UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Corso di Laurea Magistrale in informatica

TESI DI LAUREA

Sviluppo di un metodo per il calcolo del warm-up necessario per il testing di performance di applicazioni basate su microservizi nel cloud

RELATORE

Prof. Dario Di Nucci

Dott. Antonio Trovato

Università degli Studi di Salerno

CANDIDATO

Vincenzo RAIA

Matricola: 0522501442

Anno Accademico 2023-2024

Questa tesi è stata realizzata nel

"Il successo non è la chiave della felicità. La felicità è la chiave del successo. Se ami ciò che fai, avrai successo."

Abstract

Nel contesto del testing delle prestazioni delle applicazioni basate su microservizi nel cloud, la corretta identificazione del periodo di warm-up è fondamentale per garantire misurazioni affidabili e riproducibili. In questa tesi è stato sviluppato un metodo innovativo basato sull'intelligenza artificiale per il calcolo dinamico del warm-up, sfruttando tecniche avanzate di *Time Series Classification* (TSC).

Il framework proposto utilizza modelli di *Fully Convolutional Network* (FCN) e *RandOm Convolutional Kernel Transform* (ROCKET) per analizzare in tempo reale le metriche di performance e determinare automaticamente il passaggio allo steady state. L'approccio è stato validato sperimentalmente su un dataset di misurazioni raccolte da applicazioni reali basate su microservizi, evidenziando una significativa riduzione del tempo necessario per il testing rispetto ai metodi tradizionali. Il framework è stato confrontato con approcci statistici ed euristici, dimostrando un miglioramento in termini di accuratezza ed efficienza. In particolare, il sistema si è dimostrato robusto e generalizzabile su endpoint non presenti nei dati di addestramento, evidenziando un'efficace capacità di adattamento.

Questa soluzione rappresenta un avanzamento rispetto allo stato dell'arte, offrendo un metodo automatizzato, applicabile a diversi contesti di performance testing, con un impatto diretto sulla riduzione dei tempi e dei costi di valutazione delle prestazioni di applicazioni cloud-native.

Indice

Elenco delle figure

Elenco delle tabelle

CAPITOLO 1

Introduzione

1.1 Contesto applicativo

Nel panorama odierno dello sviluppo software, le applicazioni basate su architetture a microservizi stanno acquisendo un ruolo sempre più centrale, grazie alla loro capacità di affrontare esigenze crescenti in termini di **scalabilità**, **modularità** e **manutenzione**. Tali architetture, che si contraddistinguono per la suddivisione di sistemi complessi in componenti più piccoli e indipendenti, risultano particolarmente adatte per ambienti distribuiti e dinamici. Questo approccio consente alle organizzazioni di adattarsi rapidamente a nuove richieste di mercato, promuovendo **cicli di rilascio più brevi** e migliorando la **resilienza del sistema complessivo**.

Tuttavia, l'adozione di microservizi introduce una serie di sfide tecniche, che spaziano dalla gestione della comunicazione tra i servizi al **performance testing**. Quest'ultimo è cruciale per garantire che tali applicazioni possano gestire adeguatamente i carichi di lavoro previsti senza compromettere la qualità del servizio. Per raggiungere questo obiettivo, l'esecuzione di **benchmark affidabili** diventa essenziale. Il benchmarking, infatti, rappresenta una componente chiave del performance testing, poiché consente di misurare e confrontare metriche fondamentali come latenza, throughput e scalabilità. Mentre il performance testing si focalizza sull'analisi

delle capacità del sistema in condizioni specifiche, il benchmarking fornisce una base standardizzata per identificare colli di bottiglia, verificare la conformità ai requisiti e ottimizzare il comportamento complessivo del sistema in scenari reali.

Un aspetto peculiare che caratterizza le applicazioni moderne, soprattutto quelle sviluppate in linguaggi *runtime-based* come Java, Python e C#, è l'importanza della fase di **warm-up**. Durante questa fase, il sistema attraversa un periodo transitorio in cui le sue prestazioni non sono ancora stabili a causa di fenomeni come la **compilazione just-in-time (JIT)**, l'ottimizzazione del codice e l'inizializzazione delle risorse necessarie. Ad esempio, in un'applicazione Java, il compilatore JIT analizza il codice in esecuzione per generare versioni ottimizzate delle parti più frequentemente utilizzate, migliorando gradualmente le prestazioni. Questo comportamento, se non adeguatamente gestito, può introdurre **variabilità significativa** nei risultati dei test, compromettendo l'affidabilità delle misurazioni. Una volta superata la fase di warm-up, il sistema raggiunge uno stato detto di **steady state**, o stato stazionario, che rappresenta il momento in cui il sistema ha completato questa fase di assestamento e raggiunge un comportamento prestazionale stabile. Rilevare correttamente lo steady state è cruciale per garantire che le misurazioni effettuate durante i test riflettano accuratamente le prestazioni reali del sistema, eliminando l'influenza di fluttuazioni iniziali e consentendo confronti significativi e ripetibili.

Le sfide legate alla gestione della fase di warm-up sono amplificate nel caso delle applicazioni distribuite basate su microservizi, dove la **complessità del sistema** e l'**interazione tra i vari componenti** aggiungono ulteriori livelli di incertezza. Ad esempio, il comportamento di un singolo servizio potrebbe essere influenzato dal carico generato da altri servizi, rendendo ancora più difficile identificare il momento in cui il sistema raggiunge lo steady state. Ciò evidenzia l'importanza di adottare approcci metodologici in grado di distinguere con precisione tra la fase di warm-up e quella di esecuzione stabile.

Oltre alle sfide tecniche, il **tempo richiesto** per un ciclo di benchmarking è una variabile critica in contesti aziendali, dove il *time-to-market* è essenziale. I metodi tradizionali, basati su iterazioni fisse di warm-up, possono risultare rigidi e inefficaci: una sottostima porta a **misurazioni inaccurate**, mentre una sovrastima causa **spreco di risorse computazionali** e un **aumento dei tempi di esecuzione**.

1.2 Motivazioni e obiettivi

La ricerca nel campo del **performance testing** ha evidenziato la necessità di trovare un equilibrio tra la qualità dei risultati e il tempo richiesto per ottenerli. Questa sfida è particolarmente rilevante in contesti moderni, dove l'adozione di architetture complesse come quelle a microservizi aumenta significativamente il numero di variabili da considerare durante il testing. I **metodi tradizionali**, basati su un numero fisso di iterazioni per la fase di *warm-up*, presentano limiti intrinseci. Da un lato, una sottostima del numero di iterazioni necessarie può compromettere l'accuratezza delle misurazioni, con risultati non rappresentativi delle reali prestazioni del sistema. Dall'altro lato, una sovrastima del *warm-up* comporta un inutile aumento dei tempi di testing, riducendo l'efficienza complessiva e aumentando i costi computazionali.

Queste limitazioni hanno spinto lo sviluppo di approcci più sofisticati. Tecniche avanzate, basate su **euristiche dinamiche** e **modelli statistici**, cercano di adattare dinamicamente il numero di iterazioni necessarie per il *warm-up* in base all'andamento delle prestazioni. Sebbene queste soluzioni abbiano introdotto maggiore flessibilità, presentano comunque sfide legate alla loro **complessità computazionale** e alla difficoltà di applicazione in scenari reali, dove le condizioni di carico e le interazioni tra componenti del sistema possono variare significativamente. Inoltre, l'assenza di robustezza in contesti caratterizzati da un'elevata variabilità rappresenta un ulteriore ostacolo alla loro adozione su larga scala.

Infine, va considerato il panorama in rapida evoluzione delle tecniche di **performance testing**. Recenti sviluppi in ambito accademico e industriale hanno portato alla proposta di approcci innovativi basati su **euristiche dinamiche**, **algoritmi di machine learning** e **analisi statistiche avanzate**. Tuttavia, tali soluzioni presentano spesso limiti pratici legati alla **complessità computazionale** o alla necessità di configurazioni specifiche, riducendone l'applicabilità in scenari reali.

Nel frattempo, diversi strumenti e approcci si sono concentrati sull'analisi delle prestazioni a livello di sistema. Questi si focalizzano principalmente sulla misurazione dell'efficienza complessiva delle applicazioni distribuite, senza affrontare specificamente il problema della gestione della fase di *warm-up*. Sebbene utili per analisi globali, tali strumenti lasciano una lacuna importante nella gestione delle

prestazioni nella fase iniziale di stabilizzazione, che è fondamentale per garantire l'affidabilità dei risultati.

Alla luce di queste problematiche, emerge chiaramente la necessità di un approccio che non solo garantisca risultati **affidabili** e **accurati**, ma che al contempo sia **efficiente** in termini di utilizzo delle risorse e tempi di esecuzione. È qui che entra in gioco l'Intelligenza Artificiale (IA), una disciplina che ha dimostrato il suo potenziale nel migliorare molteplici aspetti dello sviluppo e del testing software. In particolare, l'applicazione di **modelli avanzati di classificazione di serie temporali** (*Time Series Classification, TSC*) rappresenta un'opportunità unica per affrontare il problema del warm-up in modo innovativo.

L'obiettivo principale di questa tesi è dunque **colmare la lacuna** esistente proponendo un framework basato sull'IA per la **gestione dinamica della fase di warm-up**. Questo framework sfrutta modelli di *Time Series Classification* (approfonditi nella sezione 2.4.1) per analizzare i dati di prestazione in tempo reale e identificare automaticamente il momento in cui l'esecuzione del benchmark raggiunge **lo steady state**. L'approccio che proponiamo mira a ridurre al minimo il numero di iterazioni necessarie, garantendo al contempo una **maggiore accuratezza** e una significativa **riduzione dei tempi di testing** rispetto ai metodi tradizionali e avanzati. In questo modo, il framework non solo migliora l'efficienza complessiva del processo di benchmarking, ma lo rende anche più accessibile e applicabile a una varietà di scenari pratici.

1.3 Risultati ottenuti

La soluzione proposta è stata implementata e validata attraverso un ampio set di esperimenti utilizzando tecniche di microbenchmarking e applicazioni basate su architetture a microservizi. Inoltre, è stato sviluppato un **meccanismo semi-automatizzato**, arricchito da API per renderne facile l'utilizzo.

Il framework utilizza l'IA per monitorare e interpretare il comportamento delle metriche di latenza, identificando automaticamente il momento in cui un endpoint di un microservizio ha raggiunto lo **stato stabile**.

Sono stati utilizzati i modelli Fully Convolutional Network (FCN) e Random Convolutional Kernel Transform (ROCKET), essi sono stati addestrati sullo stesso dataset, garantendo condizioni di valutazione uniformi. Questa scelta ha permesso un confronto equo tra i due approcci e ha reso possibile l'analisi comparativa dei risultati ottenuti.

L'analisi dei dati ha permesso di valutare nel dettaglio le prestazioni dei modelli di *Machine Learning* impiegati e di testare l'efficacia del framework proposto.

Entrambi i modelli analizzati, *Fully Convolutional Network* (FCN) e *RandOm Convolutional Kernel Transform* (ROCKET), hanno ottenuto un'accuratezza del **98%**, con un numero ridotto di falsi positivi e falsi negativi. Tuttavia, si sono riscontrate alcune differenze tra i due approcci: mentre il modello FCN ha dimostrato un **miglior recall** sulla classe *Stable*, ROCKET ha evidenziato una **minore propensione ai falsi positivi**.

Il framework sviluppato ha dimostrato di essere in grado di individuare con precisione il termine della fase di *warm-up*, permettendo così un'analisi più affidabile e automatizzata. Il confronto tra i tempi stimati dai modelli e quelli misurati manualmente ha confermato la capacità del sistema di **generalizzare** anche su endpoint non utilizzati durante la fase di addestramento. Inoltre, il framework ha dimostrato la sua **adattabilità** alle caratteristiche dei diversi endpoint, migliorando la precisione della stima del tempo di stabilizzazione.

Rispetto agli approcci basati su euristiche dinamiche, il framework basato su *Machine Learning* ha evidenziato una maggiore affidabilità e adattabilità. Mentre le tecniche euristiche si basano su soglie fisse o regole predefinite, il framework è in grado di apprendere autonomamente i pattern delle serie temporali, adattandosi ai diversi scenari senza richiedere configurazioni manuali.

Infine, un ulteriore vantaggio dell'uso di questo framework è la significativa riduzione del tempo totale di testing rispetto ai metodi manuali, garantendo così una valutazione più efficiente delle prestazioni del sistema.

1.4 Struttura della tesi

Il resto di questa tesi è strutturato nel seguente modo. Il **Capitolo 2** fornisce un'analisi dello stato dell'arte, presentando le principali metodologie per il performance

testing di sistemi a microservizi e i criteri adottati per determinare la fase di warm-up.

Il **Capitolo 3** descrive il metodo di ricerca adottato, illustrando il processo di raccolta e analisi dei dati, nonché l'integrazione di tecniche di machine learning per l'ottimizzazione della fase di warm-up.

Nel **Capitolo 4** vengono analizzati i dati sperimentali, valutando il contesto degli esperimenti, la metodologia seguita e le metriche utilizzate per misurare le prestazioni del framework proposto.

Il **Capitolo 5** presenta i risultati ottenuti, confrontando l'approccio sviluppato con lo stato dell'arte e analizzando le prestazioni dei modelli impiegati.

Il **Capitolo 6** discute le principali minacce alla validità della ricerca, esaminando i fattori che potrebbero influenzare l'affidabilità e la generalizzazione dei risultati.

Infine, il **Capitolo 7** sintetizza le conclusioni dello studio e propone possibili sviluppi futuri per migliorare ulteriormente l'identificazione automatizzata dello steady state nei test di performance.

CAPITOLO 2

Background e stato dell'arte

Questo capitolo presenta lo stato dell'arte e una panoramica degli studi esistenti in letteratura che costituiscono il punto di partenza per questo lavoro di tesi.

2.1 Performance testing per sistemi a microservizi

Il *Performance Testing* è una pratica fondamentale nello sviluppo software, progettata per garantire che un sistema sia in grado di rispondere in modo efficace alle esigenze di carico e alle richieste degli utenti. Per le applicazioni basate su architetture a microservizi, il performance testing assume una rilevanza ancora maggiore, poiché la complessità intrinseca di queste architetture introduce una serie di sfide uniche. Ogni microservizio opera in modo indipendente, spesso con diverse tecnologie e linguaggi di programmazione, e interagisce con altri componenti tramite API o meccanismi di comunicazione asincroni. Questo rende il sistema complessivo altamente flessibile, ma anche suscettibile a inefficienze e colli di bottiglia. Il **Performance Testing** [3] viene effettuato per garantire:

- **Prestazioni adeguate**: Il sistema deve rispondere rapidamente alle richieste degli utenti, mantenendo bassa la latenza anche in condizioni di carico elevato.

