POLITECNICO DI TORINO

Laurea Magistrale in Ingegneria Informatica

Tesi di Laurea Magistrale

Prototipazione di una piattaforma di osservabilità per datacenter Kubernetes

Relatore

Prof. Fulvio RISSO

Candidato

Vito TASSIELLI

4 / 2021

Sommario

L'obiettivo del progetto di tesi è studiare ed analizzare l'importanza dell'osservabilità all'interno dei sistemi informatici moderni e sviluppare una piattaforma di osservabilità che monitori costantemente il sistema, mostrando le anomalie presenti, generalmente difficili da individuare per un operatore umano.

Per valutare le prestazioni della piattaforma sviluppata, si usa l'applicazione open source Online Boutique, progettata da Google e sviluppata con 11 microservizi scritti in linguaggi di programmazione diversi. L'applicazione simula il sito di un ecommerce, consentendo la navigazione del catalogo e l'acquisto degli oggetti in vendita.

Nella prima fase del progetto si studia l'osservabilità e la sua importanza nei sistemi informatici moderni. Implementare l'osservabilità all'interno di un sistema migliora la fase di monitoraggio poiché, oltre a studiare il sistema dall'esterno, è possibile conoscere precisamente il suo stato interno.

La letteratura informatica definisce tre pilastri importanti dell'osservabilità: *logs*, per registrare gli eventi avvenuti all'interno del sistema, *metriche*, per misurare dati specifici del sistema in un intervallo di tempo, e *tracciature*, per rappresentare graficamente il percorso delle richieste gestite dal sistema distribuito.

Si prosegue con il disegno della strategia per lo sviluppo della piattaforma di osservabilità all'interno di un cluster Kubernetes, così da monitorare ad alto livello l'applicazione Online Boutique.

La strategia è composta da quattro fasi distinte:

- *generazione*: si rendono osservabili i microservizi di Online Boutique. In questa fase si generano direttamente dal codice sorgente dei microservizi i logs, le metriche e le tracciature.
- *acquisizione ed elaborazione*: installando strumenti professionali si acquisiscono e si elaborano i tre pilastri dell'osservabilità esposti dai microservizi.
- *memorizzazione*: si conservano logs, metriche e tracciature grazie all'implementazione di database specializzati.

- *visualizzazione*: utilizzando tutti i dati collezionati, si creano rappresentazioni grafiche personalizzate per riassumere lo stato interno di Online Boutique.

Disegnata la strategia di sviluppo, si individuano e si implementano i migliori strumenti open source offerti dalla comunità informatica, affinché la piattaforma di osservabilità funzioni correttamente. Pertanto, si mostrano le applicazioni implementate per il monitoraggio dei tre pilastri dell'osservabilità.

Per il **monitoraggio dei logs** si installano e si configurano:

- le *librerie standard* dei linguaggi di programmazioni che generano i logs;
- *FluentBit* e *FluentD* per la fase di acquisizione ed elaborazione;
- *Loki* specializzato nella memorizzazione dei logs;
- *Grafana* per visualizzare tutti i logs di Online Boutique, aggregati in un unico punto.

Per il **monitoraggio delle metriche** si usano:

- gli strumenti di *OpenTelemetry* che generano le metriche;
- *Prometheus* per acquisire, elaborare e memorizzare le metriche esposte dai microservizi;
- *Grafana* per creare dashboards personalizzate e dinamiche così da monitorare i microservizi.

Infine, per il **monitoraggio delle tracciature** si implementano:

- gli strumenti di *OpenTelemetry* che generano le tracciature;
- *Jaeger* per acquisire, elaborare e memorizzare le tracciature ricevute dai microservizi;
- *Grafana* per visualizzare le tracciature delle richieste elaborate da Online Boutique.

Terminato lo sviluppo della piattaforma di osservabilità, si arricchiscono le funzionalità fornite implementano nuove estensioni. Infatti, si sviluppa la generazione delle metriche dai logs di Online Boutique e si implementano strumenti per garantire l'alta affidabilità dei dati raccolti, *Thanos* ed *Elasticsearch*.

Infine, si evidenziano eventuali sviluppi futuri, non ancora offerti dalla comunità informatica, che possono migliorare il monitoraggio e lo sviluppo della piattaforma.

Si conclude il progetto di tesi verificando se siano state soddisfatte le esigenze iniziali e mostrando, pertanto, gli obbiettivi e i risultati raggiunti.

Indice

1 Introduzione 1
- 1.1 Scenario attuale . . . 1
- 1.2 Esigenza . . . 2
- 1.3 Obiettivi . . . 3
- 1.4 Risultati attesi . . . 3
- 1.5 Presentazione del progetto . . . 3
- 1.6 Struttura del documento . . . 4

2 Cenni teorici sui sistemi moderni e sull'osservabilità 7
- 2.1 Introduzione ai sistemi moderni . . . 7
  - 2.1.1 Cloud Computing . . . 10
  - 2.1.2 Microservizi . . . 12
- 2.2 Osservabilità . . . 19
  - 2.2.1 Importanza dell'osservabilità nei sistemi moderni . . . 19
  - 2.2.2 Differenza tra osservabilità e monitoraggio . . . 20
  - 2.2.3 Blackbox e whitebox monitoring . . . 20
  - 2.2.4 I tre pilastri dell'osservabilità . . . 21
  - 2.2.5 Alta affidabilità . . . 25

3 Sviluppo della piattaforma di osservabilità 28
- 3.1 Applicazione a microservizi Online Boutique . . . 29
- 3.2 Difficoltà del monitoraggio . . . 31
- 3.3 Analisi e disegno della strategia di monitoraggio . . . 31
  - 3.3.1 Generazione . . . 32
  - 3.3.2 Acquisizione ed elaborazione . . . 33
  - 3.3.3 Memorizzazione . . . 35
  - 3.3.4 Rappresentazione . . . 35
- 3.4 Strumenti per il monitoraggio del sistema . . . 36
  - 3.4.1 Monitoraggio dei logs . . . 37
  - 3.4.2 Monitoraggio delle metriche . . . 42
  - 3.4.3 Monitoraggio delle tracciature . . . 45

3.5 Piattaforma unica 48
3.5.1 Configurazione per i logs 50
3.5.2 Configurazione per le metriche 53
3.5.3 Configurazione per le tracciature 61
3.6 Estensioni enterprise della piattaforma 64
3.6.1 Alta affidabilità 65
3.6.2 Generazione delle metriche dai logs 67
3.6.3 Possibili estensioni 69
3.7 Confronti 71
3.7.1 Instrumentazione manuale realizzata e instrumentazione automatica di OpenTelemetry 71
3.7.2 Istio e livello di funzionalità raggiunto dalla piattaforma sviluppata 72
3.8 Risultati ottenuti 73
3.8.1 Rappresentazione grafica dello stato del sistema 73
3.8.2 Rappresentazione grafica della propagazione delle richieste nel sistema 76

4 Conclusioni 78
4.1 Raggiungimenti delle esigenze iniziali 78
4.2 Eventuali sviluppi futuri 78
4.2.1 Espansione delle funzionalità per colmare le differenze con Istio 79
4.2.2 Instrumentazione automatica di OpenTelemetry 79
4.2.3 Espansione dell'allarmistica 79

Bibliografia 81

Capitolo 1

Introduzione

1.1 Scenario attuale

Negli ultimi anni la scena tecnologica è stata sempre più conquistata dalla crescita di Internet, grazie sopratutto alle molte aziende che hanno visto in questo strumento una nuova fonte di guadagno su cui costruire innovativi modelli di business. Inoltre, la capacità di essere connessi con tutto il mondo in qualsiasi momento della giornata ha catturato l'attenzione di gran parte della popolazione mondiale. Sono nati così, sempre più servizi offerti tramite Internet e destinati ad un pubblico sempre più vasto. Un esempio di questi servizi sono le applicazioni di messaggistica online, i social network e le piattaforme di streaming video e musicale.

È nata quindi, da parte delle compagnie, la necessità di sviluppare all'interno dell'azienda stessa, una infrastruttura di rete all'avanguardia, in grado di gestire le richieste degli utenti finali ricevute tramite Internet. Per fare ciò, era necessario dedicare tempo e risorse allo sviluppo software ma sopratutto hardware, vista la necessità di possedere dei server per comunicare con gli utenti.

In questo contesto, sono nati nuovi paradigmi e modelli di business finalizzati a supportare le aziende nello sviluppo della loro infrastruttura di rete. Tra i diversi modelli proposti, quello che semplifica più di tutti questo processo, è il cloud computing.

Con il **cloud computing**, infatti, le piccole e grandi aziende non devono più sviluppare da zero una infrastruttura di rete, ma attraverso Internet richiedono e utilizzano i servizi di una compagnia specializzata in questa nuova tecnologia. Questo approccio permette alle aziende di gestire in maniera più accurata i propri fondi, investendo solo lo stretto necessario per erogare i propri servizi. Non si incorre nel rischio di sviluppare infrastrutture di rete o troppo limitate o inutilmente potenti. Al giorno d'oggi usufruire dei servizi di cloud computing risultata estremamente

semplice. La maggior parte delle nuove compagnie usufruisce già di questi servizi, mentre aziende meno recenti abbandonano gradualmente la loro infrastruttura di rete per migrare verso questo nuovo paradigma.

Con il diffondersi di questo paradigma, la struttura logica delle applicazioni software ha subito dei cambiamenti. Infatti se prima la maggior parte delle strategie di sviluppo software era disegnata per produrre applicazioni monolitiche, ovvero un unico blocco in grado di svolgere tutto il lavoro necessario a garantire il servizio, nell'ultimo periodo è diventato sempre più utilizzato uno sviluppo software basato sui **microservizi**, secondo cui l'applicazione software viene smontata in tante piccole componenti che comunicano tra loro. Proprio per questo motivo, le applicazioni a microservizi si integrano perfettamente con i servizi offerti dal cloud computing, introducendo però nuove difficoltà da gestire.

1.2 Esigenza

Sebbene siano molti i vantaggi offerti dalle applicazioni a microservizi, come la maggiore flessibilità nel gestire l'intera applicazione e la possibilità di scalare i singoli componenti in maniera del tutto indipendente dal resto, questo nuovo paradigma presenta nuove complicazioni non riscontrate in precedenza con lo sviluppo software monolitico. I processi di debugging e di monitoraggio del sistema, con questo nuovo tipo di applicazioni, risultano un po' più complessi poiché ad una singola richiesta ricevuta dal sistema, sono coinvolti più microservizi che devono comunicare tra loro per scambiarsi le informazioni necessarie.

Di conseguenza nasce l'esigenza di sviluppare delle infrastrutture specializzate nell'osservare e monitorare l'intero sistema, che garantiscano l'assenza di malfunzionamenti nell'esecuzione delle applicazioni e la corretta erogazione dei servizi. Tali piattaforme consentirebbero di individuare rapidamente eventuali errori all'interno del sistema, permettendo così agli sviluppatori di risolvere in tempi brevi.

