate un servizio accessibile su richiesta tramite Internet, con un modello di costo basato sull'effettivo consumo (pay-per-use).
Questo cambiamento si è rivelato vantaggioso per aziende di ogni dimensione, dalle startup alle multinazionali, grazie alla sua grande flessibilità. Invece di attendere settimane per l'acquisto di un nuovo server, un team di sviluppo può ottenerne uno virtuale in pochi minuti. Questo è possibile perché i fornitori di servici cloud gestiscono enormi quantità di risorse condivise. Quando un utente ha bisogno di un server, questo gli viene assegnato automaticamente sfruttando le risorse dell'infrastruttura comune.
L'ultimo grande passo in questa direzione è il paradigma Serverless. L'idea di base è portare l'astrazione ad un livello superiore: eliminare completamente la necessità per gli sviluppatori di pensare ai server. Con il Serverless, il codice viene eseguito solo quando si verifica un determinato evento (come un click di un utente su un'app). Le risorse necessarie vengono create al volo, usate per il tempo strettamente necessario, e poi eliminate. Questo approccio promette di ridurre ulteriormente i costi e di aumentare l'efficienza, lasciando agli sviluppatori un unico compito: concentrarsi sulla scrittura della logica che fa funzionare la loro applicazione.

2.2 Il modello Function as a Service e le differenze con Serverless

Il modo più comune per mettere in pratica la filosofia Serverless è attraverso il modello FaaS. Piattaforme molto note come Amazon AWS Lambda, Google Cloud Functions e IBM Cloud Functions sono esempi di servizi FaaS.
Con questo approccio, un'applicazione non viene più vista come un unico grande blocco di codice che gira costantemente su un server, ma viene spezzettata in tante piccole "funzioni" indipendenti. Ogni funzione è un pezzetto di codice specializzato in un unico compito, che vive all'interno di un contenitore software leggero e isolato. Quando un evento richiede l'esecuzione di quel compito, la piattaforma FaaS si occupa di tutto: avvia il contenitore con la funzione, esegue il codice e, una volta ottenuto il risultato, spegne tutto.
I vantaggi di questo modello sono:

- Velocità di sviluppo: gli sviluppatori possono scrivere e aggiornare piccole funzioni in modo molto più rapido rispetto ad un'intera applicazione.
- Nessuna gestione dei server: la responsabilità di mantenere, aggiornare e proteggere i server è interamente delegata al fornitore del servizio.
- Scalabilità automatica: se una funzione viene chiamata migliaia di volte al secondo, la piattaforma crea automaticamente migliaia di copie per gestire il carico, senza alcun intervento manuale.
- Costi: si paga solo per il tempo in cui il codice è effettivamente in esecuzione.
- Flessibilità: è possibile scrivere ogni funzione nel linguaggio di programmazione più adatto a svolgere quel compito specifico.

Inizialmente, i termini FaaS e Serverless erano quasi sinonimi. Con il tempo il concetto di "Serverless" si è allargato. Oggi, FaaS è considerato il motore di calcolo del mondo Serverless, ma non è l'unico componente. Il termine "Serverless" include anche altri servizi completamente gestiti, come database, sistemi di messaggistica o piattaforme di storage, in cui l'utente non deve preoccuparsi di gestire alcun server. In sintesi, si può costruire un'intera architettura Serverless combinando diverse funzioni FaaS con altri servizi gestiti.

2.3 Scheduling in ambienti FaaS

Una volta che una piattaforma FaaS riceve la richiesta di eseguire una funzione, deve prendere una decisione fondamentale: su quale dei tanti server disponibili nel cloud verrà effettivamente eseguito quel codice? Il componente software che prende questa decisione è chiamato scheduler.
Il compito principale dello scheduler è indirizzare ogni singola richiesta di esecuzione verso il nodo di calcolo più appropriato in quel preciso istante. La scelta non è casuale, ma guidata da alcuni obiettivi che determinano l'efficienza della piattaforma:

- Minimizzare la latenza: l'obiettivo più importante per l'utente finale è ottenere una risposta nel minor tempo possibile. Uno scheduler efficace cerca di inviare il lavoro ad un nodo che può eseguirlo velocemente.
- Massimizzare l'uso delle risorse: dal punto di vista del fornitore del servizio cloud, è fondamentale che i server non rimangano inattivi se c'è del lavoro da fare, né che vengano sovraccaricati. Lo scheduler deve bilanciare il carico in modo intelligente per usare le risorse in modo efficiente.
- Fairness: in un sistema multi-utente, lo scheduler deve assicurarsi che nessuna richiesta venga "dimenticata" o penalizzata a favore di altre.

In ambienti FaaS ci sono anche altre sfide da affrontare, quella più grande è sicuramente la gestione del "cold start". Tipicamente, quando una specifica funzione Serverless non viene invocata per un certo periodo, il provider cloud distrugge l'ambiente in cui è in esecuzione per ottimizzare i costi, risparmiare energia ed evitare di allocare inutilmente delle risorse. Questa ottimizzazione ovviamente ha una conseguenza diretta sulle performance. Quando un utente successivo richiama quella funzione, la piattaforma deve avviare da zero un nuovo ambiente di esecuzione per ospitarla. Questo tempo di avvio, che non è trascurabile, introduce una latenza significativa nella risposta, un ritardo che prende appunto il nome di "cold start".
Uno scheduler FaaS non deve quindi solo considerare quanto sono carichi i vari server (in termini di CPU e RAM), ma deve anche sapere lo stato di preparazione di una funzione su un determinato nodo. Deve sapere se esiste già un container "caldo" (warmed) e pronto per l'esecuzione immediata, o se un nodo si trova in uno stato intermedio "pre-riscaldato" (pre-warmed), con l'immagine della funzione già scaricata in cache ma con il container non ancora attivo. La scelta dell'algoritmo di scheduling è un fattore critico che impatta direttamente sulle performance, sui costi e sull'esperienza utente di una piattaforma FaaS.

2.4 Obiettivi

L'obiettivo principale di questo lavoro di tesi è la progettazione e la realizzazione di un framework FaaS, concepito come un banco di prova per l'analisi e il confronto delle performance di diversi algoritmi di scheduling, in un ambiente controllato e riproducibile.
Lo scopo è quello di sviluppare uno strumento che permetta di studiare come diverse strategie di gestione delle risorse e di distribuzione del carico influenzino le performance del sistema. Per raggiungere questo scopo, sono stati definiti i seguenti sotto-obiettivi:

- Framework modulare: costruire un'architettura funzionante, composta da un Gateway centrale e da nodi di esecuzione distribuiti, con un design che permetta di "montare" e sostituire facilmente diverse strategie di scheduling, come se fossero dei moduli intercambiabili.
- Quantificare l'impatto del cold start: implementare e analizzare le tre diverse modalità di gestione dei container (Cold, Pre-warmed e Warmed) per misurare in modo concreto e numerico la latenza introdotta dal cold start e l'efficacia delle strategie di pre-riscaldamento.
- Confrontare algoritmi di scheduling: implementare e mettere a confronto diverse policy di scheduling per valutarne l'efficacia nel bilanciamento del carico.
- Sviluppare un sistema di misurazione e analisi: creare un meccanismo automatico per la raccolta di metriche di performance chiave (tempo di esecuzione, utilizzo di CPU e RAM) e per la generazione di report e grafici che facilitino l'interpretazione e la discussione dei risultati ottenuti.

3 Stato dell'arte

3.1 Tecnologie di containerizzazione: Docker

Un container è un'unità standard di software che impacchetta il codice di un'applicazione, insieme a tutte le sue dipendenze (come librerie e file di configurazione), garantendo che funzioni in modo rapido e affidabile in qualsiasi ambiente di calcolo. La tecnologia più diffusa per creare e gestire i container è Docker.
L'ecosistema Docker distingue due concetti principali: l'immagine e il container. Un'immagine è un pacchetto eseguibile, statico e autonomo, che contiene tutti gli elementi necessari per l'esecuzione di un'applicazione, come codice, runtime e librerie di sistema. Un container è l'istanza in esecuzione di un'immagine, esso viene creato e gestito dal Docker Engine.
La containerizzazione assicura che il software si comporti in modo uniforme e prevedibile, indipendentemente dall'infrastruttura sottostante. I container isolano l'applicazione dal suo ambiente operativo, garantendo coerenza tra gli ambienti di sviluppo, di test e di produzione.
Il paradigma dei container si differenzia in modo sostanziale da quello delle Macchine Virtuali (VM). Mentre le VM si basano sulla virtualizzazione dell'hardware, richiedendo un sistema operativo ospite completo per ogni istanza, i container si basano sulla virtualizzazione a livello di sistema operativo, condividendo il kernel del sistema ospitante. Questa differenza architetturale fondamentale rende i container più leggeri, portabili ed efficienti, con tempi di avvio inferiori.