- **Scalabilità**: Dimostrare che il sistema può gestire un numero crescente di richieste o utenti, scalando orizzontalmente (aggiungendo nodi) o verticalmente (aumentando le risorse dei nodi esistenti).
- **Resilienza**: Verificare che il sistema possa mantenere operatività e qualità del servizio anche in caso di guasti o comportamenti imprevisti.
- **Utilizzo ottimale delle risorse**: Identificare e ottimizzare l'uso di risorse come CPU, memoria, rete e I/O, riducendo i costi operativi.

Nonostante la sua importanza, il **Performance Testing** presenta diverse sfide, soprattutto nel contesto di architetture a microservizi:

- **Interazioni complesse tra microservizi**: Ogni microservizio può avere dipendenze da altri, quindi un problema prestazionale in un componente può propagarsi, rendendo difficile identificarne la causa principale.
- **Eterogeneità tecnologica**: Ogni componente può essere sviluppato in linguaggi diversi e utilizzare tecnologie differenti, complicando la creazione di test unificati e standardizzati.
- **Ambienti distribuiti**: I sistemi a microservizi operano tipicamente in ambienti distribuiti, come cluster Kubernetes o infrastrutture cloud, dove il comportamento del sistema può essere influenzato dalla latenza di rete, dalla capacità dei nodi e dall'allocazione dinamica delle risorse.
- **Rilevazione della fase di warm up e steady state**:È essenziale distinguere tra la fase di assestamento iniziale (*warm up*) e quella di stabilità (*steady state*) per garantire misurazioni accurate (concetti approfonditi nella sezione 2.1.1). Tuttavia, rilevare queste transizioni rappresenta una sfida tecnica complessa.
- **Scalabilità dei test**: Simulare migliaia o milioni di utenti simultanei richiede una quantità significativa di risorse computazionali e una configurazione attenta, soprattutto per i test distribuiti.
- **Validità dei risultati**: I dati raccolti durante i test devono rappresentare fedelmente le prestazioni reali del sistema in produzione, considerando fattori come picchi di traffico, utilizzo di risorse e comportamento sotto carico prolungato.

Data la complessità delle moderne architetture a microservizi, è essenziale integrare l'automazione nel processo di **Performance Testing**. Per:

- **Simulare scenari realistici**: Generando carichi di lavoro personalizzati che replicano condizioni di produzione.
- **Rilevare anomalie**: Identificando comportamenti anomali nelle metriche di prestazione, come picchi di latenza o variazioni improvvise nel throughput.
- **Ottimizzare i tempi di testing**: Automatizzando configurazione, esecuzione e analisi, riducendo la durata dei test senza comprometterne l'affidabilità.

Tuttavia, per garantire che i test forniscano risultati affidabili, è cruciale utilizzare strumenti che combinino approcci tradizionali con tecniche di **analisi delle serie temporali** e **machine learning**, per rilevare automaticamente il raggiungimento dello *steady state* e ridurre il numero di iterazioni necessarie.

2.1.1 Warm-up e Steady State

Nel contesto del *Performance Testing*, la comprensione e la corretta gestione delle fasi di *warm-up* e *steady state* sono fondamentali per garantire che le misurazioni raccolte siano affidabili e rappresentative delle reali prestazioni del sistema [4]. La mancata distinzione tra queste due fasi può portare a conclusioni errate, con implicazioni significative per l'ottimizzazione e il dimensionamento del sistema. Durante la fase di *warm-up*, qualsiasi programma in esecuzione attraversa una fase iniziale di riscaldamento, durante la quale il sistema si stabilizza e raggiunge un funzionamento ottimale. Questa fase può influenzare diversi aspetti delle prestazioni, tra cui latenza, throughput e utilizzo delle risorse, a causa di una serie di processi a livello di sistema. Tra i principali fattori che contribuiscono a questa fase ci sono:

- **Ottimizzazione JIT (Just-In-Time)**: Nei linguaggi runtime-based, il compilatore JIT analizza e ottimizza dinamicamente il codice, eseguendo compilazioni incrementali. Questo processo migliora l'efficienza nel lungo termine, ma può introdurre una variabilità iniziale nelle prestazioni.

- **Caching iniziale**: Il caricamento in cache di dati, configurazioni o risultati intermedi avviene tipicamente durante il warm-up. Questo può temporaneamente influenzare la latenza delle richieste e causare fluttuazioni nelle misurazioni delle prestazioni.
- **Allocazione e inizializzazione delle risorse**: Operazioni come l'apertura di connessioni a database, il caricamento di moduli software o il provisioning di risorse in ambienti cloud possono richiedere tempo e introdurre ulteriori variazioni nelle prestazioni iniziali.

Le misurazioni raccolte durante il warm-up non sono rappresentative delle prestazioni effettive del sistema, poiché riflettono un comportamento **transitorio** piuttosto che uno stato operativo consolidato. Una volta completata la fase di warm-up, il sistema entra nella fase di **stabilizzazione** (*steady state*), caratterizzata da metriche di prestazione **consistenti** e **prevedibili**. Durante questa fase, si osserva una progressiva stabilizzazione del sistema. In particolare, le **fluttuazioni** nelle metriche tendono a ridursi, con parametri come **latenza** e **throughput** che raggiungono valori **stabili**, rappresentando il comportamento operativo del sistema sotto carico. Inoltre, le **risorse** vengono utilizzate in modo **ottimale**, poiché l'inizializzazione è completata e il sistema opera in conformità con i requisiti di **prestazione** previsti. Infine, le misurazioni raccolte in questo stato sono **attendibili** e possono essere impiegate per valutare l'**efficienza** del sistema, individuare eventuali **colli di bottiglia** e supportare decisioni mirate all'**ottimizzazione** delle prestazioni. La capacità di identificare il passaggio dallo stato **transitorio** del **warm-up** allo *steady state* è fondamentale per garantire l'**affidabilità** dei test di prestazione.

2.1.2 Benchmarks per il testing end-to-end

Il **testing end-to-end** per sistemi a microservizi richiede strumenti e framework progettati per **simulare traffico realistico**, analizzare il comportamento sotto carico e identificare **colli di bottiglia** a livello sia di sistema che di singoli componenti. Negli ultimi anni, sono stati sviluppati diversi approcci per rispondere a queste esigenze.

Uno strumento ampiamente utilizzato per il testing delle applicazioni distribuite è **Apache JMeter**, che consente di configurare **scenari complessi**, simulare utenti

simultanei e raccogliere metriche dettagliate sulle prestazioni del sistema. JMeter permette di valutare il comportamento dei microservizi in scenari di produzione, concentrandosi sulle **interazioni tra i vari componenti** e sulla **resilienza del sistema** nel suo complesso. Un contributo significativo è rappresentato da framework progettati per il **benchmarking su larga scala** di **data store** e microservizi. Ad esempio, **DeathStarBench**[5] è una suite open-source che offre applicazioni su larga scala con decine di microservizi unici, permettendo di studiare effetti che emergono solo su larga scala, come la contesa della rete e le violazioni QoS a cascata dovute alle dipendenze tra i livelli. Un altro esempio è **NDBench**[6], uno strumento sviluppato da Netflix per valutare accuratamente data store e altri microservizi, progettato per essere distribuito in modo disaccoppiato con la capacità di modificare dinamicamente i parametri del benchmark durante l'esecuzione, consentendo iterazioni rapide su diversi test e modalità di guasto. Tuttavia, una limitazione comune è l'assenza di meccanismi automatici per rilevare il passaggio allo *steady state* durante la fase di **warm up**, richiedendo configurazioni manuali e aumentando la **complessità operativa**.

Infine, recenti progressi nell'applicazione dell'**intelligenza artificiale** al testing software, come l'uso di modelli per l'**analisi delle serie temporali** e il **rilevamento di anomalie**, hanno aperto nuove prospettive per il miglioramento dell'**efficienza** e dell'**affidabilità** del performance testing. Ad esempio, l'utilizzo di tecniche di *Time Series Classification* (TSC) consente di **monitorare dinamicamente** le metriche di prestazione e di **identificare automaticamente il raggiungimento dello steady state**, riducendo il numero di iterazioni e migliorando i **tempi complessivi del testing** [1].

2.2 Apache JMeter: uno strumento per il performance testing di sistema

Apache JMeter[1] è uno degli strumenti open-source più utilizzati per il *performance testing* di applicazioni web, API e sistemi distribuiti [7]. Sviluppato inizialmente dalla Apache Software Foundation, JMeter è progettato per simulare carichi di lavoro realistici e raccogliere metriche dettagliate sulle prestazioni dei sistemi sotto test. La

[1]Apache JMeter `https://jmeter.apache.org/`

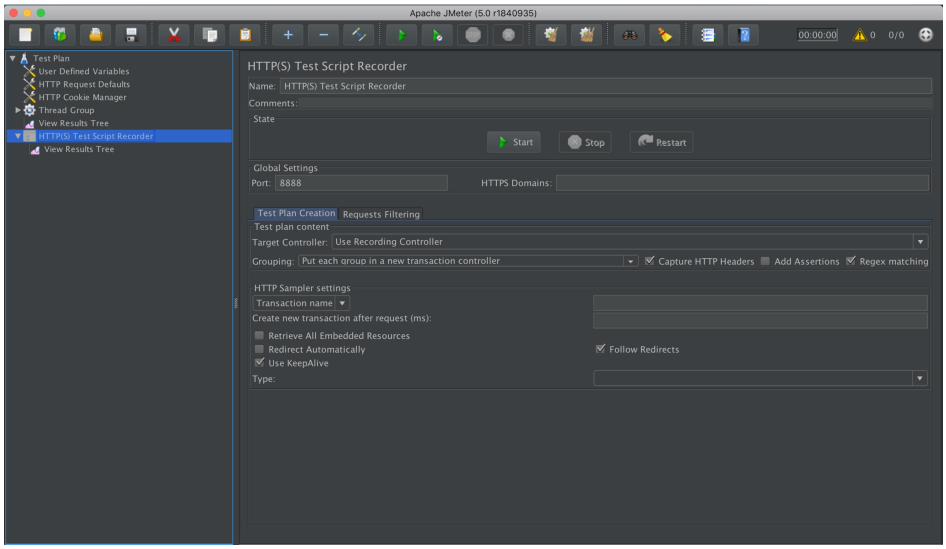

Figura 2.1: Interfaccia grafica dell'applicativo JMeter.

sua flessibilità e facilità d'uso lo rendono una soluzione estremamente popolare sia in ambito accademico che industriale, dove viene impiegato per analisi dettagliate delle prestazioni in ambienti complessi.

2.2.1 Caratteristiche principali

JMeter offre un'ampia gamma di funzionalità, che lo rendono uno strumento versatile per affrontare una varietà di scenari e requisiti:

- **Simulazione di carichi di lavoro:** Consente di simulare migliaia di utenti virtuali (*virtual users*) simultanei, replicando condizioni realistiche di produzione per testare la capacità del sistema di gestire carichi elevati. Questa funzionalità è particolarmente utile per valutare scalabilità e affidabilità dei sistemi distribuiti.

- **Supporto multi-protocollo:** JMeter supporta una vasta gamma di protocolli, tra cui HTTP, HTTPS, SOAP, REST, FTP, JDBC e molti altri. Questa caratteristica lo rende applicabile a una varietà di contesti, che spaziano dalle applicazioni web tradizionali a sistemi distribuiti complessi basati su microservizi.

- **Interfaccia grafica intuitiva:** La GUI di JMeter (mostrata in figura 2.1) consente agli utenti di progettare e configurare scenari di test in modo interattivo. Anche gli utenti meno esperti possono creare test dettagliati grazie a questa interfaccia user-friendly, che include opzioni per definire carichi, durate e punti di verifica.

- **Raccolta di metriche dettagliate:** JMeter offre un'analisi approfondita delle prestazioni, raccogliendo dati relativi a **latenza**, **throughput**, **tassi di errore** e altre metriche chiave. Questi dati possono essere visualizzati tramite grafici, tabelle o esportati per ulteriori analisi, consentendo agli sviluppatori di identificare colli di bottiglia e problemi di prestazioni.

- **Supporto allo scripting:** Uno dei principali punti di forza di JMeter è la possibilità di personalizzare i test tramite scripting. Linguaggi come BeanShell, Groovy o JavaScript possono essere utilizzati per automatizzare scenari complessi, definire logiche condizionali o modellare test specifici per applicazioni distribuite, come microservizi. Questa funzionalità consente di superare i limiti dei test preconfigurati e di adattarsi a esigenze specifiche, rendendolo uno strumento ideale per scenari complessi come quelli trattati in questa tesi.

2.2.2 Architettura e funzionamento

L'architettura di JMeter [8] è progettata per essere modulare e flessibile, consentendo di adattare lo strumento a diversi tipi di test:

- **Thread Groups:** Rappresentano la base di ogni test di JMeter e definiscono il numero di utenti virtuali, il tempo di *ramp-up* (tempo necessario per avviare tutti gli utenti) e il numero totale di iterazioni. Questa configurazione permette di modellare carichi realistici e progressivi.

- **Samplers:** Specificano le richieste da inviare al sistema, come richieste HTTP, JDBC o SOAP. Ogni sampler rappresenta un'operazione che un utente virtuale esegue durante il test.

- **Listeners:** Raccoglie e visualizza i risultati dei test. I listener offrono una varietà di formati di output, come grafici, tabelle e file di log, che possono essere utilizzati per analisi dettagliate.

- **Pre e Post Processors:** Questi componenti consentono di manipolare dati o richieste prima e dopo l'esecuzione di un sampler, aggiungendo ulteriore flessibilità e personalizzazione.

- **Controllers:** Permettono di definire la logica dei test, come cicli condizionali o operazioni sequenziali, aumentando l'accuratezza dei carichi simulati.

2.2.3 Perché è stato scelto JMeter?

Abbiamo scelto **Apache JMeter** come strumento per il performance testing in quanto rappresenta una delle soluzioni più consolidate e versatili nel settore. La sua ampia diffusione, sia in ambito industriale che accademico, ne testimonia l'affidabilità e l'efficacia nell'analisi delle prestazioni di sistemi distribuiti. Uno degli aspetti chiave che ha guidato questa scelta è la sua capacità di supportare una vasta gamma di protocolli, rendendolo adatto non solo per il testing di applicazioni web e API REST, ma anche per sistemi più complessi basati su microservizi e database distribuiti.JMeter offre anche un'interfaccia intuitiva che semplifica la configurazione degli scenari di test, pur permettendo un alto livello di personalizzazione grazie al supporto per scripting avanzato con linguaggi. Questo consente di modellare test sofisticati, adattandoli a casi d'uso specifici e superando i limiti delle configurazioni standard.Infine, la sua natura open-source e l'ampia community di supporto garantiscono una costante evoluzione dello strumento e una vasta disponibilità di risorse, documentazione ed estensioni che ne ampliano le funzionalità. Per tutte queste ragioni, **JMeter è risultato essere la scelta ideale per questo studio**, in quanto consente di effettuare un'analisi approfondita delle prestazioni di sistemi complessi, mantenendo al contempo un alto grado di flessibilità e scalabilità.