Un processo fondamentale delle applicazioni a microservizi è la comunicazione tra i diversi componenti. Senza di essa, infatti, l'intera applicazione non funzionerebbe correttamente e l'erogazione del servizio risulterebbe impossibile. Diventa importante quindi che durante il monitoraggio delle applicazioni a microservizi, le piattaforme non si limitino a verificare che i singoli componenti siano correttamente in esecuzione, ma devono studiare ed analizzare tutte le comunicazioni presenti nella rete dei microservizi.

1.3 Obiettivi

Il presente lavoro di tesi intende approfondire il tema dell'osservabilità di un sistema, analizzandone i vantaggi e valutando l'importanza e le potenzialità assunte al giorno d'oggi.

L'obiettivo è quello di sviluppare una piattaforma di osservabilità eseguita su Kubernetes, utilizzando solo strumenti open source. Il compito di questa piattaforma sarà quello di monitorare il sistema sia dall'esterno che dall'interno, elaborando le informazioni raccolte all'interno del cluster.

Infine, si vuole elaborare una rappresentazione grafica che sintetizzi in maniera efficace e dettagliata lo stato attuale e passato del sistema, così da rendere più semplice all'operatore umano il processo di monitoraggio.

1.4 Risultati attesi

Tra i risultati attesi, si auspica il corretto funzionamento di un'applicazione a microservizi instrumentata con librerie esterne al fine di garantire l'osservabilità e lo sviluppo di una infrastruttura di monitoraggio resa solida e sicura grazie a tecniche di altissima affidabilità.

Infine ci si aspetta di progettare delle dashboards personalizzate ed interattive, di facile comprensione umana, che attraverso chiare rappresentazioni grafiche sintetizzino tutte le informazioni necessarie per comprendere lo stato interno ed esterno del sistema.

1.5 Presentazione del progetto

Il progetto è diviso in tre fasi distinte: studio dell'osservabilità e dei diversi tipi di monitoraggio, disegno della strategia da attuare ed infine sviluppo della piattaforma di osservabilità.

Durante la prima fase, grazie a diverse ricerche e allo studio di documenti che riguardano l'argomento, è stato possibile studiare nel dettaglio la teoria dell'osservabilità dei sistemi. È stata data particolare attenzione alle recenti tecniche di monitoraggio e a nuovi strumenti che garantiscono l'osservabilità, come logs, metriche e tracciature.

Successivamente, si è passati al disegno della strategia per sviluppare la piattaforma di osservabilità. L'obiettivo della piattaforma è generare, elaborare e mostrare informazioni relative allo stato interno del sistema, lavorando solo con strumenti open source ed integrando strutture ad alta affidabilità.

Infine si è passati allo sviluppo pratico della piattaforma, seguendo le linee guida della strategia disegnata e testando il progetto sull'applicazione demo a microservizi Online Boutique offerta da Google.

La fase di sviluppo è caratterizzata da quattro fasi di progettazione che permette il raggiungimento degli obiettivi fissati, grazie al corretto funzionamento della piattaforma di osservabilità:

- *generazione*: in questa prima fase, l'obiettivo è quello di instrumentare il codice sorgente dell'applicazione Online Boutique per rendere i suoi microservizi osservabili. Questo permette la generazione di una collezione di dati finalizzati al monitoraggio interno.

- *acquisizione ed elaborazione*: una volta garantita dai microservizi l'esposizione di informazioni utili al monitoraggio all'esterno dell'applicazione, queste vengono acquisite ed elaborate da strumenti specializzati.

- *memorizzazione*: in seguito all'elaborazione delle informazioni, i nuovi dati devono essere memorizzati in strutture dedicate, così da poter essere accessibili da altre applicazioni.

- *visualizzazione*: infine, interrogando i dati memorizzati, si progettano le rappresentazioni grafiche che riassumono lo stato interno dell'applicazione mostrando le informazioni più rilevanti.

1.6 Struttura del documento

In questo documento è descritto tutto il risultato prodotto dal lavoro svolto ovvero le ricerche scientifiche e cosa queste hanno dimostrato, la spiegazione della strategia disegnata e come si è arrivati a certe scelte ed infine la descrizione dello sviluppo software di una piattaforma di osservabilità eseguita in un cluster Kubernetes.

Il documento inizia con una breve introduzione sull'evoluzione dei sistemi informatici avvenuta negli ultimi anni, per poi riportare il risultato delle ricerche effettuate in merito ai concetti fondamentali dell'osservabilità e su come garantirla nelle applicazioni moderne. Successivamente, seguendo un approccio bottom-up, viene descritta la soluzione ideata per sviluppare la piattaforma di osservabilità, confrontando i diversi strumenti disponibili da integrare e motivando le scelte fatte: si inizia dall'instrumentazione dell'applicazione Online Boutique con tecniche che garantiscano l'osservabilità, in seguito si installano e si configurano i tools scelti per costruire la piattaforma, specializzati in questo tipo di elaborazioni, infine si

processano i dati raccolti per mostrare graficamente una sintesi dello stato complessivo del sistema.

Nel **capitolo 2** si descrivono le caratteristiche dei sistemi moderni e la loro evoluzione degli ultimi anni, in particolare di come si è passati da sistemi on-premises a sistemi in cloud. Si descrive come questa migrazione ha permesso anche un'evoluzione nello sviluppo software delle applicazioni, che con il tempo si stanno convertendo da applicazioni monolitiche ad applicazioni a microservizi.

Infine, un piccolo cenno agli strumenti diventati più popolari in questo ambiente informatico, come Kubernetes, che si sta confermando il principale strumento di orchestrazione di container, ed Istio, un tool in grado di monitorare e di estendere l'orchestrazione dei microservizi in esecuzione.

Nel **capitolo 3** si espone la ricerca fatta sull'osservabilità e sulla sua importanza nelle applicazioni moderne. In particolare, si descrive la differenza tra osservabilità e monitoraggio, che spesso non è chiara, e si mostrano i nuovi approcci di sviluppi software per le piattaforme di monitoraggio. In merito a questo argomento si concentra l'attenzione su quelli che sono in letteratura i tre pilastri fondamentali dell'osservabilità: logs, metriche e tracciature.

Il capitolo si conclude con una descrizione dell'alta affidabilità dei sistemi e di quali sono le tecniche per garantirla all'interno di una piattaforma di monitoraggio.

Il **capitolo 4** è dedicato alla descrizione dettagliata del progetto, vertendo in particolar modo sul disegno della strategia di sviluppo della piattaforma di osservabilità e sui passaggi pratici, grazie all'uso di strumenti di open source, per eseguirla all'interno di un cluster Kubernetes.

Vista la possibilità di raggiungere gli stessi obietti con strumenti diversi, nella descrizione della strategia ideata ci si sofferma sulle motivazioni che hanno indirizzato la scelta verso gli strumenti utilizzati. Sono mostrati, infatti, i diversi confronti tra strumenti simili, descrivendo pro e contro di ciascuno.

Si spiega, inoltre, come instrumentare un'applicazione per garantire l'osservabilità dei suoi microservizi e come dalle informazioni generate sia possibile, grazie ad una accurata elaborazione, costruire delle rappresentazioni grafiche che garantiscano il monitoraggio completo del sistema.

Nella fase di costruzione pratica sono definiti tutti i passaggi necessari per costruire la piattaforma all'interno del cluster. È necessario configurare correttamente i vari strumenti utilizzati, senza quali non è possibile garantire un buon risultato.

Si presentano anche le estensioni successivamente implementate per offrire nuove funzionalità alla piattaforma progettata, come la generazione delle metriche dai logs e l'integrazione dell'alta affidabilità dei dati elaborati.

La parte finale del capito analizza i risultati ottenuti attraverso lo sviluppo di questo progetto, confrontando il livello di funzionalità raggiunto dalla piattaforma e Istio, una tecnologia open source che fornisce agli sviluppatori un modo per connettere, gestire e proteggere in maniera costante reti di diversi microservizi indipendentemente dalla piattaforma, dall'origine o dal fornitore. Infine, anche un breve confronto tra l'instrumentazione manuale di librerie di osservabilità rispetto ad una eventuale instrumentazione automatica.

Con il **capitolo 5** si conclude il documento, sintetizzando gli obiettivi raggiunti rispetto alle esigenze iniziali. Infine si considerano eventuali sviluppi futuri, come ad esempio l'allarmistica intelligente.

Capitolo 2

Cenni teorici sui sistemi moderni e sull'osservabilità

2.1 Introduzione ai sistemi moderni

Alla fine degli anni '90 del Novecento, a seguito della forte crescita di Internet e di un settore informatico sempre più in evoluzione, molte compagnie hanno dovuto investire parecchie risorse per progettare da zero la loro infrastruttura tecnologica. Era fondamentale infatti, avere una propria piattaforma di rete per essere presenti sul web e per garantire i propri servizi agli utenti finali connessi da ogni parte del mondo.

Una componente chiave di queste infrastrutture era rappresentata dai ***server***, capaci di gestire le richieste dei consumatori e di memorizzare al loro interno qualsiasi tipo di dato. Proprio per questo ruolo così importante, la gestione dei server era un lavoro molto complesso e delicato.

Con il passare del tempo, il numero di server necessari ad un'azienda per erogare i propri servizi tramite Internet, era diventato sempre più alto ed ingestibile. La maggior parte delle compagnie non avevano all'interno della loro struttura aziendale una disposizione fisica ben definita per questi dispositivi, ma spesso i server erano posizionati in modo del tutto casuale. Questo disordine ha causato una serie di problemi organizzativi e di sicurezza per molte aziende.

Nasce così una soluzione che prevedeva che tutti i server aziendali fossero raggruppati e posizionati all'interno di singole stanze chiamate ***datacenter***. Ancora oggi, i datacenter risultano essere tra le strutture tecnologiche più importanti per un'azienda. Inoltre, con gli anni si è assistito ad una loro evoluzione: dal momento in cui le operazioni informatiche sono divenute cruciali per i business delle aziende, i datacenter includono generalmente delle componenti ridondanti e di backup,

infrastrutture per la gestione dell'energia elettrica, connessioni di comunicazioni dati, controlli ambientali (come l'aria condizionata e la soppressione degli incendi) e diversi dispositivi per la sicurezza.

Sebbene la costruzione dei datacenter abbia migliorato l'organizzazione dell'infrastruttura tecnologia all'interno delle aziende, ben presto ci si è ritrovati di fronte ad un nuovo problema: un altissimo numero di server in parte inattivi, in quanto era impossibile consolidare più applicazioni su una singola macchina. Infatti, la macchina principale del server poteva essere in esecuzione con un solo sistema operativo e questo provocava diversi malfunzionamenti nel momento in cui erano processati più servizi contemporaneamente.