3.2 Problematiche di performance in FaaS: latenza e cold start

Una delle principali problematiche prestazionali nelle piattaforme FaaS è la latenza di avvio (startup latency). Prima di poter processare un evento, la piattaforma impiega un certo tempo per inizializzare l'istanza della funzione. Questa latenza non è costante, ma può variare in modo significativo, da pochi millisecondi a diversi secondi, a seconda di diversi fattori.
Serve distinguere due scenari di inizializzazione: il "warm start" e il "cold start". Si parla di warm start (avvio a caldo) quando la piattaforma riutilizza un'istanza della funzione e il suo container, mantenuti attivi da un evento precedente. In questo caso, la latenza è minima. Il problema principale risiede invece nel cold start (avvio a freddo), che si verifica quando è necessario creare da zero un nuovo container, avviare il processo host della funzione e caricare il codice. Questo processo introduce un ritardo considerevole che costituisce la preoccupazione maggiore in termini di performance.
La durata effettiva di un cold start può dipendere da molte variabili, alcune delle quali sono sotto il controllo dello sviluppatore. Tra i fattori più influenti vi sono il numero e la dimensione delle librerie importate, la complessità del codice e la configurazione specifica dell'ambiente della funzione, come la memoria allocata.
Un fattore comunque critico è la frequenza con cui si verifica un cold start. Questa è direttamente legata al volume e alla tipologia del traffico dell'applicazione. Un'applicazione che processa un flusso costante di eventi manterrà le sue istanze "calde", facendo un cold start molto raramente. Al contrario, una funzione che viene invocata sporadicamente, subirà quasi certamente un cold start ad ogni invocazione, perché i provider FaaS disattivano le istanze inattive dopo alcuni minuti per ottimizzare l'allocazione delle risorse.
L'impatto dei cold start dipende sempre dal contesto e dal caso d'uso. Per sistemi asincroni ad alto volume, che processano grandi quantità di dati in background, la latenza iniziale di alcune invocazioni potrebbe essere del tutto trascurabile. Al contrario, per servizi sincroni e interattivi che richiedono risposte a bassa latenza, come le API per applicazioni web o mobile, anche un piccolo ritardo può degradare significativamente il servizio e quindi l'esperienza utente.
La latenza di avvio rappresenta un compromesso importante nel modello FaaS. La necessità di testare le performance con carichi di lavoro realistici è quindi fondamentale per determinare la fattibilità di una soluzione Serverless per uno specifico caso d'uso.

3.3 Strategie di gestione degli stati

Nel tempo sono state sviluppate diverse strategie per gestire il ciclo di vita degli ambienti di esecuzione (tipicamente container). Queste strategie rappresentano diversi compromessi

tra l'ottimizzazione dei costi e la riduzione della latenza. Si possono identificare tre stati principali in cui un'istanza di funzione può trovarsi: Cold, Warm e Pre-warmed.

3.3.1 Lo stato Cold (Avvio a freddo)

Lo stato Cold rappresenta la condizione di partenza di una funzione che non è stata invocata di recente. Quando arriva una richiesta per una funzione in questo stato, la piattaforma FaaS deve eseguire una serie di passaggi prima di poter processare l'evento. Questo processo include il provisioning di un nuovo container, l'inizializzazione del runtime e il caricamento del codice della funzione e delle sue dipendenze. L'intera sequenza introduce una latenza significativa, nota come "cold start". Questo approccio è il più efficiente dal punto di vista dei costi, poiché le risorse vengono allocate solo quando strettamente necessario e non si paga per i periodi di inattività, ma offre le performance peggiori in termini di latenza.

3.3.2 Lo stato Warm (Avvio a caldo)

Dopo aver servito un'invocazione, la piattaforma FaaS può decidere di mantenere l'ambiente di esecuzione attivo per un breve periodo di tempo, in attesa di richieste successive. Un'istanza in questa condizione si trova nello stato Warm. Se una nuova richiesta arriva mentre l'istanza è ancora "calda", la piattaforma la riutilizza, saltando l'intero processo di inizializzazione e passando direttamente all'esecuzione del codice. Questo risulta in una latenza molto bassa, un fenomeno noto come "warm start". Tuttavia, la durata di questo stato non è garantita, dopo un periodo di inattività che può variare, il provider dealloca l'istanza per liberare risorse, facendola tornare allo stato Cold.

3.3.3 Lo stato Pre-warmed (Pre-riscaldato)

Per le applicazioni sensibili alla latenza dove i cold start sono inaccettabili, è stata introdotta una terza strategia, nota come "pre-warming". Questa strategia fa si che un numero specifico di istanze di una funzione sia inizializzato e mantenuto costantemente nello stato Warm, prima ancora che arrivi la prima richiesta. In questo modo, quando il traffico arriva, trova già un pool di ambienti pronti a rispondere, eliminando di fatto la possibilità di un cold start. Questa strategia garantisce le massime performance e una latenza prevedibile, ma introduce un costo aggiuntivo: si paga per mantenere le istanze attive, indipendentemente dal fatto che stiano processando richieste o meno.

3.4 Algoritmi di scheduling in sistemi distribuiti

Lo scheduling in un sistema distribuito consiste nell'assegnare i processi ai diversi nodi di calcolo disponibili. L'approccio più comune e rilevante per questo contesto è il Load Balancing (bilanciamento del carico), il cui scopo è distribuire il carico di lavoro tra i nodi per massimizzare il throughput complessivo del sistema. L'obiettivo è mantenere ogni nodo ugualmente impegnato, evitando che alcuni siano sovraccarichi mentre altri rimangono inattivi.
Gli algoritmi di load balancing del carico si dividono principalmente in due categorie:

- Algoritmi Statici: la distribuzione del carico avviene senza tenere in considerazione lo stato corrente del sistema. La decisione viene presa a priori. Il vantaggio principale di questi algoritmi è la loro semplicità.
- Algoritmi Dinamici: prendono decisioni basandosi sul carico attuale di ciascun nodo. Redistribuiscono dinamicamente il lavoro dai nodi sovraccarichi a quelli meno impegnati, offrendo performance superiori, specialmente quando i tempi di esecuzione dei task sono molto variabili. Anche se più complessi da progettare, sono di solito più efficienti.

Un'altra differenza riguarda l'architettura decisionale, che può essere centralizzata o distribuita.

- Architettura Centralizzata: un singolo nodo è responsabile di tutte le decisioni di assegnazione. Questo approccio è efficiente perché tutta l'informazione sullo stato del

sistema è concentrata in un unico punto, ma ha una minore tolleranza ai guasti: se il nodo centrale fallisce, l'intero sistema si blocca.

- Architettura Distribuita: la logica di scheduling è distribuita tra i vari nodi del sistema, che collaborano per prendere decisioni. Questo modello è più resiliente e non ha un singolo punto di fallimento, ma può introdurre una maggiore complessità e overhead di comunicazione.

In questo progetto è stato scelto di utilizzare un'architettura centralizzata, dove l'API Gateway è il singolo nodo responsabile di tutte le decisioni.

4 Progettazione e architettura del framework

4.1 Requisiti

Per soddisfare l'obiettivo principale del progetto, sono stati definiti una serie di requisiti funzionali e non funzionali che il framework deve possedere. Essi garantiscono che il sistema non solo sia funzionante, ma anche robusto, facile da usare per la sperimentazione e aperto a future espansioni.

4.1.1 Requisiti funzionali

Descrivono le operazioni concrete che il sistema deve essere in grado di fare.

- Gestione di funzioni e nodi: il framework deve esporre delle API per consentire la registrazione dinamica sia delle funzioni da eseguire sia dei nodi di calcolo che compongono il cluster. Il gateway deve mantenere un registro di queste entità per poter orchestrare le esecuzioni.
- Scheduling configurabile: deve essere possibile implementare e selezionare diverse policy di scheduling senza dover modificare la logica centrale del gateway.
- Supporto agli stati di invocazione: il sistema deve implementare e gestire esplicitamente le tre principali modalità di invocazione (Cold, Pre-warmed e Warmed) per quantificare l'impatto del "cold start".
- Raccolta di metriche: i dati raccolti devono essere salvati in formati facilmente analizzabili, come tabelle di testo e grafici.

4.1.2 Requisiti non funzionali

Definiscono le qualità del sistema, influenzando l'esperienza d'uso e la manutenibilità.

- Modularità ed estensibilità: l'architettura deve essere costruita in modo che ogni sua parte svolga un compito specifico, facilitando così la modifica o l'aggiunta di nuove funzionalità senza impattare il resto del sistema. L'aggiunta di una nuova policy di scheduling o di una nuova strategia di selezione del nodo non deve richiedere modifiche invasive al codice esistente, ma deve limitarsi all'implementazione di una nuova classe che rispetti un'interfaccia predefinita.
- Riproducibilità: l'intero ambiente, composto dal gateway, dai nodi e dal client, è definito tramite codice utilizzando Docker. Uno script di avvio si occupa di "ripulire" lo stato del sistema prima di ogni esecuzione, fermando e rimuovendo container residui delle esecuzioni precedenti, per garantire un punto di partenza identico per ogni test.
- Portabilità: grazie all'uso della containerizzazione, il framework deve essere agnostico rispetto al sistema operativo sottostante. L'intera infrastruttura può essere eseguita su qualsiasi macchina su cui sia installato Docker Engine, garantendo portabilità tra diversi ambienti di sviluppo e test.