2.3 Selezione dei tempi di warm-up: una panoramica degli approcci più comuni

In questa sezione viene esaminato il ruolo centrale della fase di warm up nei test di prestazioni. Successivamente, vengono analizzati gli approcci più comuni attualmente utilizzati per affrontare le sfide legate alla selezione dei tempi di warm up [9], con riferimenti alla letteratura scientifica e alle principali tecniche applicate.

2.3.1 Approcci basati su iterazioni fisse

Uno degli approcci più semplici e tradizionali per gestire la fase di warm up consiste nell'utilizzare un numero fisso di iterazioni prima di iniziare la raccolta delle misurazioni. In questo metodo, il sistema viene sottoposto a un certo numero predeterminato di richieste per consentire la stabilizzazione delle prestazioni. Questo approccio è largamente utilizzato in ambienti in cui la semplicità di implementazione è una priorità e non sono richieste configurazioni complesse.

Un esempio significativo dell'uso di approcci a iterazioni fisse si trova nello studio di Laaber et al. [10], in cui gli autori analizzano le limitazioni di metodi statici applicati a scenari di benchmarking. Lo studio evidenzia come l'adozione di un numero fisso di iterazioni possa risultare in una sovrastima o sottostima della durata del warm up, compromettendo rispettivamente l'efficienza o l'accuratezza del processo di testing. Tuttavia, la rigidità intrinseca di questo metodo ne evidenzia i limiti. Da un lato, una sovrastima delle iterazioni può comportare un aumento superfluo dei tempi di testing e uno spreco di risorse computazionali. Dall'altro lato, una sottostima può portare a misurazioni inaccurate, poiché i dati raccolti durante la fase transitoria potrebbero riflettere comportamenti non rappresentativi del sistema. Nonostante ciò, il metodo delle iterazioni fisse rimane una scelta popolare, soprattutto in contesti meno complessi, dove la sua semplicità lo rende facilmente applicabile.

2.3.2 Approcci dinamici basati su euristiche

Per superare i limiti degli approcci basati su iterazioni fisse, sono stati sviluppati metodi dinamici che utilizzano euristiche per determinare automaticamente la fine della fase di warm up. Questi approcci si basano sul monitoraggio di metriche come la latenza delle richieste, l'utilizzo della CPU o della memoria, e adottano regole predefinite per identificare il momento in cui il sistema può essere considerato stabile. Una pratica comune è l'adozione di soglie specifiche: se una metrica rimane entro un intervallo predefinito per un determinato periodo, si assume che il sistema abbia raggiunto lo stato stazionario. Un esempio rilevante è lo studio di Georges et al. [11], in cui viene proposto un approccio basato sull'analisi della stabilità delle metriche di prestazione attraverso il monitoraggio dinamico di variazioni successive.

Gli autori suggeriscono l'uso di euristiche per adattare la durata del warm-up in base al comportamento osservato, con l'obiettivo di migliorare l'affidabilità delle misurazioni senza introdurre sovraccarichi computazionali significativi.

Sebbene questi approcci siano più flessibili rispetto ai metodi tradizionali, presentano alcune difficoltà. La definizione delle soglie appropriate è spesso complessa e richiede una conoscenza approfondita delle caratteristiche specifiche del sistema. Inoltre, come evidenziato nello studio citato, le euristiche possono risultare non sufficientemente robuste in scenari con elevata variabilità delle prestazioni, come nel caso di sistemi distribuiti o di applicazioni a microservizi.

2.3.3 Approcci basati su analisi statistiche

Un altro importante filone di ricerca è rappresentato dagli approcci che utilizzano tecniche di analisi statistica per identificare il momento in cui il sistema raggiunge uno stato stabile. Questi metodi si distinguono per la loro capacità di analizzare i dati raccolti durante la fase di warm up attraverso strumenti matematici e modelli statistici, al fine di individuare i pattern di stabilizzazione delle prestazioni. Le tecniche più comuni includono l'uso della media mobile, l'analisi della varianza o il confronto tra iterazioni consecutive, che consentono di rilevare variazioni significative nei parametri di interesse.

Ad esempio, Georges et al. [11] hanno proposto un approccio che si basa sull'osservazione della convergenza delle prestazioni attraverso variazioni ridotte tra iterazioni successive. In questo lavoro, gli autori hanno dimostrato che monitorare i cambiamenti progressivi nelle metriche consente di identificare in modo affidabile la transizione dalla fase di warm up a quella stazionaria, migliorando così l'accuratezza dei dati raccolti durante il benchmarking. Lo studio sottolinea come l'adozione di modelli statistici sia particolarmente efficace per mitigare gli effetti di comportamenti transitori che potrebbero altrimenti compromettere la qualità delle misurazioni.

Questi metodi offrono una maggiore precisione rispetto agli approcci basati su iterazioni fisse o su euristiche. Essi sono particolarmente utili in scenari complessi in cui la variabilità delle prestazioni non può essere facilmente catturata da regole statiche o soglie predefinite. Tuttavia, presentano alcune sfide significative. In parti-

colare, la complessità computazionale associata all'elaborazione statistica dei dati può rappresentare un ostacolo in ambienti distribuiti, caratterizzati da grandi volumi di richieste e metriche. Ad esempio, in contesti come le architetture a microservizi, l'analisi di un ampio numero di endpoint e il monitoraggio continuo delle loro prestazioni possono richiedere un'infrastruttura computazionale avanzata per supportare il carico analitico. Un altro aspetto critico di questi approcci riguarda la loro scalabilità. Come osservato da Laaber et al. [10], la complessità dei sistemi moderni, combinata con l'interdipendenza tra i loro componenti, può ridurre l'efficacia dei modelli statistici tradizionali. La difficoltà di applicare questi metodi a sistemi eterogenei e dinamici richiede spesso un adattamento delle tecniche standard, con conseguenti costi aggiuntivi in termini di sviluppo e configurazione.

Nonostante queste limitazioni, gli approcci statistici offrono vantaggi significativi, tra cui una maggiore capacità di adattarsi alle fluttuazioni delle prestazioni e di fornire dati più accurati rispetto ai metodi statici. La loro applicabilità risulta particolarmente evidente in ambiti accademici o in ambienti industriali altamente regolamentati, dove la precisione e l'affidabilità delle misurazioni sono fondamentali.

2.4 Applicazione dell'intelligenza artificiale nel performance testing

L'utilizzo dell'Intelligenza Artificiale (IA) nel *performance testing* consente di affrontare la crescente complessità dei sistemi moderni, come le architetture a microservizi. Gli algoritmi di *machine learning*, e in particolare quelli orientati all'analisi di serie temporali, offrono strumenti avanzati per automatizzare e ottimizzare la gestione delle fasi di warm up e steady state, riducendo i tempi di configurazione e migliorando la precisione delle misurazioni.

Tra le tecniche più rilevanti si distingue la **classificazione di serie temporali** (*Time Series Classification, TSC*), che si concentra sull'analisi di dati temporali per individuare pattern distintivi che rappresentano il comportamento del sistema. A questa si aggiungono approcci avanzati come **ROCKET** (*RandOm Convolutional KErnel Transform*) e *Fully Convolutional Networks* (FCN), che utilizzano tecniche di

deep learning per estrarre caratteristiche significative dalle serie temporali.

Recentemente, l'efficacia dei modelli di TSC per la gestione dinamica del *warm up* nei microbenchmark Java è stata esplorata attraverso un framework basato su intelligenza artificiale. Questo framework è stato testato su un ampio dataset di misurazioni di performance raccolte da **JMH microbenchmarks**, utilizzando tecniche di classificazione per prevedere il raggiungimento dello *steady state* e interrompere dinamicamente le iterazioni di *warm up*. I risultati hanno dimostrato miglioramenti significativi rispetto alle tecniche tradizionali e allo *state-of-the-art*, riducendo i tempi di esecuzione senza compromettere la qualità dei risultati [1].

2.4.1 Classificazione di serie temporali (TSC)

La classificazione di serie temporali (*Time Series Classification, TSC*) è una tecnica che associa etichette predefinite a sequenze temporali ordinate, sfruttando pattern distintivi e caratteristiche intrinseche dei dati. Applicata al performance testing, consente di analizzare metriche di prestazione come latenza e throughput per identificare automaticamente il momento di transizione dal warm up allo steady state.

Un esempio dell'applicazione della classificazione di serie temporali nel contesto del performance testing è presentato nello studio di "Ai-driven Java performance testing"[1], in cui gli autori propongono un framework basato sull'intelligenza artificiale che utilizza modelli di classificazione di serie temporali per determinare dinamicamente la fine della fase di warm-up durante l'esecuzione di microbenchmark su applicativi Java. Tra gli approcci più utilizzati per la TSC [12] figurano:

- **Metodi basati sulla distanza:** Tecniche come la *Dynamic Time Warping* (DTW) confrontano serie temporali in base alla loro similarità. Sebbene efficaci in alcuni contesti, questi metodi risultano computazionalmente costosi su dataset di grandi dimensioni.

- **Approcci basati su dizionario:** Algoritmi come SAX (*Symbolic Aggregate approXimation*) convertono i dati in rappresentazioni simboliche, facilitando l'individuazione di pattern ripetuti e anomalie.

- **Modelli convoluzionali e di deep learning:** Approcci avanzati come ROCKET e FCN sfruttano trasformazioni convoluzionali per estrarre caratteristiche significative e identificare pattern complessi nei dati temporali. Questi modelli sono particolarmente efficaci nell'estrazione automatica di feature significative, grazie all'applicazione di filtri convoluzionali lungo la dimensione temporale.

Nel contesto di questa ricerca, si è scelto di seguire l'approccio proposto da Traini et al. [1], che dimostrano come l'uso della classificazione di serie temporali, in combinazione con modelli convoluzionali, sia particolarmente efficace nel performance testing di sistemi complessi.

2.4.2 ROCKET: RandOm Convolutional KErnel Transform

ROCKET [13] rappresenta un approccio innovativo per la classificazione di serie temporali, caratterizzato dall'uso di kernel convoluzionali casuali per estrarre caratteristiche dai dati temporali. Questo metodo si distingue per la sua elevata efficienza computazionale e la semplicità nella costruzione del modello. Il funzionamento di ROCKET si basa su:

- **Generazione casuale di kernel:** Vengono generati migliaia di kernel convoluzionali con parametri casuali, applicati ai dati per estrarre pattern locali.
- **Feature extraction:** I risultati delle convoluzioni producono due caratteristiche principali: il valore massimo (*max*) e la proporzione di valori positivi (*PPV*) derivati da ciascun kernel.
- **Classificazione:** Le feature generate vengono utilizzate come input per un classificatore lineare, come la regressione logistica, per la classificazione finale.

L'efficienza di ROCKET deriva dall'eliminazione della necessità di addestrare i kernel stessi, rendendolo particolarmente adatto per applicazioni in tempo reale su grandi dataset.

2.4.3 FCN: Fully Convolutional Networks

Le **Fully Convolutional Networks (FCN)** rappresentano un'architettura avanzata di reti neurali progettata specificamente per compiti di classificazione e segmentazione di serie temporali. A differenza delle reti neurali convoluzionali tradizionali (*Convolutional Neural Networks, CNN*), che utilizzano strati completamente connessi (*fully connected*) per produrre una classificazione globale, le FCN rimuovono questi strati finali, mantenendo informazioni temporali e spaziali lungo tutta la sequenza. Questo le rende particolarmente adatte a catturare pattern sia locali che globali presenti nei dati, garantendo maggiore flessibilità e precisione. L'architettura di una FCN è composta tipicamente da tre elementi principali:

- **Strati convoluzionali:** Questi strati sono progettati per applicare filtri convoluzionali lungo la dimensione temporale delle serie. Ogni filtro identifica pattern locali significativi, come variazioni rapide o tendenze di lungo termine, che possono caratterizzare diverse fasi del sistema (e.g., warm up e steady state). La profondità delle convoluzioni consente inoltre di catturare informazioni gerarchiche nei dati.

- **Global Average Pooling (GAP):** Questo meccanismo sostituisce gli strati completamente connessi tradizionali, riducendo le feature map a una rappresentazione compatta ma significativa. Il GAP calcola la media globale su tutte le dimensioni spaziali di ciascun filtro convoluzionale, mantenendo solo le informazioni più rilevanti per la classificazione. Questo approccio non solo migliora la generalizzazione del modello, ma riduce anche il rischio di overfitting, specialmente su dataset di piccole dimensioni.

- **Classificatore finale:** Un semplice strato completamente connesso o una funzione di softmax viene utilizzata per assegnare un'etichetta alla serie temporale in base alle feature estratte. Questo passaggio finale traduce le informazioni ad alta dimensionalità in una predizione interpretabile.

Un aspetto chiave che distingue le FCN è l'utilizzo delle **skip connections**, che collegano direttamente gli strati iniziali dell'architettura (encoder) con quelli finali (decoder). Queste connessioni consentono di combinare informazioni temporali

dettagliate, catturate nei primi strati, con rappresentazioni più astratte generate nei livelli profondi. Questo è particolarmente utile per preservare i dettagli locali durante la ricostruzione o la classificazione di sequenze temporali. Le FCN sono state ampiamente utilizzate in numerosi campi, tra cui:

- **Sanità:** Analisi di segnali fisiologici, come ECG (elettrocardiogrammi) e EEG (elettroencefalogrammi), per identificare anomalie o eventi critici.
- **Riconoscimento delle attività umane:** Utilizzo di dati da sensori indossabili per classificare movimenti e posture.
- **Predizione di serie temporali industriali:** Monitoraggio delle prestazioni di macchinari o impianti produttivi, con l'obiettivo di rilevare anomalie o identificare stati di degrado.