I sistemi operativi presentavano molte limitazioni sia a livello software sia nella gestione delle risorse hardware. In particolare non riuscivano:

- *a isolare le componenti condivise che offrivano*: le applicazioni che richiedevano tra i requisiti diverse versioni di una stessa libreria del sistema operativo, potevano presentare malfunzionamenti e avere comportamenti imprevedibili.
- *a gestire le risorse hardware per le singole applicazioni*: una richiesta elevata di risorse hardware per l'esecuzione di un'applicazione (es. consumo di CPU), penalizzava il corretto funzionamento degli altri servizi sulla stessa macchina.
- *a garantire un forte isolamento tra applicazioni*: il blocco e malfunzionamento di un servizio poteva facilmente compromettere l'esecuzione delle altre applicazioni.

Un altro motivo per cui era impossibile consolidare più applicazioni su uno stesso server era la legge di Moore. Con la nascita di applicazioni sempre più complete e complesse, bisognava migliorare le performance dei singoli server, ma si era raggiunto un punto critico: era diventato impossibile aumentare la frequenza delle singole CPU e l'unico modo per sopperire a questo problema era aumentare il numero di processori.

Il risultato di tutto ciò è stato lo sviluppo di datacenter con un'enorme quantità di server che lavoravano con molte CPU, molto spesso inutilizzati, a fronte di un elevato consumo di correte elettrica.

Era chiaro che bisognasse trovare un nuovo approccio allo sviluppo dei datacenter che permettesse di massimizzare tutte le risorse disponibili.

Data l'impossibilità di consolidare più applicazioni su un singolo server con un singolo sistema operativo, si adottò una nuova soluzione: la **virtualizzazione**.

La virtualizzazione è una tecnica ancora oggi molto utilizzata e perfezionata con gli anni. Essa prevede di non installare le applicazioni direttamente sull'hardware del server, ma di installare prima un sistema operativo particolare chiamato

Hypervisor. Il compito dell'Hypervisor è quello di gestire i veri sistemi operativi, definiti ***Guest OS***, installati su delle partizioni hardware virtuali, chiamate appunto ***macchine virtuali***.

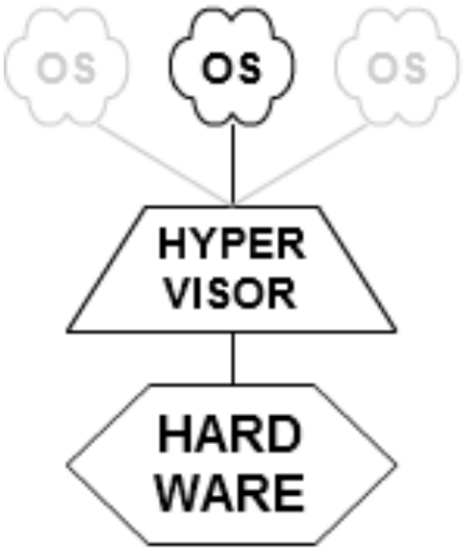

Figura 2.1: Hypervisor

L'Hypervisor è in grado di creare queste macchine virtuali poiché, essendo a contatto con l'hardware, attraverso dei processi software riesce ad astrarre le componenti fisiche dell'elaboratore su cui è in esecuzione. Questo garantisce la corretta produzione di una macchina virtuale, su cui successivamente verrà installato il Guest OS.

Tramite questo processo è quindi possibile creare su un singolo server più macchine virtuali, ognuna con un sistema operativo diverso. Questo ci fornisce la possibilità di eseguire più applicazione sul singolo server andando a sfruttare il più possibile le risorse hardware del datacenter.

Con l'introduzione della virtualizzazione per progettare data center ottimizzati, è iniziato a crescere notevolmente l'interesse per le macchine virtuali, sopratutto per la loro proprietà di essere indipendenti dall'hardware su cui sono in esecuzione. Se per un'azienda prima era importante la configurazione e la gestione dei server, con questa evoluzione questa preoccupazione viene meno. Le macchine virtuali infatti possono essere eseguite su qualsiasi tipo di server, poiché tutto è gestito automaticamente dall'Hypervisor, che analizzando il data center decide dove allocarle.

È questo il punto di partenza del **cloud computing**: non è più necessario preoccuparsi della macchina hardware, ma solo dei processi che vorremo farle

eseguire.

2.1.1 Cloud Computing

Il cloud computing è un paradigma di distribuzione su richiesta delle risorse IT tramite Internet, con una tariffazione basata sul consumo. Piuttosto che acquistare, possedere e mantenere i data center e i server fisici, è possibile accedere a servizi tecnologici, quali capacità di calcolo, storage e database, sulla base delle proprie necessità affidandosi a un fornitore cloud.

Al giorno d'oggi, i servizi di cloud computing, risultano estremamente comodi per qualsiasi tipo di azienda che necessita di una infrastruttura informatica e offrono vantaggi sia di tipo organizzativo sia economici. I principali punti di forza sono:

- *Agilità*: il cloud permette di accedere in modo semplice e veloce a diverse tecnologie, così da poter innovare e costruire qualsiasi tipo di infrastruttura desiderata. Inoltre, a seconda delle necessità è possibile aumentare rapidamente le risorse infrastrutturali, come calcolo, storage e database, ma anche capacità di calcolo per Internet of Things, machine learning e altro.

- *Elasticità*: usufruendo dei servizi del cloud computing, non si ha la preoccupazione di allocare in anticipo una quantità maggiore di risorse di quante siano realmente necessarie. Infatti, eventuali picchi nei livelli di attività aziendali, potranno sempre essere gestiti successivamente.

  Questo permette di ridimensionare le risorse usate, aumentando o riducendo il carico lavorativo in modo istantaneo, adattandolo alla necessità attuale dell'azienda.

- *Costo*: il cloud permette di evitare spese di capitale (per esempio, per data center e server fisici) in favore di una spesa variabile, pagando solo per le risorse IT realmente consumate.

- *Localizzazione*: grazie alle risorse cloud, è possibile espandere i servizi aziendali in qualsiasi regione geografica e distribuire le applicazioni globalmente in pochi semplici passaggi.

Con i servizi di cloud computing, il cliente non ha nessuna responsabilità dell'infrastruttura hardware, ma è l'azienda fornitrice che gestisce i data center e integra un'accurata virtualizzazione delle risorse. In questo modo, il cloud provider garantisce tre tipi principali di cloud computing e ciascun tipo fornisce livelli differenti di controllo, flessibilità e gestione, così che chi ne usufruisce possa selezionare il set adatto alle proprie esigenze.

I tre tipi di cloud computing sono:

- Iaas (Infrastructure as a Service): l'IaaS contiene i blocchi di costruzione predefiniti per l'IT sul cloud. Generalmente fornisce accesso a funzionalità di rete, computer (virtuali o su hardware dedicati) e spazi di archiviazione di dati. L'IaaS offre il più elevato livello di flessibilità e controllo gestionale delle risorse tecnologiche.

- PaaS (Platform as a Service): usufruendo di servizi PaaS, il cliente non deve più gestire l'infrastruttura sottostante (in genere hardware e sistemi operativi). In questo modo è possibile concentrarsi sulla distribuzione e sulla gestione delle applicazioni. Questa soluzione è caratterizzata dal massimo livello di efficienza in quanto non è più necessario dedicarsi ad attività quali l'approvvigionamento delle risorse, la pianificazione della capacità, la manutenzione del software, l'applicazione di patch o qualsiasi altro tipo di attività onerosa che possa interessare l'esecuzione delle applicazioni.

- Saas (Software as a Service): il SaaS offre un prodotto completo che viene eseguito e gestito dal fornitore di servizi cloud. Nella maggior parte dei casi, quando si parla di SaaS si fa riferimento alle applicazioni per utenti finali (come un servizio e-mail basato sul Web). Con il servizio SaaS, non ci si preoccupa della manutenzione o della gestione dell'infrastruttura sottostante su cui il servizio stesso si basa, ma il cliente finale deve solo pensare a come utilizzare il software specifico.

Al giorno d'oggi sono molti gli utenti e le compagnie che usufruiscono di servizi di cloud computing ed è importante che le operazioni svolte e i loro dati siano protetti e al sicuro. Sfortunatamente, la percezione comune è che i servizi di cloud computing non garantiscano un livello di sicurezza adeguato. Capita spesso, infatti, che aziende e utenti non si sentano protetti nella condivisione dei propri dati con un provider esterno ed erroneamente si pensa che le infrastrutture di cloud computing siano meno sicure delle tradizionali infrastrutture informatiche.

Il cloud computing, proprio come qualsiasi altra tecnologia, presenta dei rischi associati alla sicurezza, ma assolutamente risolvibili attraverso una giusta organizzazione dell'infrastruttura IT da parte del cloud provider. L'intera infrastruttura, infatti, deve essere disegnata attentamente, con lo scopo di bloccare qualsiasi tipo di malfunzionamento ed eventuali attacchi informatici volontari, che possono essere effettuati dall'esterno della struttura aziendale ma anche dall'interno. Questo significa che il cloud provider ha il dovere non solo di sviluppare un livello di protezione che isoli l'infrastruttura dal mondo esterno, ma deve anche rendere sicuro l'accesso fisico al data center attraverso controlli approfonditi sui dipendenti che hanno la possibilità di entrarci e di manipolare i server presenti.

Inoltre, quando si usano servizi di cloud computing, c'è la possibilità che la sede e il data center del cloud provider siano in un paese diverso da quello del cliente. Questa situazione, in alcuni casi, può provocare dei problemi di carattere internazionale sia politico sia economico, perché stati diversi applicano delle leggi differenti, sopratutto in materia di normative sulla privacy e di sicurezza dei dati del cliente.

Per tutelare il consumatore da questo tipo di problema, i servizi di cloud computing gestiti all'interno dell'Unione europea sono soggetti alle normative del GDPR (General Data Protection Regulation), poiché esistono dei casi in cui il cloud provider ha la possibilità di modificare il data center su cui vengono archiviate le informazioni anche senza che l'utente finale abbia la possibilità di essere informato. La questione descritta non si pone quando il trasferimento avviene all'interno dei confini dell'Unione europea, in quanto l'art. 1, par. 3, GDPR prevede espressamente che «*la libera circolazione dei dati personali nell'Unione non può essere limitata né vietata per motivi attinenti alla protezione delle persone fisiche con riguardo al trattamento dei dati personali*». Per quanto riguarda, invece, il trasferimento dei dati in Paesi esterni all'Unione è necessario che il titolare del trattamento adotti particolari cautele al fine di cercare di garantire ai dati personali un adeguato livello di protezione. L'utente, al momento della scelta del provider, deve infatti prestare attenzione ai documenti contrattuali, ai Service Legal Agreement, e a tutta la documentazione esistente anche in materia di privacy, valutando accuratamente anche l'impatto che possibili minacce e vulnerabilità possono avere sui dati personali trattati.