4.2 Architettura generale

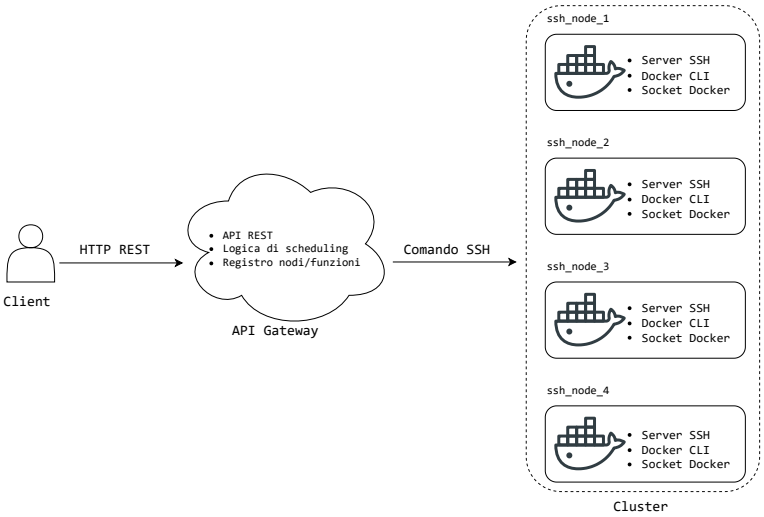

Figure 1: Schema dell'architettura generale del framework

L'architettura del framework è stata progettata seguendo un modello a microservizi distribuito, con l'obiettivo di garantire modularità, estensibilità e riproducibilità degli esperimenti. Il sistema si compone di diverse entità containerizzate che collaborano per orchestrare la registrazione e l'esecuzione delle funzioni. Ogni componente ha un ruolo specifico e ben definito, contribuendo a creare un ambiente di test controllato e flessibile.
Il cuore del sistema è l'API gateway, un server web sviluppato con il framework FastAPI in Python. Esso agisce come entrypoint e cervello dell'infrastruttura. Tutte le operazioni, dalla registrazione di una nuova funzione o di un nuovo nodo fino alla richiesta di esecuzione, passano attraverso le sue API REST. Il gateway mantiene lo stato del sistema, tenendo traccia dei nodi attivi e delle funzioni disponibili. La sua responsabilità principale è quella di applicare la policy di scheduling configurata per decidere, per ogni richiesta in arrivo, quale sia il nodo di esecuzione più idoneo a processarla, astraendo completamente questa logica dal client.
I nodi di esecuzione sono i componenti distribuiti che si occupano materialmente di eseguire il codice delle funzioni. Ogni nodo è un container Docker basato su un'immagine Ubuntu minimale, su cui è in esecuzione un server SSH per accettare comandi dal gateway. Una caratteristica chiave è che questi nodi montano il socket Docker del sistema host. Questo permette loro, anche se sono container isolati, di orchestrare l'avvio e la gestione di altri container sulla stessa macchina host. Quando il gateway seleziona un nodo, gli invia tramite SSH i comandi Docker necessari per eseguire la funzione richiesta.
Il client non è un utente finale, ma uno script Python progettato per orchestrare le sessioni di test in modo automatico. Il suo compito è simulare un carico di lavoro inviando richieste al gateway. Tipicamente, il client esegue un ciclo di operazioni: per prima cosa, registra i nodi di esecuzione e le funzioni che saranno oggetto del test, invia poi un numero predefinito di richieste di invocazione in modo sequenziale. Il client opera con totale trasparenza rispetto allo stato dell'insieme dei nodi di esecuzione: si limita ad invocare una funzione tramite il suo nome, senza sapere dove o come essa verrà eseguita.
Per simulare scenari di test realistici e quantificare l'impatto del cold start, sono state create due Immagini Docker custom per le funzioni. La prima, `custom_python_heavy`, è un'immagine "pesante" che include librerie scientifiche Python come pandas, scikit-learn e tensorflow, caratterizzata da un tempo di avvio e un ingombro di memoria significativi. La seconda, `custom_python_light`, è un'immagine "leggera" con dipendenze minime, che rappresenta un carico di lavoro più snello. L'utilizzo di queste due immagini permette di valutare il comportamento degli algoritmi di scheduling in condizioni operative differenti e di misurare l'impatto delle dimensioni dell'immagine sulla latenza di esecuzione.

4.3 Protocollo di comunicazione

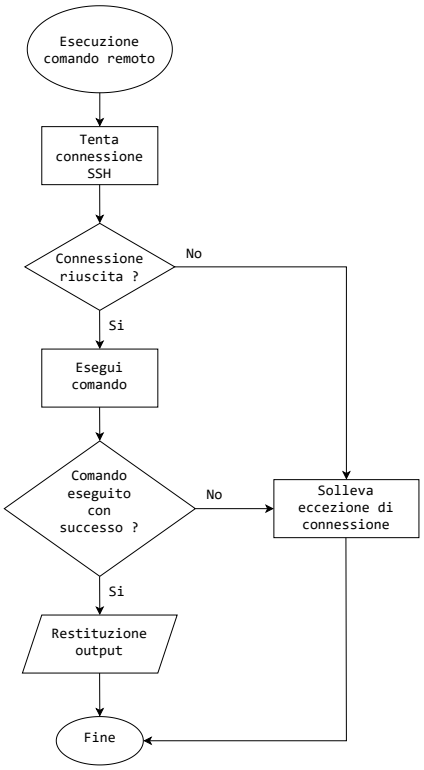

Figure 2: Diagramma di flusso per l'esecuzione di un comando remoto

Per la comunicazione tra l'API gateway e i nodi di esecuzione distribuiti è stato scelto il protocollo SSH (Secure Shell). Questa scelta è motivata dalla natura standard di SSH negli ambienti Linux e sulla sua capacità di fornire un canale crittografato per l'esecuzione di comandi su macchine remote. E' sufficiente che su ogni nodo di esecuzione sia in esecuzione un server OpenSSH, come configurato nel rispettivo Dockerfile.
Per l'implementazione di questo protocollo nel gateway, è stata usata la libreria Python `asyncssh`, in quanto si integra nativamente con il framework `asyncio` di Python, lo stesso su cui si basa FastAPI. Questo è importante per le performance del gateway. Poiché le operazioni remote come il download di un'immagine Docker (`docker pull`) o l'esecuzione di una funzione possono richiedere un tempo non trascurabile, l'approccio asincrono permette al gateway di gestire molteplici connessioni e comandi SSH in modo concorrente, senza mai bloccare il suo loop principale. In questo modo, il gateway rimane reattivo e può continuare a processare altre richieste in entrata mentre attende il completamento di un'operazione remota.
Il flusso di comunicazione è gestito dalla funzione `run_ssh_command`. Quando il gateway deve inviare un comando ad un nodo, questa funzione stabilisce una connessione SSH utilizzando le credenziali fornite al momento della registrazione del nodo. Dato che i nodi di esecuzione sono container temporanei, è stato disabilitato il controllo degli host noti (`known_hosts=None`) per semplificare e rendere più rapida la connessione SSH. Questo evita la complessità legata alla gestione delle chiavi SSH. Una volta stabilita la connessione, il comando viene eseguito sul nodo remoto, e il suo output viene catturato e restituito al gateway. Questo meccanismo fornisce al gateway il pieno controllo necessario per orchestrare il ciclo di vita dei container delle funzioni su tutta l'infrastruttura.

4.4 Design delle policy modulari

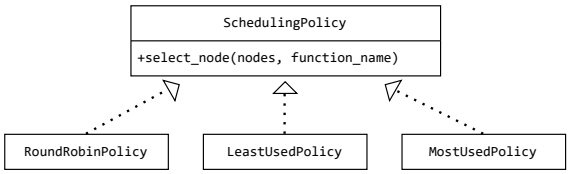

Figure 3: Diagramma delle classi per le policy di scheduling basato sul design pattern Strategy

Uno degli obiettivi di questo lavoro di tesi è la creazione di un framework che agisca da testbed, cioè da banco di prova per analizzare e confrontare le performance di diversi algoritmi di scheduling. Per soddisfare questo requisito, si deve far si che l'architettura consenta di "montare" e sostituire le diverse strategie di scheduling in modo semplice e rapido. Questo principio è alla base del requisito non funzionale di modularità ed estensibilità definito prima. Il design delle policy deve quindi permettere a chiunque voglia estendere questo progetto, di implementare un nuovo algoritmo e di integrarlo nel sistema con il minimo sforzo, senza dover modificare la logica centrale del gateway.
Per raggiungere questo livello di flessibilità, l'architettura adotta il design pattern "Strategy". Questo pattern di progettazione software permette di definire una famiglia di algoritmi, incapsulare ciascuno di essi in una classe separata e renderli intercambiabili. Nel contesto del nostro framework, ogni specifico algoritmo di scheduling è implementato come una "strategia" indipendente. Il gateway può essere configurato per utilizzare una qualsiasi di queste strategie senza che il suo funzionamento interno venga alterato.
Il fulcro di questo design è la definizione di un'interfaccia comune, che ogni classe di policy deve rispettare per poter essere considerata valida e intercambiabile. Sebbene Python non richieda interfacce esplicite come altri linguaggi, il contratto è definito dalla firma del metodo principale di ogni policy: `async def select_node(nodes: Dict[str, Any], function_name: str)`. Questo metodo accetta come parametri l'elenco dei nodi di esecuzione disponibili e il nome della funzione da invocare. La sua unica responsabilità è quella di eseguire la logica specifica dell'algoritmo che implementa e di restituire il nome del nodo che ha selezionato come più idoneo per l'esecuzione.
L'integrazione di questo design nel gateway è semplice ed efficace. All'avvio del server, viene istanziata una policy concreta, nella variabile `DEFAULT_SCHEDULING_POLICY`, e il suo oggetto viene mantenuto per tutta la durata della sessione. La logica che gestisce le richieste di invocazione è completamente disaccoppiata dall'algoritmo specifico: si limita a invocare il metodo `select_node` sulla variabile `DEFAULT_SCHEDULING_POLICY`, che contiene l'istanza della policy di scheduling scelta all'avvio del gateway..
I vantaggi di questo approccio:

- Estensibilità: per testare un nuovo algoritmo di scheduling, basta creare una nuova classe che rispetti l'interfaccia `select_node`. Non è richiesta alcuna modifica al codice core del gateway.
- Manutenibilità e testabilità: la logica di ogni algoritmo è isolata nella sua classe, rendendo il codice più pulito, più facile da mantenere e consentendo di testare ogni strategia in modo indipendente.
- Flessibilità sperimentale: cambiare l'algoritmo di scheduling per un'intera sessione di test si riduce alla modifica di una singola riga di codice nel punto in cui la policy viene istanziata.