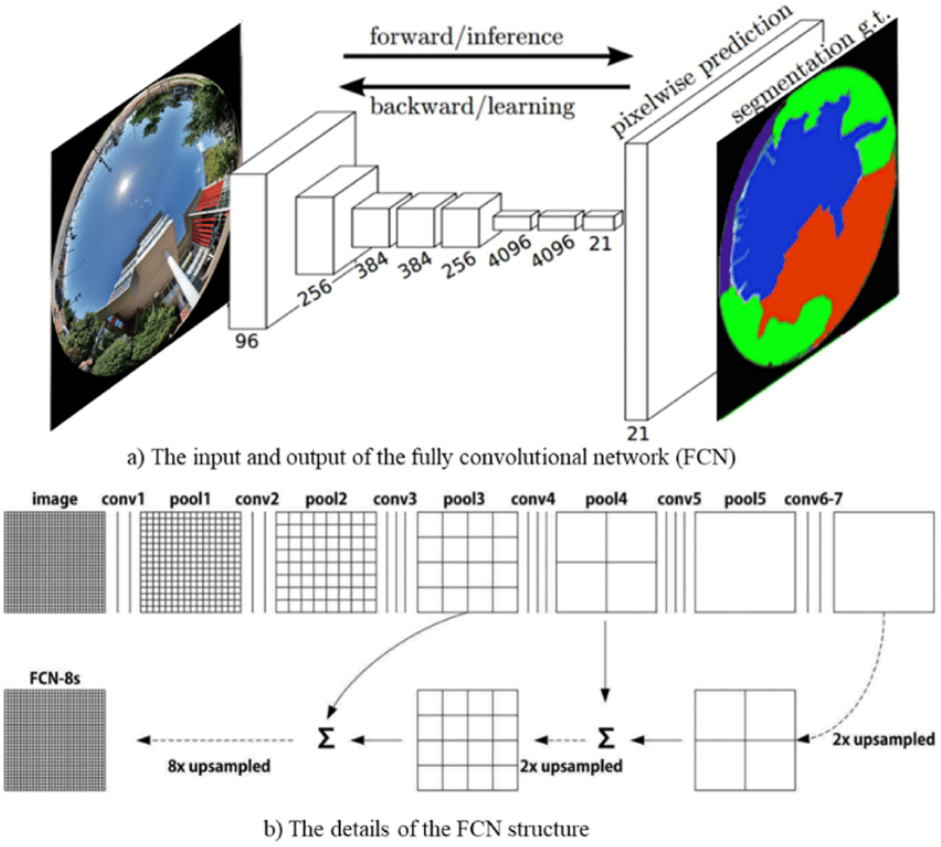

Figura 2.2: Architettura di una Fully Convolutional Network (FCN).

Un altro vantaggio delle FCN è la gestione di input variabili, ideale per serie temporali di diversa lunghezza. Questa flessibilità è cruciale nel *performance testing*, dove durata dei test e frequenza di campionamento possono variare.

Studi recenti, come quello di Wang et al. [14], hanno dimostrato che le FCN raggiungono prestazioni all'avanguardia su dataset standardizzati come l'archivio UCR [15]. In particolare, l'abilità delle FCN di estrarre automaticamente feature rilevanti, senza richiedere pre-elaborazioni complesse, rappresenta un significativo progresso rispetto ai metodi tradizionali. Questo è particolarmente utile per identificare il momento di transizione tra warm up e steady state, dove i pattern di comportamento del sistema possono essere altamente non lineari e variabili.

Nonostante la loro complessità computazionale, le FCN possono essere ottimizzate tramite tecniche come l' *batch normalization*, la regolarizzazione e l'utilizzo di hardware accelerato, come GPU. Questo le rende uno strumento potente e pratico per la classificazione di serie temporali anche in contesti di produzione industriale.

CAPITOLO 3

Un metodo per il calcolo del warm up necessario per il testing di performance di applicazioni basate su microservizi nel cloud

Questo capitolo affronta le problematiche discusse nel capitolo 2, proponendo una soluzione basata sull'integrazione dell'IA nel performance testing per affrontare le sfide legate ai microservizi. Viene analizzata la complessità di queste architetture e le difficoltà legate alla valutazione delle prestazioni, con particolare attenzione all'individuazione del momento in cui il sistema raggiunge una condizione stabile.

Per affrontare queste sfide, viene presentato un approccio basato sull'analisi delle serie temporali, che consente di determinare automaticamente la fine della fase di warm-up e l'inizio dello steady state. Vengono inoltre descritte le tecniche di raccolta e pre-elaborazione dei dati, nonché l'implementazione di modelli di Machine Learning per la classificazione dello stato del sistema. Infine, si analizza l'integrazione del framework sviluppato con strumenti di testing automatizzato, al fine di ottimizzare il processo di valutazione delle prestazioni.

3.1 Motivazioni

L'integrazione dell'Intelligenza Artificiale (IA) nel performance testing risponde alla crescente complessità dei sistemi basati su microservizi. Le architetture moder-

ne, caratterizzate da interdipendenze tra numerosi componenti, presentano sfide significative nella valutazione delle prestazioni.

Uno degli aspetti più critici è l'identificazione automatica del termine della fase di warm up, quando un sistema raggiunge lo stato stazionario e le misurazioni diventano affidabili. I metodi tradizionali, basati su un numero fisso di iterazioni o soglie empiriche, risultano inefficaci in scenari dinamici.

Per superare questi limiti, il framework proposto sfrutta tecniche di **Time Series Classification (TSC)** per analizzare metriche di prestazione in tempo reale e determinare automaticamente il passaggio allo *steady state*. In questo contesto, è essenziale un approccio basato sulla segmentazione dei dati temporali per poter riconoscere con precisione il momento in cui il sistema diventa stabile, evitando così di eseguire test di durata eccessiva o raccogliere dati non rappresentativi.

L'integrazione dei modelli di Machine Learning con **Apache JMeter** automatizza il rilevamento della fine del warm up nei test di performance. Il framework riceve dati durante l'esecuzione dei test e li analizza per determinare se il sistema ha raggiunto uno stato stabile. I dati raccolti vengono elaborati e classificati in tempo reale, consentendo di stabilire in modo dinamico il momento in cui il warm up può essere considerato concluso. Se il sistema è stabile, il test viene interrotto automaticamente; in caso contrario, l'esecuzione prosegue fino al raggiungimento dello stato desiderato. Questa integrazione permette di ottimizzare il processo di testing, riducendo l'intervento manuale e migliorando l'affidabilità delle misurazioni.

3.2 Fase di raccolta dati

Per l'addestramento e la valutazione del framework proposto, i dati sono stati raccolti attraverso l'esecuzione di microbenchmark su un sistema rappresentativo basato su microservizi. Questo sistema ha fornito un ambiente realistico per la generazione di dati relativi alle prestazioni, consentendo di analizzare metriche critiche come latenza, throughput e utilizzo delle risorse. Tutti i dati raccolti hanno dopodiché subito il processo di pulizia dei dati e pre-elaborazione dei dati prima di essere usati (processo spiegato dettagliatamente nelle sezioni 3.2.2 e 3.2.3).

La raccolta delle richieste e delle relative latenze è stata effettuata utilizzando **Apache JMeter**. I risultati sono stati successivamente esportati in un file XML per una facile manipolazione e analisi.

Formato dei Dati Raccolti: I dati raccolti da JMeter sono stati organizzati in un file XML strutturato, contenente informazioni dettagliate per ciascuna richiesta. Un esempio di registro XML è riportato nella Figura 3.1. Ogni elemento `<httpSample>` rappresenta una richiesta individuale e include i seguenti attributi:

- **t:** Tempo totale di risposta (in millisecondi).
- **lt:** Tempo di latenza (in millisecondi) per ricevere la prima risposta dal server.
- **ts:** Timestamp della richiesta.
- **s:** Stato della richiesta (*true* per successo, *false* per fallimento).
- **rc:** Codice di stato HTTP restituito dal server (ad esempio, 200 per successo).
- **java.net.URL:** URL dell'endpoint di destinazione della richiesta.

```
<httpSample t="168" it="0" lt="87" ct="0"
            ts="1736460912080"
            s="true"
            lb="HTTP Request"
            rc="200"
            tn="Thread Group 1-1"
            ng="1" na="1">
  <java.net.URL>http://192.168.31.35:32677/api
        /v1/travelservice/trips/left</java.net.URL>
</httpSample>
```

Figura 3.1: Esempio di registro XML generato da JMeter per la raccolta delle metriche.

3.2.1 Pulizia dei dati

Durante il processo di raccolta dei dati, è stato necessario eseguire una fase di pulizia e pre-elaborazione per garantire che i risultati fossero affidabili e rappresen-

tativi delle reali prestazioni del sistema. Poiché i dati grezzi includevano richieste con esiti non validi o latenza anomala, è stato adottato un approccio sistematico per rimuovere tali anomalie e ottenere un dataset coerente.

Filtraggio delle Richieste Non Valide: Tutte le richieste che hanno restituito codici di stato HTTP non associati a un'esecuzione corretta sono state identificate ed escluse. In particolare, sono stati rimossi i casi in cui il server ha risposto con errori quali **502 Bad Gateway**, **500 Internal Server Error** o altri codici simili, poiché indicano problemi di comunicazione o malfunzionamenti lato server, con il rischio di compromettere l'accuratezza dei dati relativi alla latenza.

Per garantire che l'analisi si basasse esclusivamente su misurazioni affidabili, sono state considerate unicamente le richieste con codice di stato **200**, in quanto rappresentano esecuzioni corrette e risposte valide del microservizio testato. Questo processo di selezione ha assicurato che il dataset finale fosse costituito da dati significativi per lo studio delle prestazioni. Dopo il processo di filtraggio, il dataset risultante contiene esclusivamente richieste che hanno soddisfatto i criteri di validità sopra descritti. Questo risultato è stato ottenuto grazie a due passaggi principali: (i) abbiamo ridotto il rumore nei dati, ottenendo così un set di misurazioni più affidabile; e (ii) abbiamo migliorato l'accuratezza dell'analisi eliminando valori che avrebbero potuto alterare la comprensione del comportamento degli endpoint.

3.2.2 Pre-elaborazione dei dati

La fase di pre-elaborazione dei dati ha come obiettivo la preparazione del dataset da utilizzare per l'addestramento del framework supervisionato basato su intelligenza artificiale. Questo processo parte dai dati grezzi raccolti tramite JMeter, che consistono in una serie temporale di misurazioni delle prestazioni per vari endpoint. Ogni misurazione include informazioni come il tempo di risposta (*latency*) e lo stato HTTP della richiesta, dati che rappresentano il comportamento del sistema in condizioni di carico simulato.

Identificazione dello Steady State tramite PELT: Il primo passo del processo consiste nell'applicazione dell'algoritmo PELT (*Pruned Exact Linear Time*) per individuare il punto di transizione dal warm up allo steady state. PELT rileva cambiamenti signi-

ficativi nella serie temporale analizzando le variazioni nelle metriche di prestazione. Il punto di transizione identificato rappresenta il momento in cui il sistema entra in una fase stabile, caratterizzata da misurazioni più consistenti.

Questo consente di suddividere la serie temporale in due fasi principali: la prima è la **fase di warm up**, caratterizzata da un'elevata variabilità e instabilità, mentre la seconda corrisponde allo **steady state**, in cui le misurazioni riflettono uno stato più stabile e affidabile. Un esempio è visibile in figura 3.2.

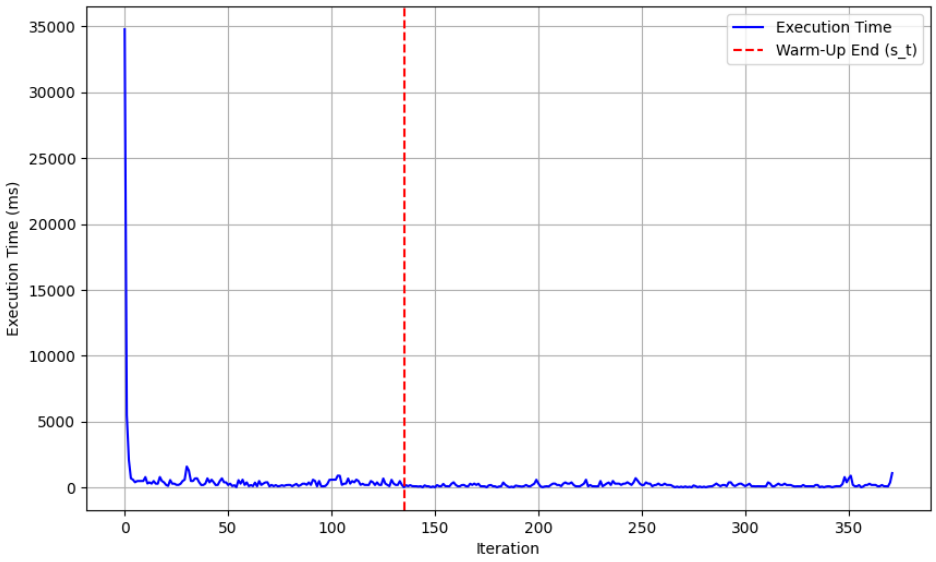

Figura 3.2: Esempio di identificazione dello steady state tramite PELT.

3.2.3 Segmentazione in blocchi e classificazione

Dopo aver identificato il punto di inizio dello steady state, la serie temporale viene segmentata in blocchi sovrapposti di dimensione fissa pari a 50 misurazioni ciascuno. Ogni blocco è costruito rimuovendo l'ultima misurazione del blocco precedente e aggiungendo una nuova misurazione in testa. In questo modo, ogni blocco rappresenta una finestra temporale scorrevole sulle prestazioni del sistema.

A ciascun blocco viene poi assegnata un'etichetta binaria basata sul punto di transizione identificato da PELT. Se il blocco contiene esclusivamente misurazioni effettuate a partire dal punto di transizione, e quindi ricade interamente nello steady state, viene classificato come **stabile**. Al contrario, se almeno una misurazione appar-

tiene alla fase precedente al punto di transizione, e quindi proviene dal warm up, il blocco viene considerato **instabile**. Questo processo è visibile in figura 3.3.