2.1.2 Microservizi

Evoluzione da sistemi monolitici a microservizi

L'**architettura monolitica** è considerata come l'approccio tradizionale per lo sviluppo software di applicazioni. Un'applicazione monolitica è costruita come un singolo blocco indivisibile, che normalmente include:

- l'unità lato cliente: l'interfaccia grafica attraverso la quale l'utente finale naviga ed invia le richieste.
- l'unità lato server: gestisce le richieste ricevute ed elabora una risposta che sarà successivamente inoltrata al cliente.
- il database: memorizza i dati utili e importanti per il corretto funzionamento dell'applicazione e viene costantemente interrogato dal server.

Le applicazioni monolitiche sono completamente unificate e tutte le loro funzioni sono gestite ed elaborate in un solo punto. Questo perché il codice sorgente dell'intera applicazione è compilato in una singola unità di deployment.

Avere tutte le funzionalità concentrate in un unico blocco offre una fase di testing molto più semplice, grazie sopratutto a test di tipo end-to-end molto rapidi. Inoltre, risulta più immediato anche il processo di installazione, in quanto non è necessario gestire più componenti ma solo un singolo file.

Tuttavia, questo tipo di architettura non offre un buon livello di modularità. Infatti, un cambiamento minimo a un'applicazione esistente impone un aggiornamento completo e un ciclo di controllo qualità a sé, che rischia di rallentare il lavoro di vari team secondari. In alcuni casi, anche l'affidabilità può essere compromessa, poiché un singolo malfunzionamento in un modulo può bloccare completamente l'intera applicazione.

Un altro punto di debolezza delle applicazioni monolitiche è la scalabilità. È impossibile, infatti, replicare dei componenti indipendenti, ma è possibile creare nuove istanze solo dell'intera applicazione. Infine, anche l'integrazione con le nuove tecnologie è estremamente difficile, visto che l'applicazione andrebbe completamente riscritta.

In seguito all'espansione delle tecniche di virtualizzazione e di servizi di cloud computing sempre più utilizzati, si è riscontrata la necessità di avere un ambiente più isolato per i processi in esecuzione su uno stesso server. Con le macchine virtuali è stato possibile raggiungere un ottimo livello di isolamento garantito a livello hardware, ovvero dalle componenti virtualizzate dall'Hypervisor. Tuttavia, questa tecnica non offre il controllo di alcune risorse fisicamente in comune tra i processi, come la gestione della CPU o l'uso della banda di rete.

È nata così una nuova sfida per il mondo informatico, quella di provare a sviluppare ambienti isolati tra loro all'interno di uno stesso sistema operativo. Questa tecnica è chiamata **virtualizzazione leggera**, ed implementa un livello di isolamento, non più a livello hardware, ma a livello software offerto dal sistema operativo stesso. Infatti, è stato possibile raggiungere questo obiettivo generando delle nuove primitive basate su due funzionalità del Linux Kernel, *namespaces* e *cgroups*. Nascono così i **container**, delle unità standard di software che includo il codice e tutte le sue dipendenze cosicché l'applicazione possa girare velocemente e in maniera affidabile da un computer ad un altro.

Un progetto completamente open source che ancora oggi continua a perfezionare la struttura dei container è **Docker**. Docker implementa un nuovo livello di astrazione all'interno del sistemo operativo, introducendo una semplicissima gestione dei container.

Docker ha anche sviluppato quelle che sono chiamate *Docker Images*, ovvero dei pacchetti eseguibili di software, leggeri ed indipendenti, che includono tutto ciò di cui ha bisogno l'applicazione per essere eseguita correttamente: codice, runtime, tools di sistema, librerie di sistema e impostazioni. I Docker Image diventano container una volta eseguiti dalla Docker Engine, una piattaforma che consente

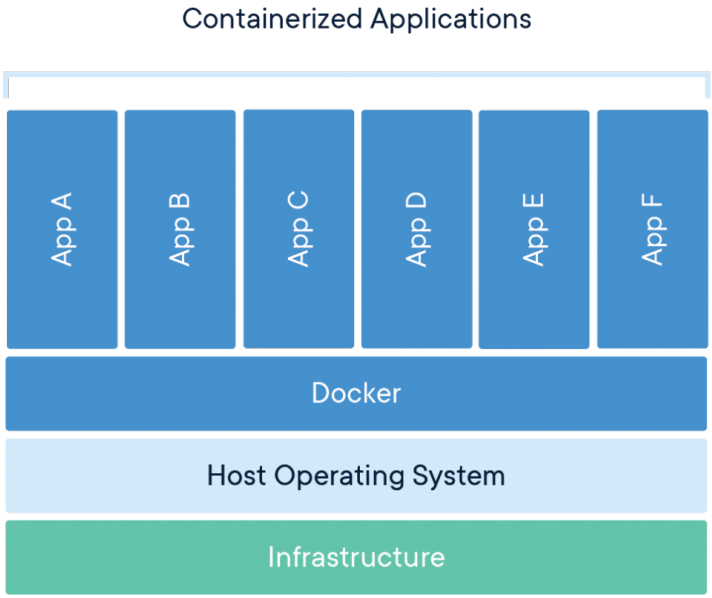

Figura 2.2: Schema dei livelli operativi di una macchina con Docker

alle applicazioni containerizzate di essere eseguite ovunque in modo coerente su qualsiasi infrastruttura, eliminando il problema dalle dipendenze software.

Al giorno d'oggi i Docker Container sono la soluzione migliore per gestire più applicazioni su uno stesso server contemporaneamente, poiché non richiedono un sistema operativo per applicazione. Inoltre, condividendo il kernel del sistema operativo su cui sono in esecuzione risultano estremamente leggeri e molto più veloci delle macchine virtuali.

L'introduzione dei container nei servizi di cloud computing, ha fin da subito evidenziato l'incompatibilità delle tradizionali applicazioni monolitiche con questo paradigma. La scarsa scalabilità di questo tipo di applicazioni e la difficoltà nella manutenzione di un prodotto che deve essere sempre online, rendono difficile il loro sviluppo su un ambiente a container. Al giorno d'oggi, infatti un'architettura monolitica è ancora applicabile alle piccole applicazioni, ma le aziende in crescita non possono permettersi tempi di inattività.

L'attenzione, quindi, si è spostata verso un nuovo tipo di tecnica per sviluppare

software, ovvero l'**architettura a microservizi**. Quello che distingue l'architettura basata su microservizi dagli approcci monolitici tradizionali è la suddivisione dell'applicazione nelle sue funzioni di base. Ciascuna funzione, denominata microservizio, comunica con le altre attraverso interfacce di programmazione delle applicazioni (API). Diventa fondamentale, quindi, poter fare affidamento su una solida infrastruttura di rete, senza la quale non ci potrà essere scambio di informazioni e quindi l'applicazione non funzionerebbe.

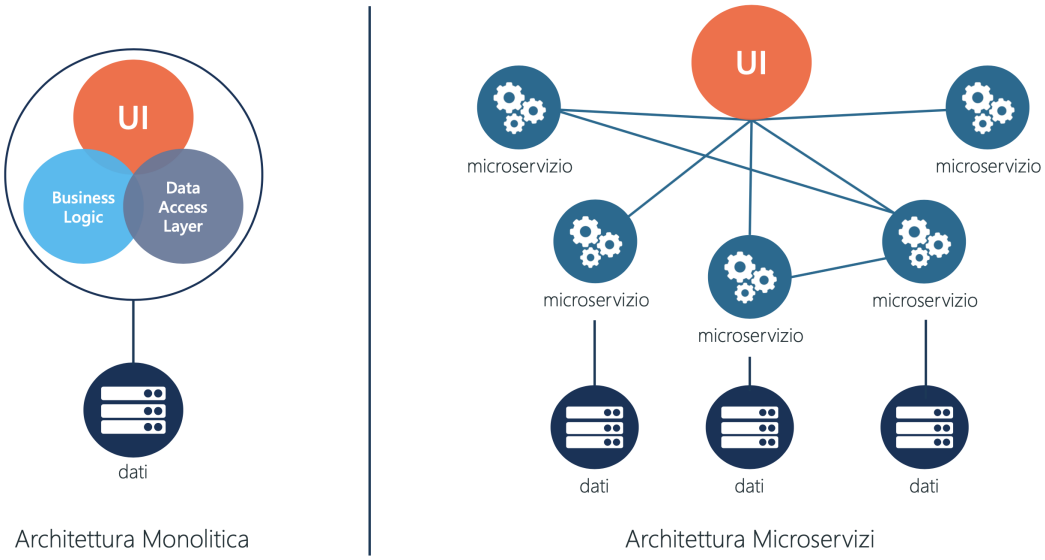

Figura 2.3: Architettura monolitica e architettura a microservizi

Tutti i microservizi di un'applicazione risultano essere completamente separati tra di loro, e possono essere compilati ed implementati in modo indipendente. Pertanto, i singoli microservizi possono funzionare, o meno, senza compromettere gli altri.

Poiché eseguito in modo del tutto indipendente, ciascun microservizio può essere aggiornato, distribuito e ridimensionato per rispondere alla richiesta di funzioni specifiche di un'applicazione. Così, man mano che la domanda per determinati servizi aumenta, i microservizi possono essere distribuiti su più server e infrastrutture, in base alle esigenze aziendali. In questo modo si possono creare più repliche solo di quei microservizi che sono soggetti ad un carico lavorativo maggiore, senza andare a creare una nuova istanza dell'intera applicazione.

Con i microservizi, si può garantire sempre l'esecuzione dell'applicazione. Ciascun microservizio, se costruito correttamente, è indipendente e non influisce sugli altri microservizi nell'infrastruttura. Ciò comporta che l'eventuale errore di un componente non determina il blocco dell'intera applicazione, come avverrebbe

invece con il modello monolitico. Per questo motivo è consigliabile creare più repliche di un singolo microservizio, così da garantire la corretta erogazione del servizio a chi utilizza l'applicazione.

Un altro vantaggio viene garantito dall'indipendenza dalla macchina su cui il microservizio è eseguito. Tale aspetto permette di assegnare diverse potenze computazionali ai vari microservizi dell'applicazione, andando a bilanciare le risorse tra i diversi componenti.

L'architettura basata sui microservizi propone una ristrutturazione dei team di sviluppo. Poiché le applicazioni più grandi vengono suddivise in parti più piccole, per gli sviluppatori è molto più semplice comprendere, aggiornare e migliorare tali componenti, e questo permette di accelerare i cicli di sviluppo. Inoltre, la natura distribuita dei microservizi offre la possibilità di sviluppare più componenti contemporaneamente, consentendo a più sviluppatori di lavorare simultaneamente alla stessa applicazione, ottenendo routine più efficienti.

Inoltre, le applicazioni a microservizi offrono ai team di sviluppo una maggiore flessibilità. Grazie all'introduzione di API indipendenti dal linguaggio, gli sviluppatori sono liberi di scegliere il linguaggio e la tecnologia ottimali per la funzione da creare.

Tuttavia, l'architettura a microservizi introduce anche nuove difficoltà. Essendo un sistema composto da più componenti è necessario scegliere e configurare attentamente le connessioni tra i vari microservizi, talvolta anche introducendo meccanismi di sicurezza. Inoltre, la fase di testing diventa molto più complicata rispetto a quella nelle applicazioni monolitiche. Occorre ricordare che un errore in una parte dell'architettura potrebbe causare un errore in un componente a vari passi di distanza, a seconda del modo in cui i servizi sono strutturati per supportarsi a vicenda.