5 Dettagli implementativi

5.1 Implementazione del gateway

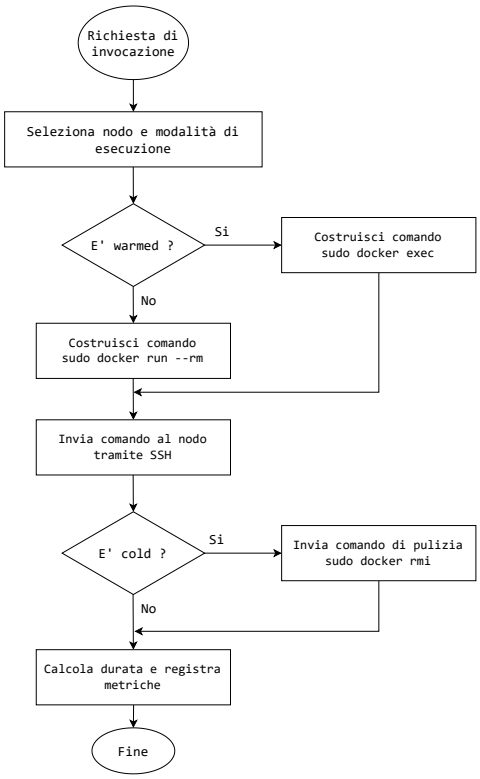

Figure 4: Diagramma di flusso della gestione di una richiesta di invocazione di funzione

Il gateway è il cervello dell'intero framework. È un'applicazione web asincrona costruita utilizzando FastAPI. Le sue responsabilità principali sono tre: esporre le API REST per la gestione del sistema, mantenere lo stato corrente di nodi e funzioni registrate e orchestrare ogni singola invocazione applicando le policy di scheduling.
La gestione dello stato è implementata attraverso semplici strutture dati in memoria, definite nel modulo `state.py`. I dizionari `function_registry` e `node_registry` contengono le informazioni relative alle funzioni e ai nodi registrati, mentre `function_state_registry` tiene traccia dello stato (es. warmed, pre-warmed) di una funzione su un determinato nodo.
Per garantire la persistenza dei risultati tra diverse esecuzioni del testbed, il gateway implementa una funzione `startup_event` che, all'avvio del server, verifica la presenza di un file `metrics.csv` preesistente e, se presente, ne carica i contenuti nella lista `metrics_log` in memoria.
L'interfaccia del gateway è definita da tre API endpoints principali.
I primi due, `/nodes/register` e `/functions/register`, gestiscono la registrazione dei componenti. Mentre la registrazione di un nodo si limita a salvarne le credenziali, quella di una funzione innesca un processo più complesso: dopo aver salvato i dati della funzione (immagine Docker e comando), invoca immediatamente il metodo `apply` della `SCHEDULING_POLICY`. Questo passaggio è cruciale, perché è qui che viene applicata la strategia di warming (se configurata) in base alla variabile globale `WARMING_TYPE`, avviando ad esempio il pre-caricamento dell'immagine o la creazione di un container persistente.
L'endpoint più importante è `/functions/invoke/{function_name}`, che orchestra l'esecuzione di una funzione.
Il flusso è il seguente:

- Selezione del nodo: viene invocato il metodo `NODE_SELECTION_POLICY.select_node`,

il quale implementa la strategia di alto livello (es. dare priorità a nodi con container warmed). Questa chiamata restituisce il nome del nodo scelto e la modalità di esecuzione effettiva (Cold, Pre-warmed o Warmed).

- Costruzione del comando: in base alla modalità di esecuzione restituita sopra, il gateway costruisce dinamicamente il comando Docker da eseguire. Se la modalità è Warmed, viene assemblato un comando `sudo docker exec` per eseguire il codice su un container già attivo. Per le modalità Cold e Pre-warmed, viene invece costruito un comando `sudo docker run --rm` per avviare un container temporaneo.

- Esecuzione remota: il comando viene inviato al nodo selezionato tramite la funzione `run_ssh_command`. L'output testuale restituito dalla funzione (una stringa di conferma "Ok") viene catturato e registrato nei log del gateway per finalità di debug. Si evita di stampare il risultato del calcolo, perché potrebbe essere molto lungo ed intasare l'output del terminale.

- Pulizia dell'immagine dopo invocazioni Cold: una volta che un'esecuzione di tipo Cold è terminata, il gateway esegue un ulteriore passo per garantire la purezza dei test futuri. Viene inviato un comando `sudo docker rmi` per rimuovere l'immagine Docker appena utilizzata dal nodo. Questa scelta metodologica è utile per assicurare che un'eventuale successiva invocazione Cold sullo stesso nodo sia forzata a riscaricare l'immagine, evitando che i tempi di esecuzione vengano alterati da una versione rimasta in cache.

- Metriche: una volta ottenuta la risposta, il gateway calcola il tempo totale di esecuzione e invoca la funzione `log_invocation_metrics` per registrare tutte le metriche della chiamata (nodo utilizzato, modalità di esecuzione, tempo impiegato, uso di CPU/RAM) sul file `metrics.csv`.

5.2 Struttura dei nodi di esecuzione

I nodi di esecuzione sono i worker del framework, responsabili dell'effettiva esecuzione del codice delle funzioni. Ogni nodo è un'istanza containerizzata indipendente, definita da un `Dockerfile` e orchestrata dal file `docker-compose.yml`. Per garantire un ambiente pulito e standardizzato, ogni nodo è basato su un'immagine ufficiale di Ubuntu.
All'interno di questo ambiente minimale vengono installati i pacchetti software essenziali per il suo funzionamento: il server OpenSSH, che permette al gateway di connettersi e inviare comandi, e la suite Docker (`docker.io`), che fornisce l'interfaccia a riga di comando per gestire i container. Una scelta architetturale importante è il montaggio del socket Docker del sistema host all'interno di ogni nodo. Questo permette ad un container (il nodo) di comunicare con il demone Docker dell'host e, di avviare, fermare e gestire altri container "fratelli" sulla stessa macchina. Questo evita la complessità di dover eseguire un demone Docker all'interno di ogni nodo.
La configurazione di runtime di ogni nodo è gestita dallo script `entrypoint.sh`.
Al momento dell'avvio, questo script esegue una serie di operazioni:

- Creazione utente SSH: viene creato un utente non privilegiato, `sshuser`, con una password predefinita. Questo utente viene utilizzato dal gateway per tutte le connessioni SSH, evitando l'uso dell'utente root.

- Configurazione dei permessi Docker: l'utente `sshuser` viene aggiunto al gruppo docker del sistema. Per superare eventuali problemi di permessi in ambienti di test, viene configurato un accesso `sudo` senza password specificamente per il comando `/usr/bin/docker`.

- Generazione dello script di metriche: lo script crea dinamicamente un file eseguibile in `/usr/local/bin/get_node_metrics.sh`. Questo script è fatto per leggere le informazioni sull'utilizzo delle risorse direttamente dai file di sistema del kernel Linux situati in `/sys/fs/cgroup/`. È in grado di rilevare automaticamente la versione di `cgroups` (v1 o v2) in uso sull'host, garantendo la portabilità. Il suo compito è calcolare l'uso percentuale di CPU e RAM del nodo stesso e restituire i dati in un formato JSON, facilmente interpretabile dal gateway.

Per assicurare la riproducibilità e il controllo degli esperimenti, ad ogni nodo di esecuzione vengono imposti dei limiti di risorse stringenti tramite la configurazione `deploy` nel file `docker-compose.yml`, vincolando l'utilizzo massimo di CPU e RAM.

5.3 Implementazione delle policy di scheduling

Le policy di scheduling sono il cuore della logica decisionale del framework. Come anticipato nel capitolo sulla progettazione, ogni policy è una classe Python a sé stante che implementa la logica per selezionare il nodo più idoneo all'esecuzione di una funzione. Tutte le policy sono definite nel file `policies.py` e condividono un meccanismo di sicurezza: prima di considerare un nodo per l'esecuzione, ne verificano l'utilizzo della RAM, scartandolo se questo supera la soglia `RAM_THRESHOLD` definita in `state.py`. Questa condizione previene l'assegnazione di lavoro a nodi già sovraccarichi.

5.3.1 RoundRobinPolicy

La `RoundRobinPolicy` è l'implementazione di un algoritmo di scheduling statico, il cui obiettivo è distribuire il carico in modo equo e ciclico tra tutti i nodi disponibili, senza considerare il loro carico di lavoro attuale.

- Logica: la classe usa `itertools.cycle` per creare un iteratore infinito che scorre la lista dei nodi registrati. Per gestire dinamicamente l'aggiunta o la rimozione di nodi durante l'esecuzione del test, la policy mantiene una cache locale dei nomi dei nodi (`_nodes_cache`). All'inizio di ogni chiamata a `select_node`, confronta la lista attuale dei nodi con la sua cache. Se rileva una differenza, ricrea l'iteratore ciclico con la nuova lista di nodi, garantendo che il sistema sia sempre aggiornato.
- Processo di selezione: quando deve selezionare un nodo, la policy estrae il prossimo elemento dall'iteratore. A questo punto, contatta il nodo scelto per recuperarne le metriche di CPU e RAM tramite la funzione `get_metrics_for_node`. Se l'utilizzo della RAM è inferiore alla soglia, il nodo viene restituito come vincitore. In caso contrario, viene scartato e il processo continua con il nodo successivo nell'iteratore, fino a trovare un candidato idoneo o a esaurire tutti i nodi disponibili.

5.3.2 LeastUsedPolicy

La `LeastUsedPolicy` implementa un algoritmo di scheduling dinamico e informato. Il suo scopo è bilanciare attivamente il carico del cluster inviando le nuove richieste al nodo che, in quel preciso momento, presenta il minor carico sulla CPU.

- Logica: questa policy necessita di una visione globale e aggiornata dello stato di tutti i nodi prima di poter prendere una decisione. Per ottenere queste informazioni, utilizza un metodo ausiliario `_get_all_node_metrics`. Questo metodo crea una lista di task `asyncio`, uno per ogni nodo da interrogare, e li esegue in parallelo tramite `asyncio.gather`. Questo approccio riduce drasticamente il tempo di attesa, poiché le chiamate SSH per il recupero delle metriche vengono eseguite contemporaneamente anziché in sequenza.
- Processo di selezione: una volta ottenute le metriche da tutti i nodi, la policy filtra l'elenco per mantenere solo i nodi "eleggibili", ovvero quelli che rispettano il vincolo sulla RAM. Se non ci sono nodi eleggibili, la selezione fallisce. Altrimenti, sul gruppo di candidati rimanenti, applica la funzione `min()` di Python, utilizzando come chiave di confronto il valore del campo `cpu_usage`. In questo modo, viene identificato e restituito il nodo con il valore di utilizzo della CPU più basso.