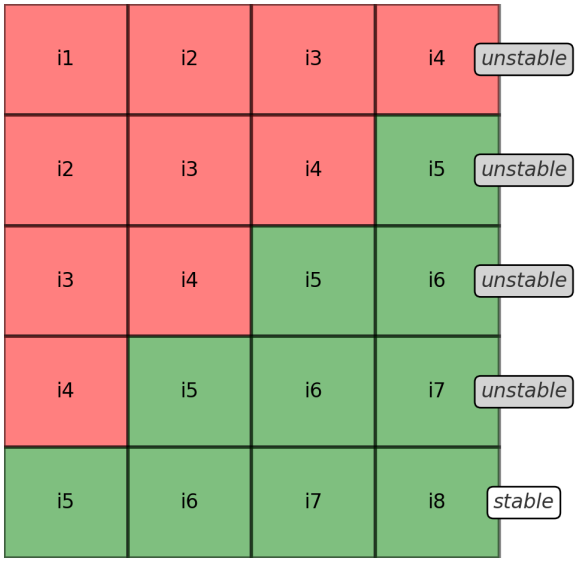

Figura 3.3: Processo di segmentazione della serie temporale e classificazione dei segmenti (immagine riadattata presa da [1])

Estrazione delle Metriche Statistiche: Per ciascun blocco vengono calcolate diverse metriche statistiche che sintetizzano il comportamento del sistema all'interno della finestra temporale considerata. Tra queste, il **valore medio** rappresenta la media delle misurazioni, mentre la **mediana** indica il valore centrale della distribuzione. La **deviazione standard** fornisce una misura della variabilità dei dati, permettendo di valutare quanto le misurazioni si discostino dal valore medio. Inoltre, vengono determinati il **valore minimo** e il **valore massimo** osservati all'interno del blocco, offrendo un'indicazione della gamma di variazione dei dati. Queste metriche, insieme all'etichetta di stabilità (*is_stable*), costituiscono il dataset utilizzato per addestrare i modelli di intelligenza artificiale.

Rappresentazione dei Dati: Ogni blocco generato è rappresentato come una riga in una tabella. Un esempio è mostrato nella tabella 3.1.

block_start	block_end	mean	median	std_dev	min	max	is_stable
132	181	148.34	105.0	96.60	27	491	False
133	182	146.24	104.0	96.55	27	491	False
134	183	144.12	104.0	89.75	27	402	False
135	184	144.3	104.0	89.87	27	402	True
136	185	145.8	104.0	88.52	27	402	True
137	186	142.76	104.0	89.92	27	402	True

Tabella 3.1: Esempio di blocchi segmentati dai dati raccolti con relative metriche e classificazione di stabilità (*is_stable*).

3.3 Modelli di Machine Learning utilizzati

Per affrontare il problema della classificazione dello stato del sistema, sono stati selezionati due modelli di Machine Learning particolarmente adatti all'analisi delle serie temporali: la RandOm Convolutional KErnel Transform (ROCKET) e il Fully Convolutional Network (FCN).

3.3.1 RandOm Convolutional KErnel Transform (ROCKET)

Per il metodo ROCKET (RandOm Convolutional KErnel Transform) sono stati generati 500 kernel convoluzionali di lunghezza variabile, ciascuno con pesi e bias assegnati casualmente. L'applicazione di tali kernel ai dati di input ha permesso di estrarre due tipi di feature principali: il massimo valore della convoluzione e la proporzione di valori positivi. Queste feature sono state poi concatenate per creare un nuovo spazio trasformato di input.

Il modello di classificazione utilizzato in ROCKET è un classificatore Ridge, addestrato con validazione incrociata su 5 fold e con alphas distribuiti in scala logaritmica da 10^{-3} a 10^{3}. Anche in questo caso, il dataset di training è stato bilanciato per evitare squilibri tra le classi, mentre il dataset di test è rimasto inalterato per valutare la robustezza del modello in un contesto realistico.

Per ulteriori dettagli su ROCKET, si rimanda alla sezione 2.4.2.

3.3.2 Fully Convolutional Network (FCN)

La Fully Convolutional Network (FCN) è stata utilizzata per individuare automaticamente pattern significativi all'interno delle feature numeriche fornite in input. Il modello implementato è stato costruito con tre strati convolutivi 1D con rispettivamente 128, 256 e 128 filtri, utilizzando la funzione di attivazione ReLU e il padding "same" per mantenere la lunghezza delle feature inalterata. Successivamente, un livello di pooling globale medio (GlobalAveragePooling1D) è stato impiegato per ridurre la dimensionalità e conservare le informazioni più rilevanti. Infine, il modello termina con uno strato denso con un singolo neurone e attivazione sigmoid, utile per la classificazione binaria.

Per garantire un efficace addestramento, i dati in input sono stati pre-processati attraverso la selezione di specifiche feature statistiche, tra cui la media, la deviazione standard, il valore minimo, massimo e la mediana. Inoltre, i dati del training set sono stati bilanciati per evitare uno sbilanciamento tra le classi stabili e instabili, mentre il test set è rimasto non bilanciato per simulare scenari reali. Dopo il bilanciamento, il dataset è stato suddiviso in un training set (80%) e un test set (20%). L'ottimizzazione del modello è stata effettuata utilizzando l'algoritmo Adam con la funzione di perdita binary cross-entropy. Il modello è stato addestrato per 50 epoche con un batch size di 32, riservando il 20% dei dati di training per la validazione. Per una spiegazione più dettagliata della FCN, si rimanda alla sezione 2.4.3.

3.4 Integrazione dei modelli di machine learning con JMeter

L'integrazione dei modelli di Machine Learning all'interno di **Apache JMeter** è stata realizzata attraverso un'API REST sviluppata con *FastAPI*, che funge da intermediario tra JMeter e i modelli di classificazione. Il framework implementato riceve finestre di misurazione da JMeter, analizza i dati in tempo reale e comunica dinamicamente a JMeter se interrompere o proseguire il test, determinando automaticamente il termine del warmup.

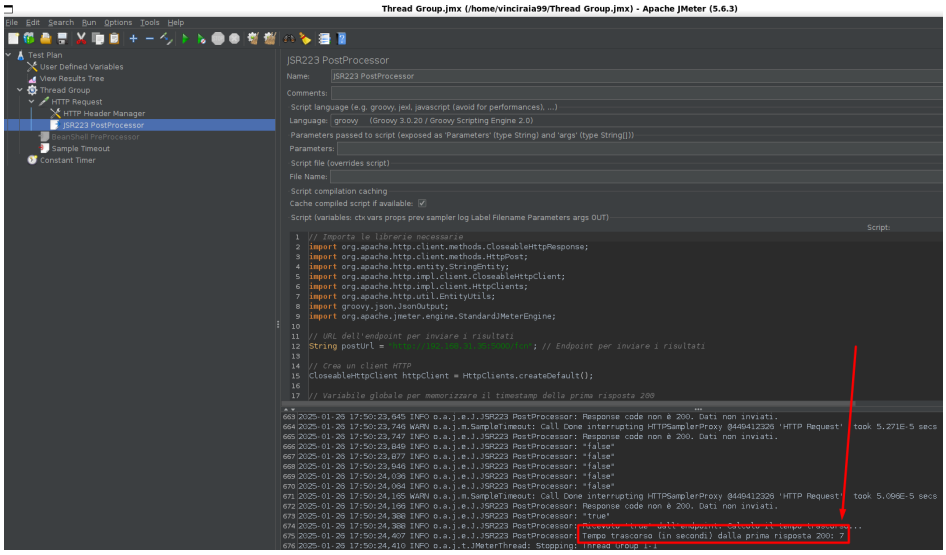

Figura 3.4: Esempio di esecuzione del framewrok

Durante l'esecuzione di un test, JMeter raccoglie metriche di latenza, throughput e stato delle richieste HTTP. Questi dati vengono inviati a un **PostProcessor JSR223**, che inoltra le informazioni all'API REST del framework. I dati ricevuti vengono processati e accumulati in un **buffer circolare** contenente le ultime 50 misurazioni. Una volta riempito il buffer, i dati vengono trasformati in un insieme di feature statistiche che vengono poi passate al modello di Machine Learning selezionato (FCN o ROCKET) per la classificazione.

Se il modello classifica il segmento di dati come appartenente allo *steady state*, l'API REST comunica a JMeter di **interrompere il test**, fornendo in output le ultime 50 misurazioni considerate stabili, che servono allo sviluppatore per vedere le prestazioni del microservizio e il tempo in secondi necessario per raggiungere lo steady state (un esempio è visibile in figura 3.4). Nel caso in cui il modello rilevi che il sistema è ancora nella fase di warm up, **il test continua fino a quando non viene raggiunto lo stato stabile**.

Questa integrazione consente di automatizzare il processo di individuazione del warm up **senza richiedere configurazioni manuali**, riducendo l'incertezza associata all'uso di soglie predefinite e migliorando la qualità dei dati raccolti.

L'intero codice del framework è disponibile su GitHub[1].

[1]Repository Github WarmUpFramework: https://github.com/vinciraia99/WarmUpFramework

CAPITOLO 4

Metodologia di raccolta e analisi dei dati

In questo capitolo viene introdotto il contesto sperimentale, vengono delineati i criteri impiegati per la costruzione del dataset, evidenziando le tecniche utilizzate per garantire la qualità e la rappresentatività delle informazioni raccolte. Inoltre, si definiscono le domande di ricerca e si approfondisce il ruolo delle metriche di valutazione nella caratterizzazione delle proprietà delle misurazioni e nella loro successiva interpretazione.

4.1 Obiettivi e domande di ricerca

L'obiettivo di questo studio è valutare l'efficacia e l'efficienza del framework basato su *Time Series Classification* (TSC) per l'analisi della fase di warm up nei test di performance delle applicazioni basate su microservizi. Il confronto viene effettuato rispetto ai metodi tradizionali e alle tecniche all'avanguardia. L'analisi si rivolge sia ai ricercatori, interessati a migliorare le tecniche esistenti di rilevamento dello *steady state*, sia ai professionisti, che necessitano di strumenti pratici ed efficienti per ottimizzare il processo di testing delle loro applicazioni. Per guidare la ricerca, formuliamo le seguenti domande di ricerca:

RQ1: Quanto sono efficaci i modelli di *Time Series Classification* nel classificare accuratamente le misurazioni di performance, distinguendo tra la fase di warm-up e lo *steady state*?

RQ2: Il framework di TSC è in grado di migliorare lo stato dell'arte rispetto ai metodi tradizionali per il rilevamento del warm-up?

Poiché le metriche utilizzate per valutare l'efficacia e l'efficienza delle strategie analizzate variano in base alla metodologia impiegata, l'analisi è articolata in più fasi. In primo luogo, descriviamo il metodo di ricerca seguito per confrontare il framework con gli approcci tradizionali, evidenziando i risultati ottenuti attraverso un'analisi empirica. Successivamente, approfondiamo il confronto tra il nostro approccio e le metodologie statistiche ed euristiche esistenti, valutandone i benefici, i limiti e il potenziale impatto nel contesto del *performance testing* su microservizi.

4.2 Contesto sperimentale

Gli esperimenti condotti in questo lavoro di ricerca sono stati progettati per valutare l'efficacia di un framework basato su Intelligenza Artificiale per l'ottimizzazione della fase di warm up nei test di performance su sistemi a microservizi. Per garantire condizioni di valutazione uniformi, gli esperimenti sono stati eseguiti su un ambiente controllato con specifiche hardware e software ben definite.

4.2.1 Setup sperimentale

Le sperimentazioni sono state condotte su un sistema caratterizzato da specifiche hardware e software ben definite. L'ambiente di esecuzione si basa su Windows 11 nella versione 10.0.26100 build 26100, con un processore AMD Ryzen 7 5800X e una scheda grafica NVIDIA RTX 3070. Il sistema dispone inoltre di 32 GB di memoria RAM DDR4 con una frequenza di 3600 MHz. L'implementazione è stata sviluppata in ambiente **Docker 24.0.7, build afdd53b**, assicurando un'esecuzione isolata e riproducibile degli esperimenti. Per quanto riguarda il software, le analisi

sono state eseguite utilizzando **Python 3.12.1** e librerie specifiche per il Machine Learning, tra cui `TensorFlow` nella versione **2.18.0** e `scikit-learn` nella versione **1.6.1**, oltre ad altre librerie per la gestione e l'elaborazione dei dati.

4.2.2 Train Ticket: un benchmark per distemi a microservizi

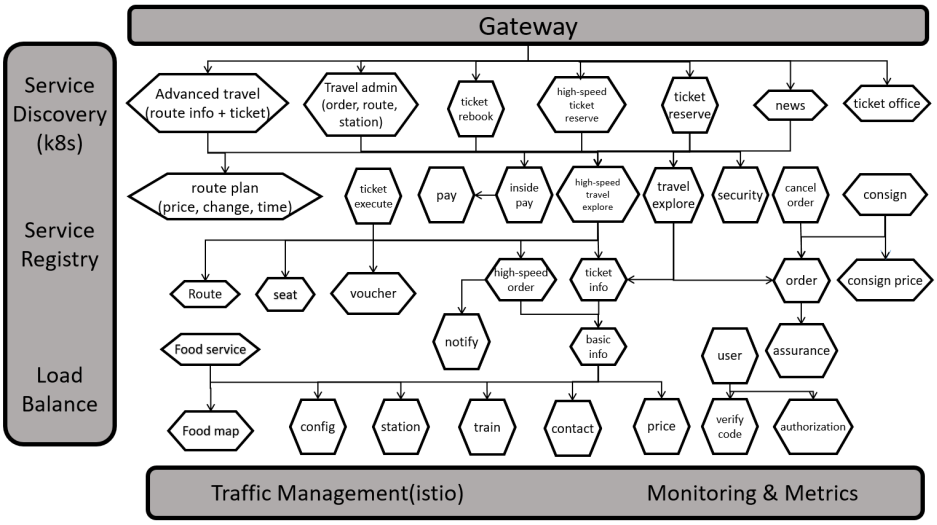

Figura 4.1: Architettura del sistema Train Ticket, composta da 41 microservizi.[2]

Per analizzare e validare le sfide sopra descritte, è stato utilizzato il sistema *Train Ticket*, un benchmark progettato specificamente per la ricerca su architetture a microservizi. Questo sistema rappresenta un'applicazione realistica di prenotazione di biglietti ferroviari, composta da 41 microservizi distinti. La varietà tecnologica e l'architettura modulare di *Train Ticket* lo rendono un caso d'uso ideale per valutare le problematiche legate al performance testing in ambienti complessi.

Tecnologie utilizzate: L'implementazione di *Train Ticket* è stata realizzata attraverso una combinazione di linguaggi di programmazione e framework, rispecchiando la diversità tecnologica tipica dei sistemi moderni. Per la creazione di servizi scalabili e affidabili, è stato utilizzato **Java** in combinazione con i framework *Spring Boot* e *Spring Cloud*. Parallelamente, **Node.js**, insieme al framework *Express*, è stato impiegato per la gestione di microservizi leggeri e ad alte prestazioni.

Alcuni componenti orientati ai dati sono stati sviluppati in **Python**, sfruttando le potenzialità del framework *Django*, mentre **Go**, in abbinamento a *Webgo*, è stato scelto per la creazione di microservizi particolarmente performanti. Per la gestione dei

dati, il sistema utilizza una combinazione di database relazionali e non relazionali, adottando *MySQL* per la gestione strutturata delle informazioni e *MongoDB* per garantire maggiore flessibilità nella memorizzazione dei dati non strutturati.