Anche la fase iniziale di installazione dell'applicazione risulta essere più critica. Per semplificare il deployment, occorre innanzitutto investire notevolmente in soluzioni di automazione. La complessità dei microservizi renderebbe il deployment manuale estremamente difficile.

È proprio la natura distribuita di queste applicazioni che rende più complessa la loro gestione e analisi. In questo tipo di architettura, ad una singola richiesta corrisponde un lungo percorso all'interno della rete dei microservizi. Se si dovesse riscontare un eventuale errore o bug, risulta quindi complicato andare ad identificare quale tra le varie componenti coinvolte ha riportato dei malfunzionamenti. Per questo motivo è diventato sempre più importante lo sviluppo di piattaforme di monitoraggio, che sono in grado di tracciare tutte le comunicazioni effettuate e controllare il corretto funzionamento dei singoli microservizi.

Kubernetes

Con il rilascio di Docker, è stato possibile per gli sviluppatori approcciarsi ai container e al loro sviluppo in maniera molto più semplice. I container devono però essere coordinati e gestiti al meglio, al fine di migliorare il lavoro dei team di sviluppo e automatizzare alcuni processi, ed è qui che entra in gioco l'*orchestrazione dei container*, che consente di:

- automatizzare le funzionalità più "meccaniche", prima eseguite dagli sviluppatori, come controllare il server che ospiterà il container, monitorarne l'avvio e l'esecuzione e gestire eventuali dipendenze;
- interagire con diversi gruppi di container contemporaneamente;
- pianificare ed implementare un registro container, per condividere centralmente le immagini generate;
- fornire funzionalità di rete, storage e security.

A partire da questa necessità, sono quindi nate le più conosciute soluzioni di orchestrazione dei container, tra cui Docker Swarm, Kubernetes e OpenShift. Per ambienti dove sono presenti cluster medio-grandi che eseguono applicazioni complesse, lo strumento più utilizzato per l'orchestrazione è **Kubernetes**.

Kubernetes è un progetto open source di Google, scelto ogni giorno da molte aziende per la sua versatilità e per la possibilità di gestione dello sviluppo su larga scala. Kubernetes, grazie ai suoi numerosi vantaggi, permette di eliminare i processi manuali, sia per quanto riguarda lo sviluppo sia per l'esecuzione delle applicazioni *containerizzate*, e consente di gestire, in modo semplice ed efficiente, cluster di host con container Linux in modo scalabile al fine di gestire al meglio i carichi di lavoro.

I container sono raggruppati in pod, ossia l'unità operativa di base per Kubernetes. Container e pod possono essere ridimensionati fino allo stato desiderato e sono gli sviluppatori a gestirne il ciclo di vita per mantenere operative le applicazioni.

Mentre i container consentono di scrivere porzioni di codice che è possibile eseguire ovunque, Kubernetes fornisce il potenziale per orchestrare e gestire tutte le risorse dei container da un singolo piano di controllo. Vengono forniti i componenti utili per rete, bilanciamento del carico, sicurezza e scalabilità in tutti i nodi Kubernetes che eseguono i container. Kubernetes offre anche un meccanismo di isolamento predefinito sotto forma di *namespace* che consente di raggruppare le risorse dei container in base ad autorizzazione di accesso, ambienti di gestione temporanea e molto altro. Questi costrutti permettono al personale IT di offrire facilmente agli sviluppatori l'accesso a risorse self-service e consentono al team di sviluppo di progettare applicazioni più complesse. Con Kubernetes infatti, è

ancora più facile ottenere un'architettura di microservizi di base che consente la distribuzione rapida e l'orchestrazione scalabile di applicazioni native del cloud.

Kubernetes è ormai così diffuso che le più grandi e conosciute piattaforme per lo sviluppo tramite container, tra cui Docker stessa, integrano ormai nativamente tale soluzione con la propria tecnologia. Proprio per questo motivo Kubernetes e Docker collaborano perfettamente insieme, rappresentando quasi uno standard nel mondo IT.

Istio

Le piattaforme cloud offrono una vasta gamma di vantaggi per le compagnie che le utilizzano. Tuttavia non si può negare che l'adozione di servizi di cloud computing possa mettere a dura prova i team di sviluppo. **Istio** è una piattaforma di service mesh open source che consente di connettere, proteggere, controllare e osservare i microservizi installati.

Ad alto livello, Istio aiuta a ridurre la complessità dei deployments e allevia il carico di lavoro degli sviluppatori. È una *service mesh*, ovvero un sistema che consente di controllare in che modo le diverse componenti di un'applicazione condividono i dati, costituendo un livello infrastrutturale che si sovrappone in modo trasparente alle applicazioni distribuite esistenti.

Il set di funzionalità diversificato di Istio riduce le difficoltà di monitoraggio e analisi dei microservizi. Inoltre, Istio offre anche degli strumenti di orchestrazione che vanno ad estendere quelli Kubernetes. Grazie ai suoi tools di *discovery* e *load balancing*, è in grado di controllare la rete di microservizi, andando a bilanciare adeguatamente il traffico di rete.

Istio è completamente compatibile con altri strumenti esterni di monitoraggio. Infatti, include delle API che permettono a questa piattaforma di integrarsi perfettamente con altri servizi di logging e tracciamento.

Per garantire questo controllo globale dell'ambiente distribuito, l'installazione di Istio all'interno del cluster affianca a tutti i microservizi presenti dei sidecar proxy particolari, in grado di intercettare tutte le comunicazioni in entrata e in uscita dal servizio. Tutti questi sidecar proxy sono gestiti dal pannello di controllo di Istio, tramite il quale è possibile monitorare e orchestrare la rete distribuita di microservizi grazie alle sue funzionalità, come:

- il load balancing automatico;
- un controllo dettagliato del comportamento del traffico;
- la gestione di un livello di politiche;

- la creazione automatica di metriche, logs e tracciature per tutto il traffico interno al cluster;
- impostare protocolli di sicurezza service-to-service.

2.2 Osservabilità

Nel nostro tempo la tecnologia è diventata ormai parte integrante della quotidianità delle persone e con la sua crescita esponenziale, le aziende che offrono di servizi in quest'ambito hanno bisogno di migliorarsi, evolversi rapidamente e di garantire una maggiore affidabilità. Se pensiamo ai tanti servizi che sono offerti online, questi devono essere in esecuzione 24h/24h senza mai interrompersi, poiché anche una pausa di pochi secondi non sarà percepita di buon grado dagli utenti. L'immagine dell'azienda sarà così manomessa e di conseguenza quella piccola interruzione si trasformerà in un danno economico.

Pertanto negli ultimi anni queste compagnie hanno deciso di investire nel miglioramento dell'affidabilità dei propri servizi, iniziando quindi a sviluppare nuove tecniche e nuovi approcci all'interno delle loro infrastrutture.

Sono nate così nuove piattaforme il cui compito all'interno dell'infrastruttura aziendale è monitorare continuamente lo stato delle applicazione e dell'intero sistema, partendo da controlli tipo hardware (CPU in uso, percentuale di RAM occupata, stato della rete, ecc.) salendo pian piano a controlli di tipo software (l'esecuzione delle applicazioni, la raggiungibilità dei servizi, ecc.). In questo modo, si è in grado di individuare la presenza di problemi ed anomalie con il vantaggio anche di sapere subito come intervenire per aggiustare il sistema. Tuttavia questi tipi di piattaforme non garantiscono l'assenza di problemi ed errori, bensì ci forniscono una buona approssimazione dello stato di salute complessivo del sistema.

2.2.1 Importanza dell'osservabilità nei sistemi moderni

In seguito alla migrazione verso ambienti distribuiti in cloud, con la diffusione di architetture a microservizi eseguite nei container, i sistemi informatici hanno raggiunto un livello di complessità molto alto, al tal punto che i tradizionali approcci di monitoraggio non risultano più scalabili e non sono in grado di individuare i nuovi tipi di problemi causati da errori fino ad ora sconosciuti. Questi nuovi errori sono anche chiamati "*unknown unknowns*" e senza un sistema osservabile è praticamente impossibile individuare la causa che li ha generati.

Diventa fondamentale, quindi, integrare il sistema con una nuova instrumentazione finalizzata a comprendere meglio le proprietà di un'applicazione e le sue

prestazioni all'interno di un sistema complesso e distribuito. Questo nuovo tipo di sistema è chiamato **sistema osservabile** e offre diversi strumenti molto utili per andare a scoprire cosa stia effettivamente accadendo dentro le applicazioni in esecuzione.

2.2.2 Differenza tra osservabilità e monitoraggio

E' importante chiarire la differenza tra i due termini *osservabilità* e *monitoraggio*, molto spesso confusi erroneamente. Il termine **osservabilità** fa riferimento ad una proprietà che misura quanto bene può essere descritto lo stato interno di un sistema, studiando gli input e gli output esterni. È quindi un attributo del sistema, non un'attività o uno strumento, che all'interno di una infrastruttura IT ci permette di avere molte più risposte su domande inerenti allo stato del sistema. Invece, il termine **monitoraggio** indica un'attività, che nello specifico viene eseguita sulle applicazione e sul sistema per determinare il loro stato, così da individuare problemi e anomalie.

Poiché i sistemi complessi e distribuiti non sono mai completamente privi di errori e per natura imprevedibili, diventano molto importanti le due metriche *TTD* (time-to-detect), il tempo necessario ad individuare il problema, e *TTR* (time-to-restore), il tempo necessario per risolvere un errore e ripristinare uno stato di salute all'interno del sistema.

Un contributo chiave alla riduzione del TTD è quello di sviluppare ottime piattaforme di monitoraggio, così da capire rapidamente se si è verificato qualche malfunzionamento. Il TTR invece rimane invariato e anche le migliori tecniche di monitoraggio, da sole non sono in grado di ridurlo. Per questo motivo è importante rendere osservabile il nostro sistema, così da poter raccogliere un gran numero di informazioni provenienti direttamente dall'interno dell'applicazione. In questo modo si ha un'idea chiara di cosa non ha funzionato e si può intervenire subito con l'adeguata riparazione, senza perdite di tempo in richere per trovare la natura del problema. Sviluppare insieme quindi, piattaforme di monitoraggio affiancate ad ambienti osservabili permette di ridurre sia il TTD sia il TTR, aumentando l'affidabilità e la manutenzione della piattaforma.

2.2.3 Blackbox e whitebox monitoring

L'introduzione dell'osservabilità ha trasformato le piattaforme di monitoraggio, andando ad adattarle alle nuove esigenze per analizzare le architetture distribuite a microservizi. Per riconoscere quindi se queste piattaforme sono in grado di studiare lo stato interno di un sistema, sono stati definiti due tipi di monitoraggio: il *blackbox monitoring* e il *whitebox monitoring*.