5.3.3 MostUsedPolicy

La `MostUsedPolicy` è l'opposto della `LeastUsedPolicy`. È un algoritmo dinamico progettato per selezionare il nodo con il maggior carico di CPU, purché la sua RAM sia al di sotto della soglia. Anche se controintuitiva per un bilanciamento del carico tradizionale, questa policy è stata implementata per scopi sperimentali:

- Permette di studiare il comportamento del sistema quando il carico viene deliberatamente concentrato su pochi nodi.
- È utile per testare strategie dove l'obiettivo è saturare le risorse di un nodo prima di passare al successivo, ad esempio per ottimizzare i costi energetici.
- Logica e processo di selezione: la sua implementazione è identica a quella della `LeastUsedPolicy`. Usa lo stesso meccanismo basato su `asyncio.gather` per la raccolta parallela delle metriche e lo stesso filtro sulla RAM. L'unica differenza è nella riga finale della selezione, dove viene utilizzata la funzione `max()` anziché `min()` per identificare e restituire il nodo con il valore di `cpu_usage` più alto.

5.4 Implementazione delle strategie di invocazione

Mentre le policy di scheduling si occupano di bilanciare il carico tra i nodi, il framework implementa una logica di livello superiore per gestire le strategie di invocazione, con l'obiettivo di minimizzare la latenza, specialmente quella derivante dal cold start. Questa logica è gestita da una serie di classi, definite in `policies.py`, che implementano un design pattern a catena di responsabilità (Chain of Responsibility).
La selezione del nodo non viene fatta direttamente da una policy di load balancing, ma da una meta-policy di selezione, definita in `main.py` dalla variabile `NODE_SELECTION_POLICY`. Questa classe è il primo anello della catena e da inizio ad un processo di selezione gerarchico. L'obiettivo è trovare un nodo che possa eseguire la funzione con la latenza più bassa possibile.
Questi sono i passaggi del processo:

- `WarmedFirstPolicy`: questo è il primo anello della catena. La sua responsabilità è controllare se per la funzione richiesta esiste un container già nello stato Warmed. Interroga il dizionario `function_state_registry`. Se trova un nodo su cui la funzione è marcata come Warmed, lo seleziona e lo restituisce, interrompendo la catena. Questo garantisce la priorità assoluta alle esecuzioni a latenza minima (`docker exec`). Se nessun container warmed viene trovato, la policy non prende una decisione, ma delega la richiesta all'anello successivo della catena, cioè `PreWarmedFirstPolicy`.
- `PreWarmedFirstPolicy`: questo secondo anello si attiva solo se il precedente ha fallito. La sua logica è simile a sopra: controlla il `function_state_registry` cercando un nodo su cui la funzione sia nello stato Pre-warmed. Questo stato indica che l'immagine Docker della funzione è già stata scaricata sul nodo (`docker pull`), garantendo un avvio (`docker run`) più rapido rispetto a un cold start completo. Se trova un nodo che soddisfa questa condizione, lo seleziona e lo restituisce. Altrimenti, anche questa policy fallisce e delega la responsabilità all'ultimo anello della catena, `DefaultColdPolicy`.
- `DefaultColdPolicy`: questo è l'anello finale della catena e agisce come fallback. Non esegue alcun controllo sullo stato dei container. La sua funzione è quella di invocare la policy di scheduling di base (es. `RoundRobinPolicy`, `LeastUsedPolicy` o `MostUsedPolicy`), contenuta nella variabile `DEFAULT_SCHEDULING_POLICY`. Sarà quindi quest'ultima a selezionare un nodo basandosi sul proprio algoritmo. Questa è la strategia che porta ad un'esecuzione di tipo Cold, dove non c'è nessuna garanzia che l'immagine sia già presente sul nodo.

5.5 Gestione dello stato e flusso di invocazione

In questo capitolo si farà una visione d'insieme del percorso completo di una funzione all'interno del framework. Il ciclo di vita di una funzione è diviso in due fasi: una prima fase di registrazione e preparazione dello stato (che avviene una sola volta) e una seconda fase di invocazione ed esecuzione (che si ripete per ogni richiesta).

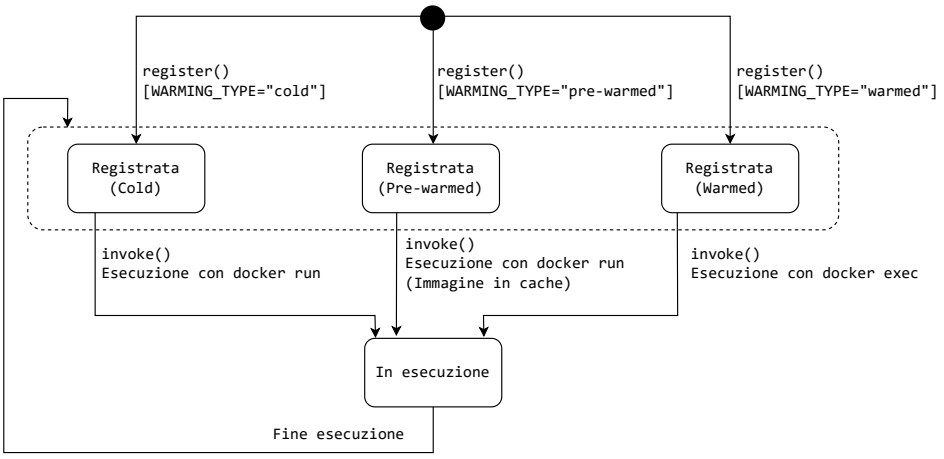

Figure 5: Diagramma a stati del ciclo di vita di una funzione nel framework

5.5.1 Registrazione e preparazione dello stato

Il ciclo di vita di una funzione inizia quando il client invia una richiesta all'endpoint `/functions/register` del gateway. Questo non è un semplice salvataggio di dati, ma un processo attivo che definisce lo stato iniziale della funzione nel cluster, basandosi sulla configurazione globale `WARMING_TYPE` del gateway. La classe `StaticWarmingPolicy` si occupa di interpretare questa configurazione e di preparare i nodi di conseguenza.
Si possono verificare tre scenari:

- Configurazione Warmed: se il framework è in modalità Warmed, la `StaticWarmingPolicy` seleziona un nodo (usando la policy di scheduling di default) e gli invia tramite SSH il comando per avviare un container persistente (`sudo docker run -d ... sleep infinity`). Ad operazione completata, lo stato della funzione su quel nodo viene registrato come Warmed nel `function_state_registry`.
- Configurazione Pre-warmed: la policy seleziona un nodo e gli invia il comando per scaricare l'immagine Docker della funzione (`sudo docker pull`). L'immagine viene così memorizzata nella cache del nodo, e lo stato della funzione viene registrato come Pre-warmed.
- Configurazione Cold: non viene eseguita alcuna azione sui nodi di esecuzione. Il gateway si limita a registrare l'esistenza della funzione nel `function_registry`. Nessun'informazione viene aggiunta al `function_state_registry`, indicando che non esistono istanze pre-riscaldate.

5.5.2 Invocazione ed esecuzione

Quando il client invia una richiesta all'endpoint `/functions/invoke`, inizia la fase di esecuzione. A questo punto, lo stato del cluster è già stato preparato e il compito del gateway è scegliere il percorso di esecuzione più efficiente possibile, seguendo la logica della catena di responsabilità descritta in precedenza (`WarmedFirstPolicy`).
Il risultato è uno dei seguenti percorsi di esecuzione:

- Percorso Warmed (latenza minima): la `WarmedFirstPolicy` trova un'istanza Warmed nel `function_state_registry`. Il gateway costruisce un comando `sudo docker exec` e lo invia al nodo corrispondente. Questo è il percorso più veloce in assoluto, perché salta sia il tempo di creazione del container sia il download dell'immagine.
- Percorso Pre-warmed (latenza ridotta): se non esistono istanze Warmed, la `PreWarmedFirstPolicy` trova un'istanza Pre-warmed. Il gateway costruisce un comando `sudo docker run --rm`. L'esecuzione è più lenta del percorso Warmed perché richiede la creazione di un nuovo container, ma è comunque più veloce di un cold start completo perché evita il tempo di download dell'immagine, che è già presente nella cache del nodo.

- Percorso Cold (latenza massima): se nessuno dei controlli precedenti ha successo, la `DefaultColdPolicy` seleziona un nodo usando la policy di bilanciamento del carico. Il gateway invia un comando `sudo docker run --rm`. Questa è la situazione che genera la latenza più alta, poiché il tempo di esecuzione include sia la creazione del container sia l'intero tempo di download dell'immagine Docker dal registry.

6 Analisi sperimentale e risultati

6.1 Metodologia e ambiente di test

Per garantire la validità e la riproducibilità dei risultati, tutti gli esperimenti sono stati eseguiti seguendo una metodologia rigorosa e in un ambiente hardware e software ben definito. Ogni sessione di test è stata orchestrata in modo automatico dallo script `start.py`, il quale assicura un punto di partenza identico per ogni esecuzione. Prima di avviare l'infrastruttura, lo script esegue una pulizia completa dell'ambiente Docker, rimuovendo eventuali container e immagini residue da esecuzioni precedenti, per poi avviare l'intero sistema tramite `docker-compose up`.

6.1.1 Hardware e software

I test sono stati eseguiti su un computer MacBook Pro dotato di processore Apple M4, 16 GB di RAM e sistema operativo macOS Tahoe. L'ambiente di containerizzazione è stato gestito da Docker Engine versione 28.4.0, con Docker Compose versione 2.39.2. L'intero codice del framework, incluse le funzioni di test e gli script di orchestrazione, è stato eseguito utilizzando l'interprete Python 3.13.7.

6.1.2 Configurazione del framework

L'infrastruttura di test definita nel file `docker-compose.yml` è composta da un API Gateway centrale e quattro nodi di esecuzione. Per simulare un ambiente con risorse limitate e per garantire la coerenza dei risultati tra i diversi test, ad ogni nodo di esecuzione sono state imposte delle limitazioni hardware stringenti: un massimo di 0.50 core di CPU e 512 MB di RAM. Per ogni scenario sperimentale, il client è stato configurato per eseguire un totale di 100 invocazioni per ogni funzione registrata, così da raccogliere un campione di dati sufficientemente ampio per l'analisi statistica.