L'architettura di *Train Ticket* è altamente modulare, seguendo le migliori pratiche dell'architettura a microservizi. Ogni microservizio è autonomo e comunica con gli altri attraverso protocolli leggeri come HTTP e gRPC. Questa configurazione supporta la scalabilità orizzontale e facilita l'integrazione con strumenti di monitoraggio e tracciamento. L'architettura modulare di *Train Ticket* supporta la scalabilità orizzontale e facilita l'integrazione con strumenti di monitoraggio e tracciamento. *Train Ticket* è stato adottato per testare il framework proposto in scenari realistici di performance testing.

L'uso di *Train Ticket* nel contesto di questa ricerca ha fornito un terreno di prova per lo sviluppo e la validazione di un framework basato su IA, dimostrando come sia possibile migliorare l'efficienza del performance testing nei sistemi a microservizi. A differenza di altri benchmark disponibili, *Train Ticket* offre una combinazione unica di complessità architetturale, diversità tecnologica e scenari d'uso realistici, rendendolo un'opzione ideale per testare la capacità del framework di adattarsi a sistemi distribuiti con elevata eterogeneità. Inoltre, la sua ampia documentazione e il supporto da parte della comunità scientifica lo rendono un benchmark consolidato per esperimenti di ricerca accademica e industriale.

4.3 Organizzazione del dataset di train e test per FCN e ROCKET

Per l'addestramento e la valutazione dei modelli *Fully Convolutional Network* (FCN) e *ROCKET*, i dati raccolti dal sistema *Train Ticket* sono stati suddivisi in due insiemi distinti: un **dataset di training**, utilizzato per addestrare i modelli, e un **dataset di test**, impiegato per valutarne le prestazioni. Per garantire un equilibrio tra le classi durante l'addestramento, il dataset di training è stato bilanciato mediante un'operazione di *resampling*, uniformando il numero di esempi per la classe "stabile" (*steady state*) e "instabile" (*warm-up phase*). Tuttavia, il dataset di test non è stato

sottoposto a tale bilanciamento, in modo da riflettere la distribuzione reale dei dati nel contesto applicativo e valutare le prestazioni dei modelli in condizioni realistiche.

4.3.1 Preprocessing e feature extraction

Il preprocessing dei dati ha incluso diverse operazioni fondamentali per preparare i dati all'input nei modelli di machine learning. In primo luogo, è stata effettuata la selezione delle feature chiave, che comprendevano `mean`, `std_dev`, `min`, `max`, e `median`. Un esempio dei dati utilizzati per l'addestramento può essere visualizzato in figura 3.3. Successivamente, i valori sono stati convertiti in array NumPy, permettendo così un formato adeguato per l'input nei modelli. Infine, è stata applicata una normalizzazione dei dati al fine di migliorare la stabilità e l'efficacia del modello. La suddivisione in train e test è stata effettuata utilizzando *train-test split* con una proporzione dell'80% per il training e del 20% per il test.

4.3.2 Architettura del modello FCN e mitigazione overfitting

Il modello *Fully Convolutional Network* (FCN) è stato progettato con diversi strati convoluzionali, insieme a tecniche di batch normalization e dropout, al fine di prevenire il problema dell'overfitting. L'architettura del modello è composta da più componenti: in primo luogo, sono stati utilizzati tre strati convoluzionali, con 128, 256 e 128 filtri rispettivamente, un kernel di dimensione 2 e una funzione di attivazione ReLU. Per stabilizzare la distribuzione degli input ai livelli successivi, sono stati inseriti strati di batch normalization. Inoltre, è stato applicato un dropout al 40% per migliorare la capacità del modello di generalizzare. Per ridurre la dimensionalità senza perdere informazioni cruciali, è stato utilizzato il *Global Average Pooling*. Infine, l'architettura prevede un livello denso finale con funzione di attivazione sigmoide, che consente la classificazione binaria.

4.3.3 Architettura e trasformazione con ROCKET

Il metodo *ROCKET* (Random Convolutional Kernel Transform) sfrutta un insieme di kernel convoluzionali casuali per estrarre caratteristiche significative dai dati. La

trasformazione segue una serie di passi principali: innanzitutto, vengono generati 500 kernel convoluzionali casuali con lunghezza variabile. Successivamente, questi kernel vengono applicati ai dati per calcolare il valore massimo della convoluzione e la proporzione di valori positivi risultanti. Ogni campione viene quindi rappresentato attraverso un vettore di feature, costruito in base ai kernel applicati. Infine, un classificatore Ridge viene addestrato per eseguire la classificazione finale dei dati.

4.4 Metodologia degli esperimenti

Per garantire la riproducibilità e l'affidabilità delle misurazioni, il sistema *Train Ticket* (descritto nella sezione 4.2.2) è stato ospitato all'interno di container Docker. Questa scelta ha permesso di isolare l'ambiente di esecuzione, minimizzando le interferenze esterne e garantendo la comparabilità dei risultati tra esperimenti successivi. L'intero processo sperimentale è stato automatizzato per assicurare un controllo rigoroso sulle variabili in gioco.

4.4.1 Configurazione dell'esperimento

L'esperimento è stato configurato seguendo un approccio sistematico per garantire la raccolta di dati coerenti e affidabili, riproducendo condizioni il più possibile vicine a un ambiente di produzione. Per questo motivo, è stato monitorato il consumo di CPU e memoria durante l'esecuzione dei test, in modo da evitare interferenze esterne che potessero compromettere la validità delle misurazioni. Questo controllo ha permesso di assicurare che le prestazioni osservate fossero effettivamente attribuibili al comportamento del sistema e non a fattori esterni. Per simulare un carico di lavoro realistico, è stato utilizzato Apache JMeter, configurato per inviare richieste agli endpoint del sistema con un incremento progressivo della frequenza fino al raggiungimento del carico massimo previsto. Questo approccio ha consentito di valutare la capacità del sistema di gestire variazioni graduali del traffico, replicando scenari tipici di utilizzo reale. La **durata dei test** è stata calibrata in modo da coprire l'intera transizione dal warm up allo steady state, come discusso nel capitolo 3. Tale scelta ha garantito che l'analisi delle prestazioni si basasse su dati significativi, evitando

distorsioni dovute a misurazioni effettuate in condizioni transitorie. Infine, per aumentare la robustezza della valutazione e ridurre il rischio di overfitting nei modelli di analisi, sono stati introdotti processi di disturbo che simulano condizioni operative reali. Questo ha permesso di testare il comportamento del sistema in presenza di variazioni impreviste del carico, migliorando l'affidabilità delle conclusioni tratte dall'esperimento.

4.4.2 Processo di raccolta dati

Durante gli esperimenti, sono stati raccolti dati relativi alle metriche di latenza, throughput e utilizzo delle risorse, al fine di analizzare le prestazioni del sistema. La metodologia adottata per la raccolta dei dati è descritta nel capitolo 3, dove vengono illustrate le fasi di pulizia, pre-elaborazione e segmentazione delle serie temporali.

Le metriche di performance sono state estratte in tempo reale e archiviate in un formato strutturato, garantendo un'organizzazione chiara dei dati. La segmentazione è stata realizzata applicando la metodologia descritta nella sezione 3.2.2, utilizzando la tecnica PELT (*Pruned Exact Linear Time*) per individuare il punto di stabilizzazione della serie temporale. Una volta raccolti, i dati sono stati elaborati per essere utilizzati nell'addestramento e nella valutazione dei modelli di Machine Learning, con l'obiettivo di analizzare il comportamento del sistema nelle fasi di warm up e di *steady state*. Per garantire un'analisi approfondita, sono stati testati vari endpoint del sistema *Train Ticket*, ciascuno dei quali rappresenta un microservizio con specifiche funzionalità (descritte nella tabella 4.1). Le richieste sono state progettate per simulare scenari realistici, considerando diverse condizioni di carico per valutare la scalabilità del sistema, la variazione dei parametri di input per testare la robustezza degli endpoint e l'introduzione di perturbazioni controllate per verificare la stabilità delle prestazioni, soprattutto per evitare l'overfitting dei modelli e garantire una generalizzazione efficace.

Tabella 4.1: Descrizione dei task testati sugli endpoint di *Train Ticket*.

Nome Task	**Descrizione**
Prenotazione di biglietti ferroviari	Simulazione di carichi generati da utenti che effettuano richieste di prenotazione in tempo reale.
Gestione delle transazioni	Interazioni con il database per confermare o annullare prenotazioni.
Ricerca e disponibilità	Richieste per la verifica della disponibilità di biglietti su tratte specifiche.
Generazione CAPTCHA	Richieste inviate al microservizio responsabile della generazione di codici CAPTCHA per la sicurezza nelle operazioni di login.

4.5 Metriche per la valutazione dei risultati

In questa sezione vengono definite le metriche utilizzate per valutare le prestazioni dei modelli di machine learning (ROCKET e FCN) e del framework proposto nel suo complesso. Tali metriche sono progettate per analizzare sia l'efficacia dei modelli nella classificazione dei blocchi stabili e instabili, sia l'affidabilità del framework nel rilevare correttamente lo *steady state* durante il *performance testing* [16].

4.5.1 Metriche per i modelli di classificazione

Le seguenti metriche, ampiamente utilizzate nella letteratura per la valutazione delle prestazioni dei modelli di classificazione [16], sono state adottate per analizzare l'accuratezza e l'affidabilità dei modelli **ROCKET** e **FCN**.

Per una maggiore chiarezza nella presentazione delle formule, utilizziamo le seguenti abbreviazioni: **TP** (True Positives) per indicare i veri positivi, **FP** (False Positives) per i falsi positivi, **TN** (True Negatives) per i veri negativi e **FN** (False Negatives) per i falsi negativi.

- **Precision:** Misura la proporzione di predizioni positive corrette rispetto al totale

delle predizioni positive effettuate dal modello. Rappresenta la capacità del modello di evitare falsi positivi.

$$\text{Precision} = \frac{TP}{TP + FP}$$

- **Recall:** Indica la proporzione di veri positivi correttamente identificati rispetto al totale dei positivi effettivi nel dataset. Fornisce un'indicazione sulla capacità del modello di identificare correttamente tutti i blocchi stabili.

$$\text{Recall} = \frac{TP}{TP + FN}$$

- **F1-score:** È la media armonica tra precision e recall, bilanciando entrambe le metriche in un unico valore. È particolarmente utile per valutare modelli in presenza di dataset sbilanciati.

$$\text{F1-score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

- **Accuratezza:** È la proporzione di predizioni corrette rispetto al totale delle osservazioni nel dataset, considerando sia la classe stabile che quella instabile.

$$\text{Accuratezza} = \frac{TP + TN}{TP + TN + FP + FN}$$

Queste metriche forniscono un quadro completo delle prestazioni dei modelli, evidenziando sia la loro capacità di distinguere correttamente le classi, sia la loro robustezza in scenari complessi.

4.5.2 Metriche per il framework

Abbiamo utilizzato le metriche citate in "AI-Driven Java Performance Testing" [1], le quali sono state impiegate per analizzare l'efficacia e l'efficienza del framework nel rilevare lo *steady state* e nell'ottimizzare il processo di *performance testing*:

- **Errore di Stima del warm up (warm up Estimation Error - WEE):** Misura la precisione del framework nel determinare il termine della fase di warm up. È calcolato come la differenza assoluta, in secondi, tra il vero momento di ingresso nello *steady state* (s_t) e il momento stimato dal framework ($\hat{s_t}$).

$$\text{WEE} = |s_t - \hat{s_t}|$$

- **Deviazione della Misurazione:** Valuta la qualità delle misurazioni raccolte dal framework durante lo *steady state*, confrontandole con le misurazioni raccolte manualmente a partire dal vero *steady state*. La deviazione è misurata utilizzando intervalli di confidenza per il rapporto tra le misurazioni del framework (M) e quelle di riferimento (M^*).

$$\text{Ratio} = \frac{M}{M^*}$$

Le metriche sopra descritte sono state scelte per fornire una valutazione approfondita dell'efficacia del framework nel migliorare il *performance testing*, sia in termini di accuratezza dei risultati che di ottimizzazione delle risorse e del tempo impiegato.

CAPITOLO 5

Risultati

In questo capitolo vengono presentati i risultati sperimentali ottenuti dai modelli di classificazione analizzati, con particolare attenzione alle prestazioni delle soluzioni basate su **FCN e ROCKET**. Viene esaminata l'efficacia del framework sviluppato per la gestione automatizzata della fase di warm up nei test di performance, mettendo in luce i suoi vantaggi rispetto agli approcci tradizionali.

Si analizzano le capacità dei modelli nel distinguere tra warm up e steady state, valutando il loro comportamento su diversi scenari di test. Viene approfondito il confronto tra i metodi proposti e le tecniche statistiche ed euristiche comunemente impiegate, evidenziandone le differenze in termini di precisione e affidabilità.

Infine, si discutono le implicazioni pratiche dell'adozione del machine learning nel performance testing dei microservizi, evidenziando i benefici di un approccio automatizzato nella riduzione dei tempi di testing e nell'ottimizzazione delle risorse.

5.1 RQ1: Quanto sono efficaci i modelli di Time Series Classification nel classificare accuratamente le misurazioni di performance, distinguendo tra la fase di warm up e lo steady state?

Per valutare le prestazioni delle soluzioni ottenute, sono stati analizzati i risultati prodotti dai modelli **Fully Convolutional Network (FCN)** e **RandOm Convolutional KErnel Transform (ROCKET)**. L'analisi è stata condotta utilizzando le metriche definite nella sezione 4.5.1. Nella tabella 5.1 sono riportati i risultati ottenuti da entrambi i modelli, suddivisi per ciascuna classe analizzata.

Tabella 5.1: Risultati ottenuti dai modelli FCN e ROCKET

Modello	Classe	Veri Positivi	Falsi Positivi	Falsi Negativi	Veri Negativi
FCN	Unstable	492	148	20	3605
FCN	Stable	3605	20	148	492
ROCKET	Unstable	3720	135	17	3634
ROCKET	Stable	3634	17	135	3720

Per comprendere in modo più chiaro l'andamento delle predizioni effettuate dai modelli, nelle figure 5.1 e 5.2 vengono riportati rispettivamente il report di classificazione e la matrice di confusione per FCN e ROCKET.