Come suggerisce il nome, nel **blackbox monitoring** l'intero sistema è paragonato ad una scatola nera. La piattaforma di monitoraggio può solo analizzarlo dall'esterno, studiando la sua reazione a determinati input esterni. È possibile monitorare valori legati all'infrastruttura hardware come CPU in elaborazione, RAM occupata, spazio libero sul disco e traffico di rete. Purtroppo, utilizzando solo questa tecnica, non è possibile scoprire cosa effettivamente stia succedendo all'interno dell'applicazione monitorata.

Per poter monitorare le applicazioni dall'interno ed analizzare in maniera dettagliata i loro processi, è necessario sviluppare un nuovo tipo di piattaforma che monitori un *ambiente osservabile*. Questo tipo di tecnica viene chiamata **whitebox monitoring** e permette di studiare il sistema andando ad analizzare dati ed informazioni elaborati direttamente dalle applicazioni stesse e successivamente esposti per essere raggiungibili dall'esterno. È quindi il sistema stesso che descrive il suo funzionamento interno.

Grazie a questa tecnica, il team di sviluppo può studiare lo stato interno del sistema ed è in grado di comprendere come effettivamente stiano lavorando le applicazioni. In questo modo è possibile riconoscere immediatamente eventuali errori presenti dentro il sistema.

Le tecniche di whitebox monitoring sono molto utili in ambienti con applicazioni a microservizi, dove per una singola richiesta sono coinvolti molti componenti. Tuttavia negli ultimi anni, diversi team di sviluppo hanno iniziato ad implementare questo tipo di approccio, anche per monitorare applicazioni monolitiche, che con il tempo sono diventate sempre più complesse, a causa dell'elevata interazione con molte risorse esterne.

2.2.4 I tre pilastri dell'osservabilità

Data la complessità e la natura imprevedibile dei sistemi distribuiti con applicazioni a microservizi, le compagnie hanno iniziato ad integrare sempre di più nei loro ambienti di sviluppo diversi strumenti di osservabilità, nel tentativo di comprendere meglio le modalità di errore e di rintracciare i problemi alla loro causa principale.

Essendo una nuova tecnologia, non esistono standard informatici per garantire l'osservabilità in un sistema, ma è tutto in continua evoluzione con il rilascio giornaliero di nuovi tool e applicazioni. Tuttavia la letteratura informatica, ha definito quelli che sono *i tre pilastri dell'osservabilità*, ovvero tre strumenti che grazie alla loro integrazione nell'ambiente di sviluppo offrono una buona descrizione dello stato interno del sistema: i *logs*, le *metriche* e le *tracciature*.

Logs

I logs sono linee di testo immutabili che registrano gli eventi avvenuti all'interno del sistema. La struttura del log è composta da tre parti: il *timestamp* registrato al momento della creazione del log, un *codice di stato* che descrive la categoria dell'evento e il *payload*, che conserva informazioni relative all'evento.

In particolare, i dati contenuti nel payload risultano essere molto utili per le fasi di monitoraggio. Grazie alla possibilità di arricchire questi dati, si possono aggiungere molte informazioni che descrivono il contesto presente al momento della registrazione del log. Questo genere di informazioni aiuta molto gli sviluppatori a comprendere meglio l'ambiente del sistema ed eventuali cause di malfunzionamenti.

Figura 2.4: Struttura del log

Generare ed integrare i logs in una applicazione risulta essere una procedura molto semplice, grazie sopratutto alla possibilità di formattarli in formato JSON, uno standard che al giorno d'oggi è compatibile con quasi tutti i linguaggi di programmazione. Inoltre, i logs si presentano come la miglior soluzione per la visualizzazione di informazioni ad alta granularità, sopratutto in presenza di un contesto molto complesso.

Sebbene la loro implementazione sia semplice, il più grande svantaggio dei logs è l'elevata complessità di elaborazione, sopratutto in fase di memorizzazione e di trasporto. Inoltre i logs conservano ogni singola richiesta avvenuta all'interno del sistema. Perciò quando si hanno in esecuzione applicazioni a microservizi è importante aggregare tutti i logs dei singoli container in un unico luogo, così da avere una corretta descrizione di cosa sia successo all'interno dell'intera applicazione. Questa operazione di aggregazione in alcuni casi può risultare eccessivamente dispendiosa, sopratutto in presenza di applicazioni molto complesse e distribuite. Diventa fondamentale quindi, alleggerire il traffico dovuto al trasporto dei logs, impostando dei controlli e delle tecniche di filtraggio, che permettano l'aggregazione solo dei logs effettivamente necessari.

Metriche

Le metriche sono rappresentazioni numeriche di dati misurati in un intervallo di tempo e sfruttano i modelli matematici per ricavare il comportamento del sistema su periodi temporali, presenti e futuri.

Dal momento in cui i numeri risultano perfetti per la memorizzazione, l'elaborazione, la compressione e il recupero, le metriche consentono una conservazione più lunga dei dati e l'uso di query più semplici. Ciò rende le metriche perfettamente adatte alla creazione di dashboard che riflettono le tendenze storiche del sistema. Inoltre, dopo un certo periodo di tempo, i dati conservati possono essere aggregati in frequenze giornaliere o settimanali.

I moderni sistemi di monitoraggio implementano un'evoluzione della struttura della metrica, che agevola le fasi di analisi e filtraggio. Le metriche sono definite da un *nome univoco* che identifica la metrica, e da una serie di *coppie chiave-valore* aggiuntive chiamate *labels*. Ciò consente un alto grado di dimensionalità nel modello di dati.

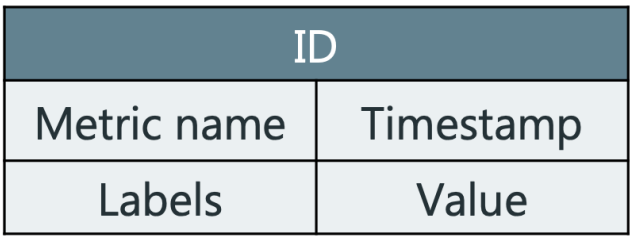

Figura 2.5: Struttura della metrica

In generale, il più grande vantaggio delle metriche rispetto ai logs è che, a differenza della generazione e memorizzazione dei logs, il trasporto e la memorizzazione delle metriche, a parità di traffico utente, risulta essere più contenuto, senza comportare un forte aumento dei dati.

Con le metriche, un aumento del traffico verso un'applicazione non comporterà un aumento significativo dell'utilizzo del disco, della complessità dell'elaborazione, della velocità di visualizzazione e dei costi operativi come per i logs. L'archiviazione delle metriche aumenta con più permutazioni dei valori delle labels (ad esempio, quando vengono attivati più host o container, quando vengono aggiunti nuovi servizi o quando i servizi esistenti vengono maggiormente strumentati), ma l'aggregazione lato utente può garantire che il traffico delle metriche non aumenti proporzionalmente al traffico degli utenti.

Le metriche, una volta raccolte, sono più malleabili per trasformazioni matematiche, probabilistiche e statistiche come il campionamento, l'aggregazione, il riepilogo e la correlazione. Queste caratteristiche rendono le metriche più adatte per segnalare lo stato di salute generale di un sistema.

Le metriche sono anche più indicate per gestire ed attivare gli avvisi, poiché l'esecuzione di query su un database basato sulle serie temporali è molto più efficiente

rispetto all'esecuzione di una query su un sistema distribuito, dove bisognerebbe aggregare tutti i risultati prima di decidere se è necessario attivare un avviso.

Se utilizzati in modo ottimale e con il giusto compromesso, i logs e le metriche garantiscono buona descrizione dello stato interno del sistema. Tuttavia, sebbene questi possano essere sufficienti per comprendere le prestazioni e il comportamento dei singoli servizi non sono sufficienti per comprendere la durata e il percorso di una richiesta attraverso il sistema distribuito.

Questo problema è risolto dalle tracciature, uno strumento adatto per la comprensione di applicazioni a microservizi.

Tracciature

Una tracciatura è una rappresentazione grafica di una serie di eventi distribuiti correlati in modo casuale che rappresentano il flusso di richieste end-to-end attraverso un sistema distribuito. Una singola tracciatura può fornire informazioni sia sul percorso attraversato da una richiesta che sulla struttura di una richiesta. Avere una rappresentazione grafica del percorso fatto da una richiesta consente al team di sviluppo di comprendere i diversi servizi coinvolti durante la sua elaborazione.

Sebbene le discussioni sulle tracciature tendano a ruotare attorno alla sua utilità in un ambiente di applicazioni a microservizi, è giusto suggerire che qualsiasi applicazione sufficientemente complessa che interagisce e si contenda diverse risorse come la rete, il disco o un mutex, può trarre vantaggio nell'implementazione delle tracciature.

All'inizio di una richiesta, viene assegnato un ID univoco globale, che viene quindi propagato in tutto il suo percorso, in modo tale che ogni funzione sia in grado di inserire o arricchire con informazioni utili, il contesto della tracciatura, prima di saltare al punto successivo nel flusso tortuoso di una richiesta. Ogni salto lungo il flusso, che solitamente corrisponde al passaggio da un microservizio ad un altro, è rappresentato come uno span.

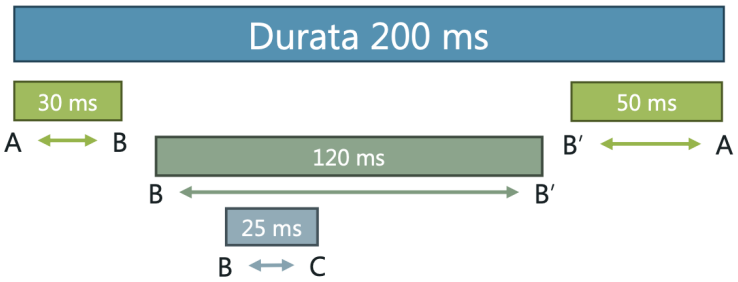

Figura 2.6: Struttura della tracciatura

Quando il flusso di esecuzione raggiunge i diversi punti instrumentati in uno di questi servizi, viene emesso un record insieme ai metadati. Questi record vengono solitamente registrati in modo asincrono e memorizzati su disco prima di essere inviati ad un collector, che può quindi ricostruire il flusso di esecuzione in base alle informazioni dei diversi record emessi dalle parti del sistema.

Avere una comprensione dell'intero ciclo di vita delle richieste, rende possibile eseguire il loro debug sopratutto quando abbracciano più servizi. Ciò permette di individuare facilmente l'origine dell'aumento della latenza o dell'utilizzo delle risorse.

Le tracciature sono di gran lunga tra gli strumenti più difficili da integrare in un'infrastruttura esistente, perché affinché sia veramente efficace, ogni componente nel percorso di una richiesta deve essere modificata per propagare le informazioni di tracciamento. Tuttavia, la letteratura informatica sostiene che avere delle lacune nel flusso di una richiesta non supera gli svantaggi, poiché si preferisce, in ogni caso, avere una tracciatura frammentata rispetto a non avere nessuna informazione. Infatti, una tracciatura incompleta risulta sempre utile e raramente può rendere più difficile la fase di debug.