6.1.3 Funzioni di test e carico di lavoro

Per simulare carichi di lavoro eterogenei e per isolare l'impatto delle diverse componenti della latenza (inizializzazione vs. esecuzione), sono state definite due funzioni di test:

- Funzione `fibonacci-image-big-func-small`: questa funzione associa l'immagine "pesante" (`custom_python_heavy`) ad un carico computazionale leggero (`COMMAND_LIGHT`). Lo scopo è massimizzare l'impatto del cold start, rendendo il tempo di download e avvio del container la componente dominante della latenza totale.
- Funzione `fibonacci-image-small-func-big`: questa funzione associa l'immagine "leggera" (`custom_python_light`) ad un carico computazionale pesante (`COMMAND_HEAVY`). In questo scenario, l'overhead di inizializzazione è minimo, e la latenza misurata è quasi interamente attribuibile al tempo di esecuzione effettivo del codice sulla CPU.

Il carico computazionale effettivo di entrambe le funzioni è un calcolo della serie di Fibonacci. La complessità del calcolo viene definita nel client tramite due comandi distinti, `COMMAND_LIGHT` e `COMMAND_HEAVY`, che passano il numero di iterazioni come argomento a riga di comando allo script `loop_function.py` eseguito all'interno del container.

6.2 Metriche di valutazione

Per condurre un'analisi quantitativa e oggettiva delle diverse strategie di scheduling e di invocazione, è stato definito un set di metriche. Queste metriche vengono raccolte automaticamente dal framework per ogni singola invocazione di funzione e registrate in un file in formato CSV (`metrics.csv`), permettendo un'analisi dettagliata e la generazione di grafici. Le metriche sono state suddivise in una metrica primaria, che valuta la performance percepita dall'utente, e due metriche secondarie, che descrivono lo stato interno del sistema e l'efficacia del bilanciamento del carico.

6.2.1 Metrica primaria: latenza di esecuzione

La latenza di esecuzione (tempo di esecuzione) è il principale indicatore di performance del framework. Viene definita come l'intervallo di tempo totale che intercorre tra il momento in cui il gateway riceve una richiesta di invocazione e il momento in cui l'esecuzione della funzione sul nodo remoto è terminata.
Questa metrica è fondamentale perché rappresenta la latenza end-to-end del servizio, ovvero il ritardo effettivo che un utente finale o un altro servizio dovrebbe attendere.
L'analisi di questa metrica ha due obiettivi:

- Confrontare la performance media delle diverse strategie, calcolando la media aritmetica dei tempi di esecuzione per ogni scenario di test.
- Analizzare la distribuzione statistica e la consistenza dei tempi di risposta attraverso l'uso di box plot, che permettono di visualizzare la mediana, i quartili e l'eventuale presenza di outlier.

6.2.2 Metriche secondarie: utilizzo delle risorse

Per valutare l'efficacia degli algoritmi di scheduling nel distribuire il carico di lavoro, sono state definite due metriche secondarie che descrivono lo stato di un nodo al momento della selezione:

- Uso della CPU (%): indica la percentuale di carico sulla CPU di un nodo, misurata subito prima che lo scheduler gli assegni l'esecuzione della funzione.
- Uso della RAM (%): indica la percentuale di memoria RAM occupata sul nodo, misurata subito prima dell'assegnazione del task.

Questi dati vengono raccolti tramite uno script eseguito su ogni nodo, il quale legge le informazioni direttamente dal file system `cgroups` del kernel Linux per ottenere una misurazione accurata delle risorse consumate dal container del nodo stesso. Queste metriche non misurano direttamente la performance di una singola invocazione, ma sono importanti per verificare se una policy di scheduling stia effettivamente bilanciando il carico in modo efficiente.

6.3 Scenari di test e risultati

Per valutare in modo completo le performance del framework, sono stati definiti due scenari di test principali. Il primo scenario si concentra sull'analisi dell'impatto delle diverse strategie di invocazione (Cold, Pre-warmed, Warmed) sulla latenza. Il secondo scenario mette a confronto l'efficacia delle policy di scheduling (`RoundRobinPolicy`, `LeastUsedPolicy`, `MostUsedPolicy`) nel bilanciare il carico di lavoro tra i nodi. Ciascuno di questi scenari è stato eseguito per entrambe le funzioni di test (`fibonacci-image-big-func-small` e `fibonacci-image-small-func-big`), al fine di analizzare il comportamento del sistema sia in condizioni di cold start dominante sia di carico computazionale dominante. Per ogni scenario, verranno presentati i risultati ottenuti divisi per singola funzione.

6.3.1 Scenario 1: Analisi delle strategie di invocazione

- Obiettivo: lo scopo di questo esperimento è misurare e quantificare l'impatto del cold start sulla latenza di esecuzione, confrontando le performance dei tre stati di gestione dei container implementati nel framework: Cold, Pre-warmed e Warmed.
- Configurazione: per isolare l'effetto della strategia di invocazione, la policy di scheduling è stata mantenuta costante per tutti i test, utilizzando `RoundRobinPolicy`. Sono state eseguite tre sessioni di test distinte e indipendenti per ciascuna delle due funzioni, modificando di volta in volta la variabile di configurazione `WARMING_TYPE` nel gateway.
- Aspettative: l'ipotesi di partenza è che ci sia una differenza statisticamente significativa tra le latenze delle tre modalità. Ci si aspetta che la modalità Cold presenti la latenza media più alta e la maggiore variabilità, a causa del tempo necessario per il download dell'immagine e l'inizializzazione del container. La modalità Pre-warmed

dovrebbe mostrare una latenza notevolmente inferiore, eliminando il tempo di download dell'immagine. La modalità Warmed dovrebbe garantire la latenza più bassa e più costante in assoluto, poiché l'esecuzione avviene su un container già attivo.

- Risultati: i dati raccolti confermano le aspettative, evidenziando l'impatto delle strategie di warming sulla latenza. L'analisi viene fatta separatamente per le due funzioni, in quanto i risultati mostrano differenze significative legate alla natura del carico di lavoro.

Funzione `fibonacci-image-big-func-small` (Cold Start dominante). Come mostrato nei grafici 6 e 7, in questo scenario la latenza è dominata dal tempo di inizializzazione dell'immagine pesante. Il tempo medio di esecuzione in modalità Cold è di 6.542 secondi. Il passaggio alla modalità Pre-warmed, che elimina il solo tempo di download dell'immagine, riduce la latenza a 5.4022 secondi, con un miglioramento del 17.42%. La modalità Warmed, che elimina anche il tempo di avvio del container, abbassa ulteriormente la latenza a 5.2647 secondi, risultando il 19.52% più veloce rispetto alla Cold. Il box plot mostra una maggiore variabilità e la presenza di outlier nella modalità Cold, a fronte di una distribuzione molto più compatta e prevedibile per la modalità Warmed.

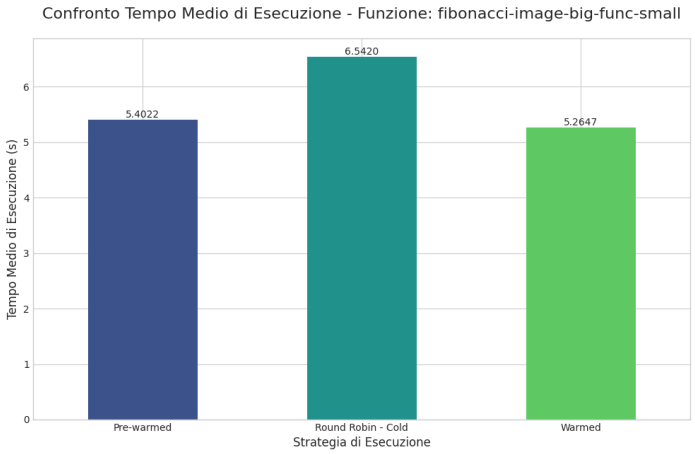

Figure 6: Confronto tempo medio di esecuzione (s) per la funzione con immagine pesante

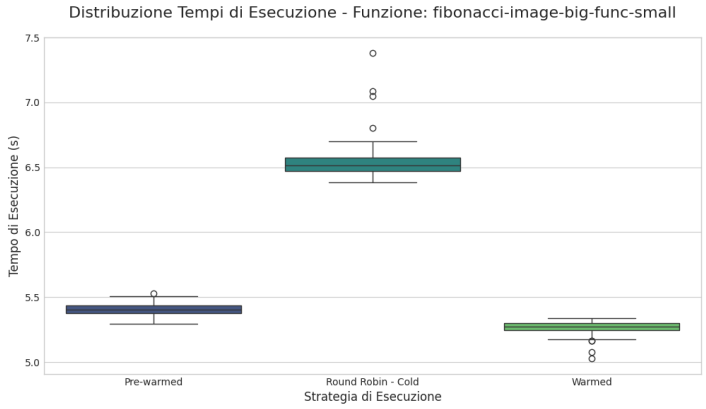

Figure 7: Distribuzione dei tempi di esecuzione (s) per la funzione con immagine pesante

Funzione `fibonacci-image-small-func-big` (carico CPU dominante). In questo scenario, dove l'overhead di inizializzazione è minimo e la latenza è determinata principalmente dal lungo tempo di calcolo, le strategie di warming mostrano un impatto positivo ma percentualmente inferiore. Come visibile nei grafici 8 e 9, il tempo medio in modalità Cold è di 13.3288 secondi. La modalità Pre-warmed riduce il tempo a 12.2149 secondi (un miglioramento del 8.36%), mentre la modalità Warmed scende a 12.0481 secondi, con un miglioramento complessivo del 9.61% rispetto alla Cold. Anche in questo caso, la modalità Warmed non solo è la più veloce, ma anche quella con la distribuzione dei tempi più prevedibile e consistente.