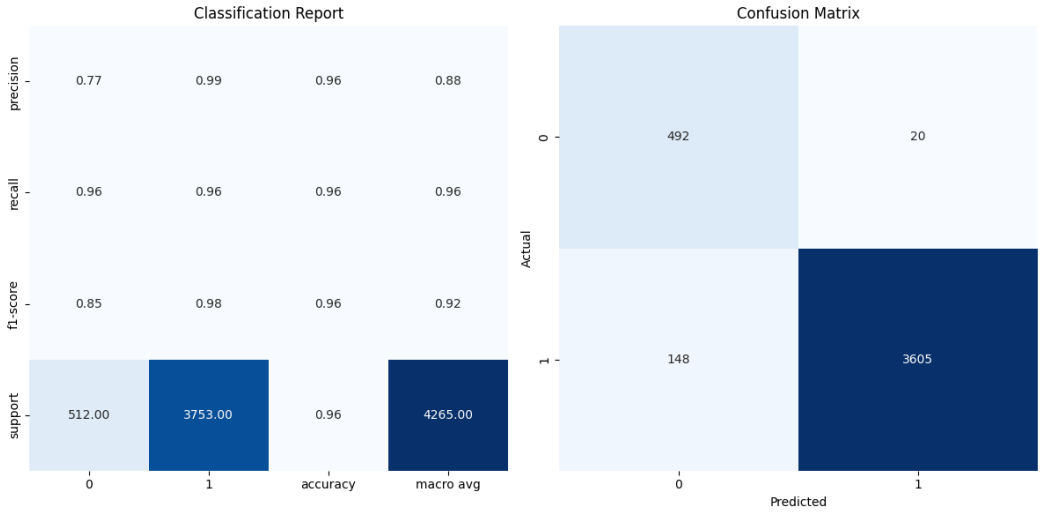

Figura 5.1: Report di classificazione e matrice di confusione per il modello FCN

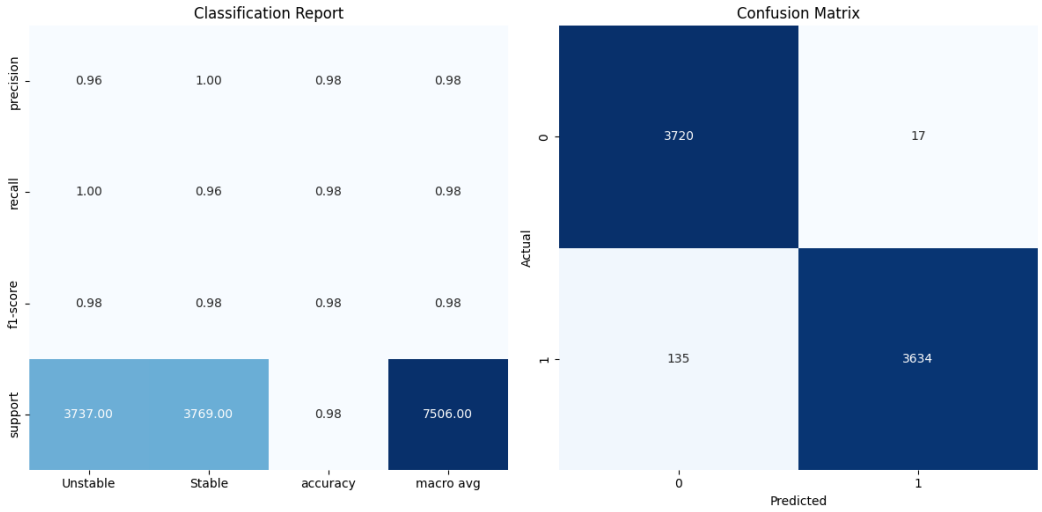

Figura 5.2: Report di classificazione e matrice di confusione per il modello ROCKET

Analizzando i risultati riportati in tabella 5.1, emergono diverse considerazioni riguardanti l'efficacia dei due approcci. Il modello **ROCKET** raggiunge una precisione del 96,5% per la classe *Unstable* e del 99,5% per la classe *Stable*. Il recall per la classe *Unstable* è del 99,5%, mentre per la classe *Stable* è del 96,4%. L'F1-score per la classe *Unstable* si attesta al 98,0%, mentre per la classe *Stable* è pari al 97,9%. Infine, l'accuratezza complessiva del modello è del 98,0%. D'altra parte, il modello **FCN** mostra una precisione del 76,9% per la classe *Unstable* e del 99,4% per la classe *Stable*. Il valore di recall per la classe *Unstable* è del 96,1%, mentre per la classe *Stable* è sempre del 96,1%. Considerando la combinazione tra precision e recall, il valore di

F1-score per la classe *Unstable* è del 85,7% e per la classe *Stable* raggiunge il 97,7%. L'accuratezza complessiva del modello è del 96,1%.

Un aspetto cruciale nella valutazione dei modelli è la loro capacità di individuare il punto di **steady state**. Il modello FCN si distingue per un comportamento più conservativo, il che implica una maggiore prudenza nell'identificazione del passaggio alla stabilità. Questo si traduce in una minore probabilità di falsi positivi, ma può comportare un ritardo nel riconoscere la stabilizzazione effettiva del sistema. Di conseguenza, il modello FCN potrebbe risultare meno adatto a scenari in cui è necessario rilevare rapidamente la stabilità. D'altro canto, ROCKET mostra una tendenza più aggressiva nella classificazione, riuscendo a individuare il punto di steady state in tempi più brevi rispetto a FCN. La maggiore sensibilità di ROCKET riduce il numero di falsi negativi, permettendo una reattività superiore nel riconoscere quando il sistema ha raggiunto una condizione stabile. Tuttavia, questa maggiore sensibilità si accompagna a un numero leggermente superiore di falsi positivi, il che potrebbe portare a segnalazioni premature di stabilità in alcune condizioni limite.

I risultati empirici dimostrano che i modelli di Time Series Classification (TSC) sono altamente efficaci nel distinguere tra warm up e steady state. La precisione dei modelli FCN e ROCKET è risultata superiore rispetto ai metodi manuali e statistici, con una riduzione significativa degli errori di sovrastima e sottostima del warm up. Inoltre, la capacità di generalizzazione del framework ha permesso di ottenere buone prestazioni anche su endpoint non inclusi nel dataset di addestramento. L'analisi dettagliata dei risultati mostra che entrambi i modelli hanno raggiunto alte performance con un numero ridotto di errori. Il confronto tra i modelli evidenzia che FCN ha leggermente superato ROCKET in termini di recall sulla classe Stable, mentre ROCKET ha mantenuto un'elevata precisione complessiva. Ciò suggerisce che FCN è più efficace nell'identificare i casi positivi, mentre ROCKET è più accurato nel limitare i falsi positivi. Tuttavia, FCN mostra una tendenza leggermente maggiore a perdere alcuni casi positivi (falsi negativi), mentre ROCKET ha una lieve predisposizione a identificare erroneamente alcuni negativi come positivi (falsi positivi).

La scelta tra i due modelli dipende quindi dal contesto applicativo: se si desidera un modello più prudente e selettivo, ROCKET potrebbe essere preferibile; se invece è essenziale catturare il maggior numero possibile di istanze positive, FCN potrebbe

rappresentare la soluzione più adeguata. In entrambi i casi, i modelli hanno dimostrato un'accuratezza media superiore al 90%, con un basso tasso di falsi positivi nel rilevamento dello steady state.

Risultato RQ1: I modelli di TSC sono altamente efficaci nel classificare le misurazioni, garantendo alta precisione nella distinzione tra warm-up e steady state, con un'accuratezza media superiore al 90%.

5.2 RQ2: Il framework di TSC è in grado di migliorare lo stato dell'arte rispetto ai metodi tradizionali per il rilevamento del warm up?

Per verificare l'efficacia del framework, sono stati testati endpoint diversi rispetto a quelli utilizzati durante la fase di addestramento. Le misurazioni sono state eseguite contattando gli endpoint in locale sulla stessa macchina in cui *Train Ticket* è stato deployato via Docker. Questo ha garantito un ambiente di test controllato, minimizzando le interferenze dovute alla latenza di rete e fornendo una valutazione accurata delle prestazioni del framework.

Nella Tabella 5.2 sono riportati i tempi impiegati dagli endpoint per raggiungere lo steady state, espressi in secondi, per ciascun metodo di stima utilizzato. In particolare, i modelli di machine learning, **FCN** e **ROCKET**, forniscono una stima del tempo basata sull'analisi automatica dei dati. Inoltre, la tabella include i tempi calcolati mediante un approccio **euristico** e un metodo **statistico** (metodologie citate nella sezione 2.3). Infine, come termine di confronto, è riportato il valore **manuale**, che rappresenta il tempo effettivamente misurato senza l'ausilio di modelli predittivi. I risultati delle metriche derivate, come l'Errore di Stima del warm-up (WEE), sono invece presentati nella Tabella 5.3.

Tabella 5.2: Misurazioni del framework su endpoint non presenti nei dati di addestramento. Tutti i tempi sono espressi in secondi e le colonne indicano i diversi tipi di approccio.

Endpoint contattato	FCN	ROCKET	Euristico	Statistico	Manuale
API OrderOther Refresh	6	8	15	21	11
API Order Refresh	21	24	37	28	20
API Verify Code Generate	5	5	14	19	7
API Consign Service Account	11	10	18	12	12
API AdminBasic Stations	18	19	20	22	18
API AdminBasic Contacts	6	5	14	11	4
API AdminBasic Trains	9	8	16	17	10
API AdminBasic Configs	6	6	12	11	6

Tabella 5.3: Metriche di errore di stima del framework (WEE)

Endpoint contattato	FCN	ROCKET	Euristico	Statistico
API OrderOther Refresh	5	3	4	10
API Order Refresh	1	4	17	8
API Verify Code Generate	2	2	7	12
API Consign Service Account	1	2	6	0
API AdminBasic Stations	0	1	2	4
API AdminBasic Contacts	2	1	10	7
API AdminBasic Trains	1	2	6	7
API AdminBasic Configs	0	0	6	5

Le metriche utilizzate sono state approfondite nella sezione 4.5.1. L'analisi dei risultati in tabella 5.3 mostra che i modelli **FCN** e **ROCKET** hanno ottenuto stime accurate del tempo di **warm up**, con errori contenuti rispetto ai valori reali. In particolare, per l'endpoint *API OrderOther Refresh*, il modello **FCN** ha registrato un errore di stima del **warm up** (**WEE**) di **5 secondi**, mentre **ROCKET** ha ottenuto un

valore di **3 secondi**. Gli approcci **euristico** e **statistico**, invece, hanno mostrato errori più elevati, pari rispettivamente a **4** e **10 secondi**.

Per quanto riguarda la **deviazione della misurazione**, si osserva che per *API Verify Code Generate* entrambi i modelli di **machine learning** hanno ottenuto un **Ratio** di **0.71**, mentre gli approcci **euristico** e **statistico** hanno presentato una sovrastima con valori di **2.00** e **2.71**.

L'analisi evidenzia che entrambi i modelli forniscono stime molto vicine ai valori reali, anche per endpoint non osservati durante la fase di addestramento. Tuttavia, emergono alcune differenze significative tra i due approcci. Il modello **FCN** tende a produrre stime più prudenti, mantenendosi generalmente più vicino ai tempi misurati manualmente. Al contrario, in alcuni casi, **ROCKET** ha restituito previsioni leggermente più basse rispetto al tempo reale. Inoltre, l'uso degli approcci **euristici** e **statistici** può rallentare il processo di testing, poiché le loro stime tendono a sovrastimare il tempo necessario per raggiungere lo **steady state**. Questo può portare a un utilizzo inefficiente delle **risorse di testing** e a una maggiore durata complessiva delle analisi, riducendo così l'efficienza complessiva del processo.

I risultati della tabella 5.3 dimostrano la capacità del **framework** di generalizzare bene su endpoint nuovi, garantendo un'accurata individuazione dello **steady state** anche in scenari non visti in fase di training. Inoltre, l'efficacia del **framework** è stata dimostrata nel ridurre il **tempo complessivo di testing**, consentendo un'analisi più rapida delle prestazioni del sistema rispetto all'approccio manuale. La possibilità di integrare il **framework** con strumenti di **performance testing automatizzati** potrebbe migliorare ulteriormente l'efficienza del processo di testing, riducendo il costo computazionale e il tempo richiesto per determinare lo **steady state** di un sistema.

L'analisi comparativa tra il framework di Time Series Classification (TSC) e i metodi tradizionali ha evidenziato un miglioramento significativo nella capacità di rilevare correttamente la fase di warm up. Dai risultati presentati in 5.2, emerge che i modelli basati su TSC mostrano una maggiore capacità di adattamento a diversi pattern di carico, riducendo l'errore di stima rispetto agli approcci euristici e statistici. Inoltre, l'approccio basato su deep learning permette di riconoscere schemi complessi non facilmente modellabili con metodi tradizionali.

I risultati mostrano che il framework TSC riduce la variabilità dei tempi di warm

up stimati, con una minore dispersione delle previsioni rispetto ai metodi statistici. Questo porta a una maggiore affidabilità nell'individuazione dello steady state, permettendo di ottimizzare il processo di performance testing.

Risultato RQ2: Il framework di TSC migliora lo stato dell'arte grazie a una maggiore capacità di adattamento ai dati, riducendo l'errore di stima rispetto ai metodi tradizionali e fornendo stime più stabili.

CAPITOLO 6

Minacce alla Validità

In questo capitolo vengono analizzate le potenziali minacce alla validità dei risultati dello studio, focalizzandoci sugli aspetti critici dell'esperimento condotto per la determinazione automatica del periodo di warm-up nel performance testing di microservizi. L'obiettivo è individuare le condizioni in cui le risposte alle domande di ricerca potrebbero risultare imprecise o non del tutto affidabili.

6.1 Validità di Costrutto

Una possibile minaccia alla validità di costrutto riguarda la scelta della latenza come metrica principale per determinare il passaggio dal warm-up allo steady-state. Sebbene la latenza sia un indicatore rilevante, altre metriche come il throughput o l'utilizzo delle risorse computazionali potrebbero fornire una visione più completa del comportamento del sistema. Studi precedenti [17] hanno mostrato che metriche multiple, anziché una sola, possono migliorare la robustezza dei modelli di rilevazione dello steady-state.

Se il metodo di rilevazione dello steady-state si basa unicamente sulla latenza, potrebbe non essere in grado di cogliere tutti gli aspetti della stabilizzazione del sistema, portando a una sovrastima o sottostima del tempo necessario al warm-up. Inoltre, la variabilità delle misurazioni potrebbe influenzare il modello predittivo,

introducendo un margine di errore non trascurabile nelle risposte alla domanda di ricerca RQ1, che indaga l'accuratezza dei modelli di classificazione nel distinguere le fasi del sistema. Un'ulteriore minaccia riguarda l'adozione di finestre temporali fisse per l'analisi delle metriche. Diversi microservizi possono avere tempi di warm-up significativamente differenti e un approccio statico potrebbe non essere ottimale per tutti i casi [11]. La scelta di segmentare la serie temporale in blocchi sovrapposti di dimensione fissa pari a 50 misurazioni è motivata da diversi fattori. In primo luogo, tale dimensione consente di mantenere un buon compromesso tra granularità e stabilità della classificazione delle misurazioni. Inoltre, una finestra di 50 misurazioni permette una classificazione più stabile e meno sensibile al rumore. Infine, la scelta di utilizzare blocchi sovrapposti garantisce una transizione più fluida tra le diverse fasi del sistema, migliorando la capacità del modello di adattarsi in tempo reale alle variazioni nelle prestazioni dei microservizi.