Il costo delle tracciature non è così elevato come quello dei log, principalmente perché le tracciature vengono quasi sempre campionate in modo consistente per ridurre il sovraccarico runtime e i costi di archiviazione. Le decisioni di campionamento che possono essere prese sono:

- all'inizio di una richiesta, prima che qualsiasi tracciatura sia generata;
- alla fine, dopo che tutti i servizi partecipanti hanno registrato le tracciature per l'intero corso dell'esecuzione della richiesta;
- durante il flusso della richiesta, quando solo i servizi a valle segnalerebbero la tracciatura;

2.2.5 Alta affidabilità

L'integrazione dell'osservabilità all'interno di un sistema distribuito prevede che i dati raccolti vengano archiviati ed analizzati, permettendo agli sviluppatori di studiare approfonditamente il comportamene del sistema, sopratutto in condizioni di particolare utilizzo delle applicazioni. In questo modo, il team di sviluppo, a partire dai malfunzionamenti e dei problemi rilevati, può evolvere il sistema di monitoraggio in esecuzione e correggere il funzionamento delle applicazioni installate.

La memorizzazione dei dati, quindi, rappresenta un tema molto delicato, ed in quanto tale è necessario integrare alla piattaforma di monitoraggio tecniche e

strumenti di alta affidabilità, che garantiscano la conservazione delle informazioni raccolte ed evitino eventuali perdite di dati, sopratutto in seguito a malfunzionamenti e comportamenti inaspettati del sistema.

Secondo la letteratura informatica, per garantire l'alta affidabilità all'interno di un sistema, è necessario integrare alla piattaforma sviluppata strumenti di *ridondanza* e di *distribuzione* del database.

Ridondanza

La ridondanza dei dati è una condizione creata all'interno di un database in cui la stessa porzione di dati è conservata in due o più luoghi separati. Questa tecnica viene eseguita a scopo di backup e ripristino dei dati.

A tal proposito però, la ridondanza dei dati produce mancanza di efficienza nella gestione e nell'occupazione della memoria, e ciò può causare anche inconsistenza dei dati. Diventa importante quindi aggiornare correttamente tutte le diverse repliche quando un dato viene modificato o eliminato. Inoltre bisogna prestare attenzione anche a non memorizzare lo stesso dato con nomi diversi, altrimenti si complicherebbe la gestione del database e aumenterebbe la possibilità di errore producendo incertezza su quale sia la versione corretta. Un altro punto fondamentale è il formato usato per la memorizzazione, che deve essere sempre lo stesso così da evitare incongruenze.

È chiaro quindi che integrare ridondanza dei dati all'interno di un sistema richieda una serie di attenzioni: è necessario affidarsi a tecniche e strumenti specializzati in questo approccio, rendendo più semplice la sua implementazione.

Distribuzione

All'interno di un sistema, più i dati da conservare crescono, più rallentano le operazione sul database, che funge da collo di bottiglia per le prestazioni dei servizi. Sebbene sia possibile aggiungere nuove risorse fisiche per supportare le operazioni del database, si arriverà ad un punto di stallo e a causa di questi limiti hardware le operazioni elaborate sui dati ne soffriranno.

Diventa quindi necessario migliorare le prestazione del database attraverso la *scalabilità orizzontale* dei dati, anche chiamata *sharding*. Lo sharding è una tecnica di distribuzione e memorizzazione di un singolo insieme di dati che viene diviso in più sottoinsiemi, chiamati anche *shards*. L'idea principale è quella di distribuire i diversi shards all'interno dell'ambiente di sviluppo, permettendo la memorizzazione di dataset più grandi conservando prestazioni elevate.

Tuttavia, se la distribuzione orizzontale in shards non è implementata correttamente, è possibile che si vadano ad incrementare i tempi delle operazioni di

lettura dei dati, poiché la ricerca di un singolo dato si estende su tutti i singoli database presenti nel sistema. Un aumento eccessivo dei tempi di elaborazione non garantisce il corretto funzionamento del sistema, poiché si andrebbero a penalizzare tutte le altre operazioni in esecuzione, con il rischio di compromettere l'intera infrastruttura. Per questo motivo, le tecniche di distribuzione dei database devono essere affiancate da algoritmi di ottimizzazione in grado di velocizzare le operazioni di lettura e scrittura.

Capitolo 3

Sviluppo della piattaforma di osservabilità

Compresa l'importanza dell'osservabilità negli ambienti distribuiti a microservizi, si passa all'applicazione pratica del progetto. Come anticipato, il lavoro di tesi consiste nel disegnare e sviluppare una piattaforma di osservabilità che sia in grado di analizzare e segnalare anomalie generalmente difficili da individuare da un operatore umano, implementando applicazioni open source attualmente disponibili per gli utenti.

Prima di passare alla fase di sviluppo della piattaforma, è necessario preparare il sistema che si andrà ad analizzare e monitorare. Tutte le operazioni di sviluppo sono state eseguite su una macchina virtuale con sistema operativo *CentOS 7 x86-64 minimal*, eseguita localmente sul computer. È stata scelta la versione *minimal* del sistema operativo, poiché non essendo necessario ai fini del progetto interagire con l'interfaccia grafica di CentOS, si è preferito installare un sistema operativo gestito esclusivamente dal terminale. Inoltre, disponendo di risorse hardware limitare sul computer principale, la versione *minimal*, essendo molto più leggera e performante, ha garantito la corretta installazione ed esecuzione di applicazioni software di media-alta complessità.

In seguito alla configurazione della macchina virtuale e della sue risorse di rete, è stato installato *Docker*, per l'uso delle applicazioni nei container, e *minikube*, per l'orchestrazione delle applicazioni e container in esecuzione. **Minikube** è un tool che permette di installare Kubernetes localmente, eseguendo un singolo nodo del cluster, facilitando l'uso su un computer personale.

Per poter sviluppare la piattaforma di osservabilità e accertarsi che il monitoraggio funzioni correttamente, è necessario installare un'applicazione a microservizi

all'interno del cluster. Per il progetto si è scelto di usare *Online Boutique*, un'applicazione open source sviluppata da Google. L'uso principale di questa applicazione sarà quello di testare la piattaforma di osservabilità, e che tutte le sue componenti siano configurate come desiderato. Inoltre, essendo pubblico il codice sorgente dell'applicazione, è possibile creare dei test molto precisi ed accurati per lo sviluppo del progetto.

Successivamente, la fase di progettazione della piattaforma sarà focalizzata sul disegnare ed implementare un livello di osservabilità, basato esclusivamente su tecnologie open source, in grado di raccogliere dati, correlarli e mostrarli in un modo significativo e chiaro attraverso l'uso di rappresentazioni grafiche all'interno dello stesso prodotto software. Le tipologie di dati principalmente utilizzate saranno *logs*, *metriche* tecnologiche e *tracciature* di navigazione degli utenti.

3.1 Applicazione a microservizi Online Boutique

Online Boutique è un'applicazione demo a microservizi nativa per cloud, sviluppata direttamente da Google, ed utilizzata per dimostrare le potenzialità di uno sviluppo software basato su un'architettura a microservizi e l'uso di nuove tecnologie come Kubernetes, Istio, Stackdirver e gRPC. Online Boutique è composta da 11 microservizi ed è progettata per essere installata facilmente senza nessun tipo di configurazione.

Si presenta all'utente come applicazione web di un e-commerce, dove gli utenti possono navigare tra gli oggetti in vendita, aggiungerli al carrello e infine comprarli. Essendo una demo, non sono state sviluppate funzioni per gestire il login/logout dell'utente e le impostazioni di pagamento.

Gli 11 microservizi dell'applicazione sono scritti in diversi linguaggi di programmazione, e comunicano tra loro tramite protocollo gRPC:

- **frontend** [Go]: espone un server HTTP per gestire le richieste del sito web. Non è richiesto un signup/login e genera automaticamente degli ID sessioni per tutti gli utenti.
- **cartservice** [C]: memorizza gli oggetti dell'utente del carrello in Redis e li recupera.
- **productcatalogservice** [Go]: fornisce la liste dei prodotti letti da un file JSON, la possibilità di cercarli e di mostrarli individualmente.
- **currencyservice** [Node.js]: converte le cifre in denaro in altre valute. Usa le reali quotazioni di valute offerte dalla Banca Centrale Europea.

- **paymentservice** [Node.js]: addebita la spesa sulla carta di credito fornita (fittizia) con l'importo specificato e restituisce un ID transazione.
- **shippingservice** [Go]: fornisce una stima dei costi di spedizione in base agli oggetti nel carrello. Spedisce gli articoli all'indirizzo indicato (fittizio).
- **emailservice** [Python]: invia all'utente la conferma dell'ordine via mail (fittizio).
- **checkoutservice** [Go]: elabora il carrello dell'utente, prepara l'ordine e orchestra il pagamento, la spedizione e la notifica e-mail.
- **recommendationservice** [Python]: suggerisce altri oggetti all'utente in base ai prodotto inseriti nel carrello.
- **adservice** [Java]: fornisce pubblicità testuale mostrata durante la navigazione web.
- **loadgenerator** [Python/Locust]: invia continuamente richieste a frontend, imitando flussi di acquisto realistici degli utenti.

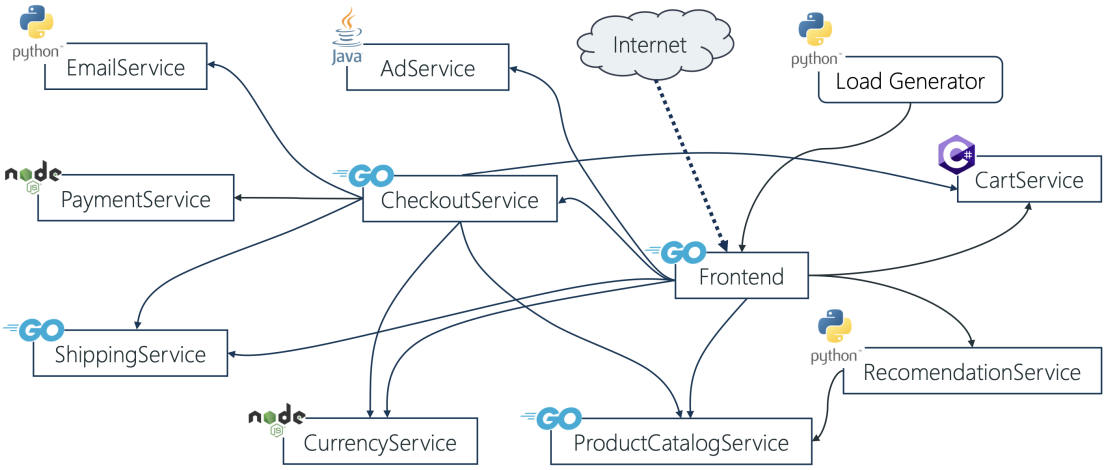

Figura 3.1: Struttura di Online Boutique

Online Boutique è stata disegna per girare su Kubernetes e grazie all'integrazione di Skaffold può essere installata lanciando un solo comando. **Skaffold** è un tool di linea di comando che facilita l'installazione per le applicazioni native Kubernetes. Gestisce infatti autonomamente l'intero flusso di azioni per compilare, installare ed eseguire tutti i microservizi come pods all'interno del cluster.