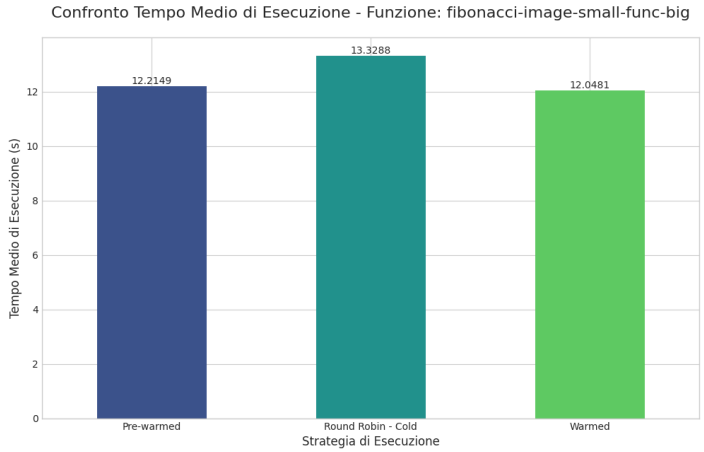

Figure 8: Confronto tempo medio di esecuzione (s) per la funzione con calcolo pesante

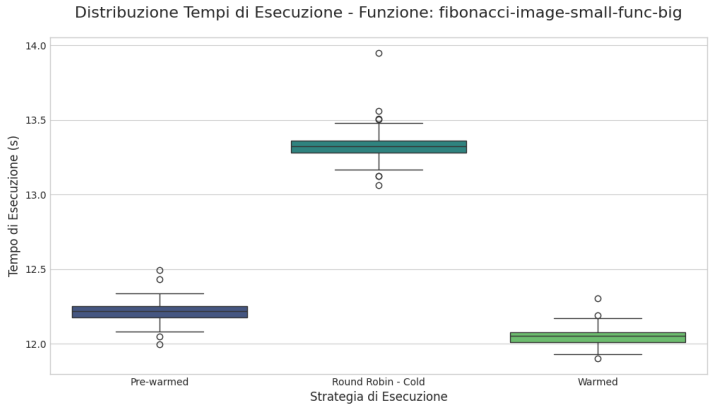

Figure 9: Distribuzione dei tempi di esecuzione (s) per la funzione con calcolo pesante

6.3.2 Scenario 2: Confronto delle policy di scheduling

- Obiettivo: questo esperimento ha lo scopo di confrontare l'efficacia di una policy di scheduling statica (`RoundRobinPolicy`) rispetto a due dinamiche e informate (`LeastUsedPolicy` e `MostUsedPolicy`) nel bilanciare il carico di lavoro e nell'influenzare le performance complessive del sistema.

- Configurazione: per garantire che la scelta del nodo non fosse influenzata da stati preesistenti, tutti i test di questo scenario sono stati eseguiti in modalità Cold. Sono state condotte tre sessioni di test separate per ciascuna delle due funzioni, modificando unicamente la policy di scheduling di default nel gateway.

- Aspettative:
  - `RoundRobinPolicy`: ci si attende che distribuisca le invocazioni in modo quasi perfettamente uniforme tra i quattro nodi.
  - `LeastUsedPolicy`: dovrebbe mostrare un comportamento più intelligente, tendendo a favorire i nodi con un utilizzo di CPU inferiore, bilanciando il carico.
  - `MostUsedPolicy`: si prevede un comportamento opposto, concentrando il carico sul nodo più utilizzato per creare deliberatamente uno scenario di carico non bilanciato.
- Risultati: i dati raccolti confermano le aspettative riguardo la distribuzione del carico e rivelano un impatto misurabile delle policy sulla consistenza della latenza. L'analisi è presentata separatamente per le due funzioni.

Funzione `fibonacci-image-big-func-small` (Cold Start dominante) In questo scenario, la `RoundRobinPolicy` distribuisce equamente le 100 invocazioni (25 per nodo), e la `LeastUsedPolicy` ottiene un risultato quasi identico, dimostrando la sua efficacia nel mantenere il sistema bilanciato (Grafico 10). La `MostUsedPolicy`, come atteso, concentra il carico, assegnando 35 invocazioni a `ssh_node_1`. Dal punto di vista della latenza (Grafico 11), le tre policy mostrano una latenza mediana quasi identica. Tuttavia, la `LeastUsedPolicy` presenta la distribuzione più compatta, indicando performance più consistenti e prevedibili. Al contrario, la `MostUsedPolicy` mostra la variabilità maggiore e la presenza di outlier, a conferma che la concentrazione del carico porta a una minore stabilità delle performance.

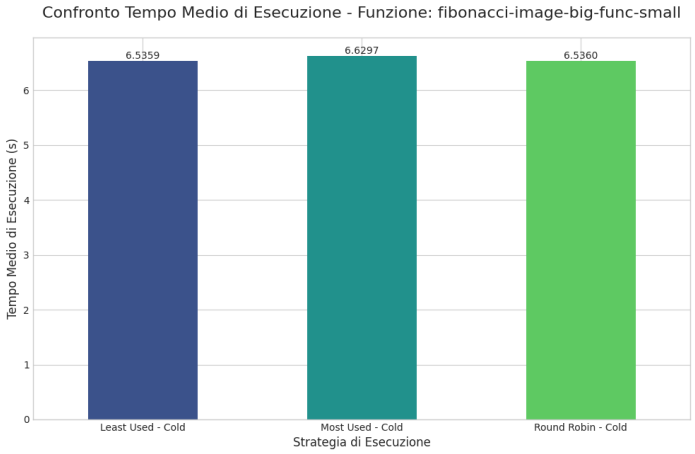

Figure 10: Distribuzione delle invocazioni per la funzione con immagine pesante

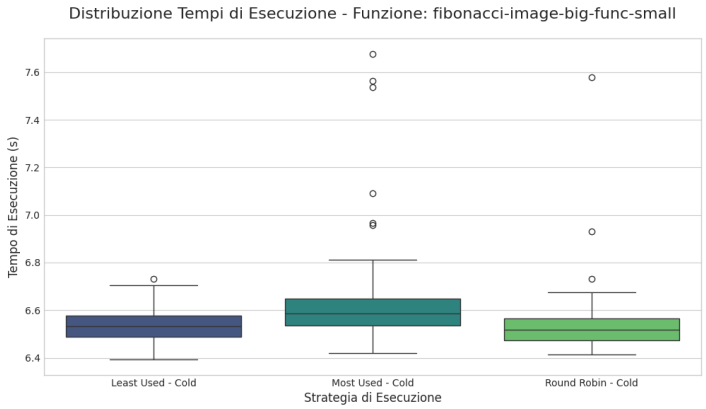

Figure 11: Distribuzione dei tempi di esecuzione per la funzione con immagine pesante

Funzione `fibonacci-image-small-func-big` (Carico CPU dominante) Anche con un carico computazionale pesante, i risultati sulla distribuzione del carico sono coerenti (Grafico 12). La `MostUsedPolicy` mostra un comportamento ancora più estremo, assegnando 42 invocazioni a `ssh_node_4`. L'impatto sulla latenza (Grafico 13) segue lo stesso trend osservato in precedenza. Sebbene la latenza media sia simile tra le policy, la `LeastUsedPolicy` si conferma la strategia più stabile, con la distribuzione dei tempi più ristretta. La `MostUsedPolicy` è di nuovo la policy con la variabilità più alta, suggerendo che la concentrazione di task computazionalmente intensivi sullo stesso nodo porta ad un degrado della prevedibilità delle performance.

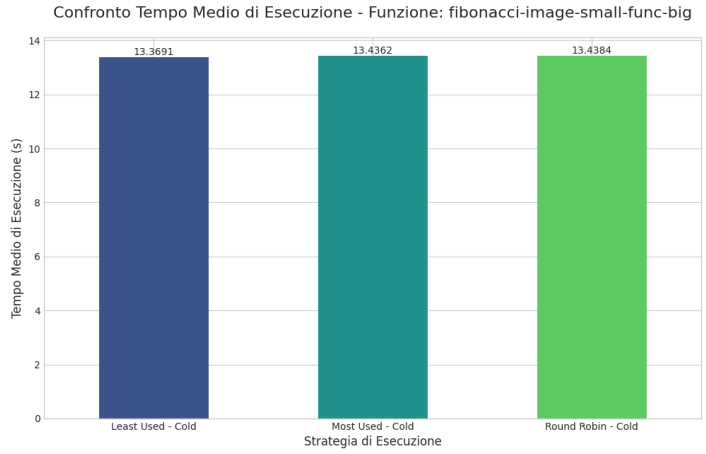

Figure 12: Distribuzione delle invocazioni per la funzione con calcolo pesante

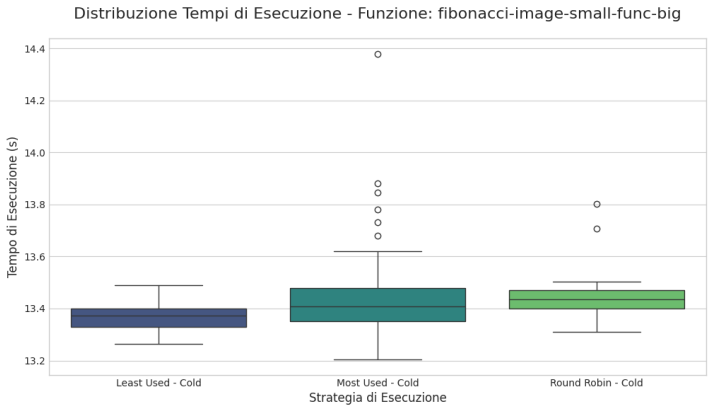

Figure 13: Distribuzione dei tempi di esecuzione per la funzione con calcolo pesante

6.4 Discussione dei risultati

L'analisi dei risultati presentati nel capitolo precedente permette di trarre conclusioni significative sull'efficacia delle diverse strategie implementate. In questa sezione, i dati verranno interpretati per spiegare i fenomeni osservati, mettendo in luce i compromessi tra le varie soluzioni e collegando i risultati agli obiettivi iniziali della tesi.