6.2 Validità Interna

L'affidabilità dei risultati potrebbe essere influenzata da fattori esterni non considerati nell'esperimento. Una delle principali minacce riguarda la variabilità dell'ambiente di test. Differenze nell'infrastruttura cloud, nelle politiche di allocazione delle risorse o nella concorrenza tra processi potrebbero influenzare la durata effettiva del warm-up e portare a stime diverse. L'addestramento dei modelli di machine learning su un dataset limitato rappresenta un'ulteriore minaccia. Se il dataset non fosse sufficientemente diversificato, il modello potrebbe non essere in grado di generalizzare correttamente su nuovi contesti, compromettendo la validità interna dei risultati. Per mitigare questo rischio, sarebbe necessario ampliare la raccolta dati includendo scenari più vari e realistici [1].

Un altro possibile fattore limitante dell'esperimento è l'assenza di politiche di autoscaling, che potrebbe influenzare significativamente il comportamento del warm-up. In un'infrastruttura dinamica, la variazione automatica delle risorse allocate potrebbe modificare la fase di stabilizzazione del sistema rispetto a quanto osservato nei test condotti in ambiente statico. Questo aspetto potrebbe introdurre una fonte di variabilità nei risultati, incidendo sulla capacità del metodo proposto di

identificare con precisione il passaggio allo steady state. La mancanza di autoscaling potrebbe quindi influenzare la generalizzabilità dei risultati all'interno di contesti di produzione reali, dove le risorse sono gestite dinamicamente [18].

6.3 Validità Esterna

Un potenziale limite dell'approccio adottato riguarda la sua applicabilità a diversi ambienti e stack tecnologici. Le ottimizzazioni e la gestione della memoria nei diversi ambienti di esecuzione potrebbero alterare i tempi di stabilizzazione, portando il metodo proposto a fornire risultati diversi rispetto a quelli ottenuti nell'esperimento. I test sono stati condotti su un ambiente containerizzato con un numero limitato di microservizi, mentre in scenari più complessi, caratterizzati da migliaia di servizi in esecuzione simultanea, potrebbero emergere fenomeni di interdipendenza che influenzano la stabilizzazione del sistema.

Il benchmark utilizzato, *Train Ticket*, segue pattern di carico specifici che potrebbero non essere rappresentativi di tutte le tipologie di microservizi esistenti. Se il metodo fosse applicato a sistemi con carichi di lavoro più imprevedibili o caratterizzati da picchi improvvisi, i risultati potrebbero variare, rendendo difficile trarre conclusioni generalizzabili. Alcuni studi [19] hanno analizzato le differenze tra workload sintetici e reali, evidenziando la necessità di dataset più variegati per la valutazione di modelli di machine learning nel performance testing. Abbiamo deciso di fermarci all'uso del benchmark *Train Ticket* poiché esso rappresenta un ottimo stato dell'arte per le architetture a microservizi, offrendo una struttura realistica e complessa, con una suddivisione in numerosi microservizi e interazioni realistiche tra essi. Inoltre, *Train Ticket* fornisce un'ampia gamma di API, il che lo rende particolarmente utile per l'addestramento di modelli di machine learning volti all'analisi delle prestazioni. Questi aspetti lo rendono una scelta solida per la valutazione dell'approccio proposto, garantendo un buon bilanciamento tra realismo e controllabilità sperimentale.

6.4 Validità della Conclusione

La principale minaccia alla validità delle conclusioni è l'assenza di test statistici sui risultati ottenuti. L'accuratezza delle conclusioni dipende infatti dalla robustezza statistica dei dati analizzati e, senza un'adeguata verifica formale, non è possibile determinare se le differenze osservate siano statisticamente significative o dovute alla naturale variabilità dei dati raccolti. Anche se è stato introdotto un indicatore di errore come il *Warm-up Estimation Error* (WEE) per quantificare la precisione del metodo, questo da solo non è sufficiente a garantire una validazione rigorosa dei risultati. Sarebbe necessario condurre test statistici appropriati, come test di significatività o intervalli di confidenza, per rafforzare la validità delle conclusioni tratte. Inoltre, un ulteriore aspetto critico è la necessità di ampliare il numero di scenari testati per valutare la capacità predittiva del modello in condizioni più diversificate. Se il modello di machine learning avesse una capacità predittiva inferiore rispetto a quanto atteso, le risposte alle domande di ricerca potrebbero risultare meno affidabili, rendendo necessaria una revisione delle ipotesi iniziali e una riconsiderazione del framework [1].Pertanto, per migliorare la solidità delle conclusioni, sarebbe opportuno includere un'analisi statistica più approfondita e ampliare il dataset sperimentale, in modo da ridurre il rischio di overfitting e garantire risultati più generalizzabili.

CAPITOLO 7

Conclusioni

In questa tesi è stato affrontato il problema della **determinazione del periodo di warm-up** ottimale per il testing delle prestazioni delle applicazioni basate su **microservizi nel cloud** è cruciale per garantire risultati affidabili e riproducibili, eliminando la variabilità introdotta dai processi di inizializzazione dei microservizi e dalle dinamiche dell'ambiente cloud.

Il contributo principale di questo lavoro è lo sviluppo di un **framework basato su tecniche di Time Series Classification (TSC)** per l'identificazione automatizzata del termine della fase di warm-up. L'approccio adottato sfrutta modelli avanzati di machine learning, ossia **FCN e ROCKET**, per classificare accuratamente le misurazioni di performance e distinguere tra warm-up e steady state. I risultati hanno dimostrato che il framework migliora significativamente lo stato dell'arte rispetto ai metodi tradizionali, offrendo maggiore precisione nell'individuazione del warm-up e riducendo il tempo complessivo di testing.

L'analisi comparativa tra il framework TSC e i metodi euristici e statistici ha evidenziato **una maggiore affidabilità e adattabilità del framework**, capace di adattarsi dinamicamente ai pattern di carico senza necessità di intervento manuale. Inoltre, l'inclusione di processi di disturbo nelle misurazioni ha confermato la capacità del framework di gestire scenari realistici, riducendo il rischio di overfitting e migliorando

la generalizzazione delle previsioni.

I risultati sperimentali hanno mostrato che l'errore massimo osservato per FCN è stato di 7 secondi, mentre per ROCKET di 5 secondi, rispetto alle misurazioni manuali, suggerendo che ROCKET offre stime leggermente più precise, sebbene FCN fornisca previsioni più conservative. Inoltre, l'uso del framework ha portato a una riduzione significativa del tempo di testing rispetto ai metodi manuali, contribuendo a un'ottimizzazione del processo di benchmarking.

L'applicazione di tecniche di intelligenza artificiale nel performance testing apre nuove prospettive per l'ottimizzazione dei processi di valutazione delle prestazioni dei microservizi. Il framework sviluppato può essere ulteriormente migliorato integrando nuovi modelli di classificazione e strategie di adattamento automatico ai workload variabili. In futuro, potrebbe essere utile esplorare l'uso di modelli transformer per l'analisi delle serie temporali e valutare l'integrazione del framework con strumenti di performance testing automatizzati. Inoltre, un'analisi più approfondita dell'impatto delle caratteristiche architetturali dei microservizi sulle prestazioni del framework potrebbe fornire ulteriori spunti per il miglioramento del modello.

L'efficacia del framework sviluppato dimostra che l'adozione di metodi basati su machine learning rappresenta un passo avanti significativo rispetto agli approcci tradizionali. Grazie alla capacità di adattarsi dinamicamente ai pattern di carico e alla riduzione del tempo complessivo di testing, il framework proposto si configura come una soluzione innovativa e applicabile a diversi scenari di performance testing, migliorando l'efficienza e l'affidabilità della valutazione delle prestazioni nei sistemi basati su microservizi.

Bibliografia

[1] L. Traini, F. Di Menna, and V. Cortellessa, "Ai-driven java performance testing: Balancing result quality with testing time," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. ACM, Oct. 2024, p. 443–454. [Online]. Available: http://dx.doi.org/10.1145/3691620.3695017 (Citato alle pagine iv, 11, 18, 19, 28, 40, 51 e 53)

[2] Hechuan73, "Train ticket - a benchmark for microservices performance testing." [Online]. Available: https://github.com/hechuan73/train_ticket (Citato alle pagine iv e 34)

[3] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*, 2015, pp. 27–34. [Online]. Available: https://ieeexplore.ieee.org/document/7371699 (Citato a pagina 7)

[4] P. Mahajan and R. Ingalls, "Evaluation of methods used to detect warm-up period in steady state simulation," in *Proceedings of the 2004 Winter Simulation Conference, 2004.*, vol. 1, 2004, p. 671. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/1371374 (Citato a pagina 9)

[5] C. Delimitrou *et al.*, "An open-source benchmark suite for microservices and their design trade-offs," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 3–18. [Online]. Available: https://dl.acm.org/doi/10.1145/3297858.3304013 (Citato a pagina 11)

[6] I. Papapanagiotou and V. Chella, "Ndbench: Benchmarking microservices at scale," *arXiv preprint arXiv:1807.10792*, 2018. [Online]. Available: https://arxiv.org/abs/1807.10792 (Citato a pagina 11)

[7] J. Wang and J. Wu, "Research on performance automation testing technology based on jmeter," in *2019 International Conference on Robots Intelligent System (ICRIS)*, 2019, pp. 55–58. [Online]. Available: https://ieeexplore.ieee.org/document/8806309 (Citato a pagina 11)

[8] Apache Software Foundation, *Apache JMeter Component Reference*. [Online]. Available: https://jmeter.apache.org/usermanual/component_reference.html (Citato a pagina 13)

[9] P. Mahajan and R. Ingalls, "Evaluation of methods used to detect warm-up period in steady state simulation," 01 2005, pp. – 671. [Online]. Available: https://www.researchgate.net/publication/4111771_Evaluation_of_Methods_Used_to_Detect_Warm-Up_Period_in_Steady_State_Simulation (Citato a pagina 14)

[10] C. Laaber, J. Klinglmayr, and P. Leitner, "Predicting unstable software benchmarks using static source code features," *Empirical Software Engineering*, 2021. [Online]. Available: https://link.springer.com/article/10.1007/s10664-021-09996-y (Citato alle pagine 15 e 17)

[11] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically rigorous java performance evaluation," *OOPSLA*, pp. 57–76, 2007. [Online]. Available: https://doi.org/10.1145/1297027.1297033 (Citato alle pagine 15, 16 e 51)

[12] H. I. Fawaz, G. Forestier, J. Weber, L. Idoumghar, and P.-A. Muller, "Deep learning for time series classification: a review," *Data Mining and*

Knowledge Discovery, vol. 33, no. 4, pp. 917–963, 2019. [Online]. Available: https://doi.org/10.1007/s10618-019-00619-1 (Citato a pagina 18)

[13] A. Dempster, F. Petitjean, and G. I. Webb, "Rocket: Exceptionally fast and accurate time series classification using random convolutional kernels," *Data Mining and Knowledge Discovery*, 2020. [Online]. Available: https://link.springer.com/article/10.1007/s10618-020-00701-z (Citato a pagina 19)

[14] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," *arXiv:1611.06455*, pp. 1299–1305, 2017. [Online]. Available: https://arxiv.org/abs/1611.06455 (Citato a pagina 22)

[15] H. A. Dau, E. Keogh, K. Kamgar, C.-C. M. Yeh, Y. Zhu, S. Gharghabi, C. A. Ratanamahatana, B. Hu, N. Begum, A. Bagnall *et al.*, "The ucr time series archive," *IEEE/ACM Transactions on Knowledge Discovery from Data*, vol. 11, no. 3, pp. 332–336, 2019. [Online]. Available: https://arxiv.org/abs/1810.07758 (Citato a pagina 22)

[16] J. Opitz, "A closer look at classification evaluation metrics and a critical reflection of common evaluation practice," *Transactions of the Association for Computational Linguistics*, vol. 12, p. 820–836, 2024. [Online]. Available: http://dx.doi.org/10.1162/tacl_a_00675 (Citato a pagina 39)

[17] C. Laaber and P. Leitner, "An evaluation of open-source software microbenchmark suites for continuous performance assessment," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, 2018, pp. 119–130. [Online]. Available: https://ieeexplore.ieee.org/document/8595195 (Citato a pagina 50)

[18] M. Shahrad and D. Wentzlaff, "Availability knob: Flexible user-defined availability in the cloud," ser. SoCC '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 42–56. [Online]. Available: https://doi.org/10.1145/2987550.2987556 (Citato a pagina 52)

[19] Y. Gan and C. Delimitrou, "The architectural implications of cloud microservices," *IEEE Computer Architecture Letters*, vol. PP, pp. 1–1, 05 2018.

[Online]. Available: https://www.researchgate.net/publication/325307768_The_Architectural_Implications_of_Cloud_Microservices (Citato a pagina 52)

Ringraziamenti

Concludere questo percorso di studi rappresenta un traguardo importante nella mia vita, e non avrei potuto raggiungerlo senza il sostegno di molte persone che mi sono state vicine in vari momenti di questo lungo viaggio.

Innanzitutto, desidero esprimere la mia più sincera gratitudine al mio relatore, il **Prof. Dario Di Nucci**, per la sua guida preziosa, la pazienza e l'incoraggiamento costante. Un ringraziamento speciale va anche al mio co-relatore, il **Dott. Antonio Trovato**, per la sua disponibilità, i suoi suggerimenti costruttivi e il suo sostegno concreto, che sono stati fondamentali per il completamento di questa tesi. Un sentito grazie anche a tutti i docenti e i collaboratori dell'**Università degli Studi di Salerno (UNISA)** che, con la loro professionalità e disponibilità, hanno arricchito il mio percorso formativo. Un ringraziamento affettuoso va alla mia famiglia, che mi ha sempre supportato con amore, comprensione e fiducia incondizionata. La loro presenza è stata la base solida su cui ho potuto costruire ogni passo di questa esperienza. Infine, desidero ringraziare i miei amici per il loro sostegno morale, per avermi fatto sentire sempre meno solo e per avermi fatto sorridere anche nei momenti più difficili. A tutte queste persone va il mio più sincero grazie, perché senza di loro questo traguardo non sarebbe stato possibile.