6.4.1 Analisi dello Scenario 1

I risultati dello Scenario 1 dimostrano che le strategie di warming sono il fattore più critico per la riduzione della latenza nei sistemi FaaS, confermando le ipotesi iniziali.
Per la funzione con immagine pesante, il passaggio dalla modalità Cold a quella Pre-warmed ha prodotto un miglioramento netto. Questo guadagno è quasi interamente attribuibile all'eliminazione del tempo di download dell'immagine Docker, che in questo caso rappresenta una porzione importante della latenza totale. Il passaggio ulteriore alla modalità Warmed offre un miglioramento minore ma comunque rilevante, dovuto all'eliminazione dei tempi di creazione e inizializzazione del container.
L'analisi della funzione con calcolo pesante è comunque importante. Sebbene il miglioramento percentuale sia inferiore, il tempo assoluto risparmiato è simile. Questo dimostra che il beneficio del warming è consistente, ma il suo impatto relativo diminuisce all'aumentare del tempo di esecuzione proprio della funzione e al diminuire della grandezza dell'immagine.
La modalità Warmed si conferma la soluzione con le performance assolute migliori e la più prevedibile e consistente. La scelta tra le tre strategie rappresenta quindi un compromesso: la modalità Cold ottimizza i costi a discapito della latenza, mentre le modalità Pre-warmed e Warmed scambiano un maggior utilizzo di risorse per un significativo e misurabile abbattimento dei tempi di risposta.

6.4.2 Analisi dello Scenario 2

Lo Scenario 2 ha evidenziato un aspetto altrettanto importante: l'impatto della policy di scheduling sulla stabilità e prevedibilità delle performance.
I risultati sulla distribuzione del carico hanno validato il design degli algoritmi: `RoundRobinPolicy` si è dimostrata un distributore statico ed equo, mentre `LeastUsedPolicy` e `MostUsedPolicy` hanno adattato dinamicamente le loro decisioni in base al carico della CPU, bilanciandolo nel primo caso e concentrandolo nel secondo.
Anche se la latenza media non è variata drasticamente tra le tre policy (probabilmente per la natura sequenziale del carico di test), la distribuzione dei tempi di risposta è cambiata in modo significativo. Per entrambe le funzioni, la `LeastUsedPolicy` ha prodotto la distribuzione più compatta, con una minore escursione tra i valori minimi e massimi. Questo suggerisce che, anche in condizioni di basso carico, la scelta di evitare nodi con carichi di CPU contribuisce a rendere le performance più consistenti.
Al contrario, la `MostUsedPolicy` ha peggiorato la prevedibilità, aumentando la variabilità dei tempi di risposta e il numero di outlier. Questo dimostra che un cattivo bilanciamento del carico non solo rischia di sovraccaricare i nodi, ma introduce anche un elemento di imprevedibilità nella latenza percepita dall'utente.

7 Conclusioni

7.1 Sintesi del lavoro svolto

Questo lavoro di tesi ha affrontato la necessità di disporre di strumenti flessibili e riproducibili per l'analisi delle performance nei sistemi FaaS. La crescente adozione del paradigma Serverless ha reso importante la comprensione di fenomeni come il cold start e l'impatto degli algoritmi di scheduling, che influenzano direttamente la latenza percepita dall'utente e l'efficienza dell'infrastruttura.
Per rispondere a questa esigenza, è stato progettato e implementato da zero un framework FaaS in Python, concepito specificamente come un testbed. L'architettura, basata su un API Gateway centrale e nodi di esecuzione containerizzati controllati via SSH, si è dimostrata robusta e flessibile. L'adozione di design pattern consolidati ha garantito un'alta modularità, rendendo il sistema facilmente estensibile.
È stata implementata una metodologia di test rigorosa, basata su due profili di funzione distinti per isolare e analizzare l'impatto della latenza di inizializzazione rispetto a quello del carico computazionale. Il framework sviluppato ha permesso di orchestrare esperimenti controllati per confrontare diverse policy di scheduling e strategie di gestione dei container. L'analisi sperimentale ha infine permesso di confrontare le diverse strategie implementate.
I risultati hanno confermato quantitativamente l'efficacia delle strategie di warming, che hanno ridotto la latenza media negli scenari dominati dal cold start. Inoltre, è stato evidenziato come la scelta della policy di scheduling impatti principalmente sulla stabilità delle performance: sebbene in condizioni di basso carico la latenza media non subisca variazioni significative, policy dinamiche come la `LeastUsedPolicy` hanno dimostrato di offrire una maggiore consistenza e prevedibilità dei tempi di risposta rispetto a un approccio statico.
Il lavoro svolto non solo fornisce una validazione delle teorie analizzate, ma offre alla comunità uno strumento pratico per future ricerche nel campo delle performance dei sistemi Serverless.

7.2 Limitazioni framework attuale

In quanto prototipo finalizzato alla ricerca, il framework presenta alcune limitazioni, dettate dalla necessità di privilegiare la semplicità e la controllabilità dell'ambiente di test rispetto a requisiti tipici di un sistema di produzione.

- Sicurezza: le credenziali per la comunicazione SSH tra il gateway e i nodi sono gestite in chiaro e la verifica degli host è disabilitata. Questa configurazione è del tutto inadeguata per un deployment reale.
- Persistenza dello stato: anche se i risultati delle metriche vengono salvati su file CSV, i registri delle funzioni e dei nodi disponibili sono mantenuti in memoria. Questo significa che ad ogni riavvio del gateway, lo stato del sistema viene perso e deve essere ricostruito da zero tramite le chiamate di registrazione del client.
- Scalabilità statica dei nodi: l'architettura del cluster di test (il numero di nodi di esecuzione) è definita staticamente nel file `docker-compose.yml`. Il framework non supporta la scalabilità dinamica, ovvero l'aggiunta o la rimozione di nodi a runtime.
- Interfaccia utente: l'interazione con il framework è possibile solo tramite API e script (`client.py`), manca un'interfaccia grafica (GUI) che potrebbe semplificare la configurazione degli esperimenti e la visualizzazione dei risultati.
- Gestione degli errori: la logica di gestione degli errori è basilare. Se l'invocazione di una funzione fallisce su un nodo, l'errore viene registrato, ma non vengono implementate strategie avanzate, come un meccanismo di retry automatico su un altro nodo.

7.3 Sviluppi futuri

Le limitazioni sopra aprono la strada a molti ed interessanti sviluppi futuri, che potrebbero trasformare il framework da un prototipo di ricerca a uno strumento ancora più potente e completo.

- Miglioramenti di sicurezza e usabilità: un primo passo potrebbe essere quello di rafforzare il protocollo di comunicazione, introducendo l'autenticazione basata su chiavi SSH. Si potrebbe poi sviluppare una semplice interfaccia web per la gestione degli esperimenti e la visualizzazione dei dashboard dei risultati.

- Monitoring avanzato: l'attuale sistema di raccolta metriche potrebbe essere potenziato integrando strumenti come Prometheus per la raccolta di dati time-series e Grafana per la creazione di dashboard interattive, offrendo una visione molto più completa ed in tempo reale dello stato del sistema.

- Estensione delle policy di scheduling: il framework si presta come base per implementare e testare algoritmi di scheduling più sofisticati. Si potrebbero esplorare strategie predittive, basate su machine learning, che tentino di stimare il tempo di esecuzione di una funzione o il carico futuro dei nodi.

- Supporto multi-language: per aumentare la versatilità del testbed, si potrebbe estendere il supporto a funzioni scritte in altri linguaggi di programmazione, permettendo di analizzare le performance di diversi runtime in ambiente FaaS.

- Gestione di richieste concorrenti: permetterebbe di testare il framework in condizioni di stress più realistiche, valutando non solo la latenza della singola operazione, ma anche il throughput totale del sistema. Si potrebbe così analizzare il comportamento delle policy di scheduling in scenari di contesa delle risorse e misurare la reattività del gateway quando deve gestire un grande numero di connessioni SSH e comandi Docker in parallelo.

Infine, il codice sorgente del progetto verrà reso disponibile sotto una licenza open source. Si auspica che questo possa incoraggiare studenti, ricercatori e appassionati ad utilizzare, modificare ed estendere il framework, contribuendo collettivamente alla ricerca e alla comprensione delle performance dei sistemi Serverless.

8 Bibliografia

- IBM - Cos'è il cloud computing?
  https://www.ibm.com/it-it/think/topics/cloud-computing
- Wikipedia - Cloud Computing
  https://it.wikipedia.org/wiki/Cloud_computing
- Italcom - L'evoluzione del Cloud Computing: dalle virtual machine ai container fino al serverless
  https://www.italcom.it/levoluzione-del-cloud-computing
- Red Hat - Cos'è il Function-as-a-Service (FaaS)
  https://www.redhat.com/it/topics/cloud-native-apps/what-is-faas
- Cloudfare - What is serverless computing?
  https://www.cloudflare.com/learning/serverless/what-is-serverless
- Red Hat - Understanding containers
  https://www.redhat.com/en/topics/containers
- Docker - Use containers to Build, Share and Run your applications
  https://www.docker.com/resources/what-container
- Martin Fowler - Serverless Architectures
  https://martinfowler.com/articles/serverless.html
- Geeks for Geeks - Scheduling and Load Balancing in Distributed System
  https://www.geeksforgeeks.org/computer-networks/scheduling-and-load-balancing-in-distributed-system
- AsyncSSH - Asynchronous SSH for Python
  https://asyncssh.readthedocs.io/en/latest
- Wikipedia - Strategy Pattern
  https://en.wikipedia.org/wiki/Strategy_pattern
- Wikipedia - Chain of Responsibility Pattern
  https://en.wikipedia.org/wiki/Chain-of-responsibility_pattern