3.2 Difficoltà del monitoraggio

Le difficoltà nel comprendere la struttura e il funzionamento delle applicazioni a microservizi sono dovute proprio alla loro natura distribuita. Essendo composte da tanti blocchi tra loro collegati ed integrando una struttura modulare, studiare il codice sorgente risulta estremamente complesso. Per questo motivo, è stato necessario leggere attentamente la documentazione dell'applicazione per capire tutte le sue funzionalità e le comunicazioni che avvengono all'interno.

Successivamente si è passati allo studio del codice sorgente, identificando da subito le funzioni e i punti di estremità delle comunicazioni attraverso il protocollo gRPC. Le porzioni di codice relative alla comunicazione tra microservizi sono estremamente importanti e delicate poiché garantiscono il corretto funzionamento dell'applicazione. Inoltre, in presenza di problemi nella trasmissione dei messaggi, devono saper gestire l'errore, isolando ed evitando il blocco completo del servizio.

Un'altra difficoltà nel lavorare con architetture a microservizi è sicuramente l'uso di più linguaggi di programmazione nello sviluppo software. Sebbene così sia possibile garantire maggiore flessibilità ai team di sviluppo, che possono usare il linguaggio di programmazione più indicato per integrare le funzionalità del servizio, per una singola persona rappresenta sicuramente una grossa difficoltà, poiché raramente un singolo sviluppatore possiede alti livelli di programmazione per tanti linguaggi diversi. Le aziende, infatti, per lo sviluppo di applicazioni molto complesse, preferisco creare dei team di sviluppatori specializzati che lavorino esclusivamente con un solo linguaggio di programmazione.

I microservizi di Online Boutique sono scritti in 5 linguaggi di programmazione diversi. Prima di analizzare il codice sorgente, è stato necessario infatti recuperare quelle nozioni di teoria dei linguaggi in cui si è meno pratici e studiare la sintassi di quelli nuovi. Solo così è stato possibile comprendere ogni singola funzione di Online Boutique ed avere un'idea dettagliata del suo funzionamento.

3.3 Analisi e disegno della strategia di monitoraggio

Una volta sviluppato il sistema che si vuole monitorare, si passa alla progettazione della piattaforma di osservabilità. Lo sviluppo della piattaforma è stato diviso in due fasi distinte: la prima di analisi e disegno della strategia da seguire, la seconda di sviluppo pratico della piattaforma di osservabilità.

Gli strumenti di analisi e di disegno della strategia includono tecniche e tools di whitebox monitoring, con lo scopo di studiare dall'interno i processi di elaborazione

di Online Boutique. È importante quindi, controllare costantemente che tutti i suoi microservizi siano in esecuzione correttamente e che la comunicazione tra loro sia priva di errori. Per fare ciò, si è scelto di rendere osservabile l'applicazione Online Boutique, con l'obiettivo di generare e raccogliere logs, metriche e tracciature, successivamente analizzate e studiate.

La strategia ideata è composta da quattro fasi distinte, completamente collegate tra loro, che garantiscono un alto livello di osservabilità del sistema ed un monitoraggio che riassume il suo stato interno attraverso rappresentazioni grafiche.

La prima fase è quella di *instrumentazione e generazione*. Studiando i principi del whitebox monitoring, si rende osservabile l'applicazione Online Boutique usando librerie open source, che generano logs, metriche e tracciature e le espongono all'esterno.

Segue la seconda fase di *acquisizione ed elaborazione*. All'interno del cluster, si installano diversi tools, ognuno specializzato per un singolo pilastro dell'osservabilità, che acquisiscono ed elaborano le informazioni esposte da Online Boutique.

La terza fase è di *memorizzazione*, poiché a seguito dell'elaborazione delle informazioni raccolte, sono generati dati estremamente importanti per il monitoraggio dello stato interno del sistema.

Infine, la strategia termina con la fase di *visualizzazione*. Attraverso la configurazione di applicazioni specializzate nello studio dell'osservabilità, con l'analisi dei dati conservati si generano delle dashboards personalizzate che riassumono lo stato interno di Online Boutique.

3.3.1 Generazione

In seguito allo studio dell'osservabilità e di come si possa garantirla all'interno dei sistemi distribuiti, è chiaro che ancor prima di installare e configurare tutte le applicazioni che compongono la piattaforma di osservabilità, sia necessario istrumentare il sistema che si vuole monitorare, in modo tale da ottenere maggiori informazioni sul suo stato interno.

Nel caso di Online Boutique, si instrumenta manualmente il suo codice sorgente con delle funzioni che generano direttamente dall'interno dell'applicazione logs, metriche e tracciature.

I logs sono uno strumento già supportato dai principali linguaggi di programmazione, che offrono diverse librerie standard per la loro generazione ed esportazione. Sono infatti utilizzati già da molto prima dell'avvento dell'osservabilità all'interno delle infrastrutture informatiche. Pertanto, analizzando il codice di Online Boutique si identificano subito, in tutti i microservizi, quelle righe dove sono definiti i logs ed esportati attraverso la stampa a video, in modo tale da essere letti dagli sviluppatori.

Metriche e tracciature, invece, sono due strumenti molto legati al tema recente dell'osservabilità e per questo motivo, non sono ancora integrati nelle librerie dei linguaggi di programmazione. Per poter instrumentare Online Boutique anche con metriche e tracciature, è necessario aggiungere all'applicazione delle librerie open source sviluppate da gruppi indipendenti della comunità informatica. Sono tanti, infatti, i progetti compatibili con i più comuni linguaggi di programmazione che offrono molte funzioni utili per garantire l'osservabilità all'interno dei sistemi.

Una volta instrumentato il codice per la generazione dei tre pilastri dell'osservabilità, è importante aggiungere delle funzioni che rendano possibile la lettura delle informazioni prodotte dall'esterno dell'applicazione. Per i logs, essendo uno strumento molto diffuso e compatibile anche con alcune funzioni di Kubernetes, è preferibile lasciare la stampa a video già integrata. Metriche e tracciature, invece, devono essere esportate attraverso una comunicazione diretta con delle applicazioni esterne compatibili, in grado di acquisirle correttamente.

3.3.2 Acquisizione ed elaborazione

Instrumentata correttamente, Online Boutique a seguito di input esterni attraverso la sua interfaccia web, oltre ad elaborare le richieste ricevute, inizia a generare logs, metriche e tracciature. Non essendo ancora possibile leggere queste informazioni, è necessario installare nel cluster delle applicazioni che siano in grado di catturarle e che siano compatibili con il loro formato così da analizzarle con successo.

Inoltre, poiché Online Boutique è un'applicazione a microservizi eseguita in un ambiente Kubernetes, è importante raccogliere tutte le informazioni dai singoli componenti, per poi unirle coerentemente in un unico punto, così da avere una visuale globale su tutta l'applicazione.

Data la vasta gamma di strumenti open source disponibili che lavorano nel campo del whitebox monitoring, si è scelto di sviluppare tre strategie distinte di acquisizione ed elaborazione, ognuna specifica per un pilastro dell'osservabilità. Questa scelta offre un buon livello di flessibilità per lo sviluppo della piattaforma di monitoraggio, in quanto gli strumenti di una strategia sono completamente indipendenti da quelli usati delle altre. In un contesto lavorativo più complesso, adottare questo approccio risulta estremamente utile, poiché permette di creare team dedicati ad una singola strategia per che possono lavorare parallelamente, accelerando lo sviluppo del progetto.

Questa divisione in tre strategie distinte, permette anche di scegliere strumenti altamente specifici per il pilastro dell'osservabilità studiato. Nello sviluppo della piattaforma, infatti, sono presenti tre insiemi distinsi di tools, ognuno che opera esclusivamente o con i logs, o con le metriche o con le tracciature.

Inoltre, sebbene le tre strategie disegnate implementino processi lavorativi differenti, l'obiettivo finale di ognuna è sempre quello di generare dati utili per rappresentare graficamente lo stato interno del sistema.

I logs essendo esportati all'esterno di Online Boutique attraverso la stampa a video, è opportuno acquisirli con un'applicazione DaemonSet installata all'interno del cluster. Il **DaemonSet** è uno strumento di Kubernetes che gestisce le repliche dei Pods, il cui compito è quello di assicurarsi che un determinato pod sia eseguito in tutti i nodi del cluster. Intanto che i nodi vengono aggiunti al cluster, vengono aggiunti dei pod. Quando i nodi vengono rimossi dal cluster, tali pod vengono eliminati.

Dopo la configurazione del DaemonSet, che assicura una corretta acquisizione dei logs stampati a video dai microservizi, è necessario aggregare tutti i dati in un singolo punto. L'applicazione DaemonSet deve quindi inoltrare le informazioni catturate ad un una nuova applicazione specifica per la loro elaborazione, che subito unificherà coerentemente tutti i dati ricevuti.

Il compito principale di questa applicazione è quello di filtrare i logs realmente necessari per monitorare lo stato del sistema, scartando tutte le informazioni superflue. Inoltre deve manipolare i logs attraverso delle funzioni di trasformazione, aggiungendo nuovi campi di informazioni utili per il futuro studio di Online Boutique. È compito del team di sviluppo configurare correttamente questa applicazione, in modo tale che esegua i processi di elaborazione necessari per ottenere il tipo di dato desiderato.

Per l'acquisizione e l'elaborazione di metriche e tracciature, invece, si è optato per seguire una strategia diversa rispetto a quella dei logs. Poiché sono disponibili parecchie applicazioni open source che offrono un alto livello di elaborazione per questi due pilastri dell'osservabilità, si è preferito integrare alla piattaforma di monitoraggio degli strumenti molto più completi.

Si implementa, quindi, una fase di acquisizione delle informazioni progettata esclusivamente su una comunicazione diretta tra Online Boutique e le due nuove applicazioni dedicate all'elaborazione di metriche e tracciature. Così facendo, si ottimizzano i tempi di esecuzione, evitando l'installazione di più strumenti.

Tra le funzionalità offerte dalle applicazioni open source più complete, sono presenti i processi ottimizzati per l'elaborazione dei dati e anche una semplice interfaccia grafica, accessibile tramite browser. Sebbene queste interfacce non siano necessarie ai fini dello sviluppo della piattaforma di osservabilità, sono uno strumento molto utile in fase di progettazione, in quanto offrono un supporto immediato in modo accessibile. Infatti, ci permettono di verificare subito che metriche e tracciature arrivino correttamente, e che tutte le configurazioni predisposte precedentemente producano il risultato desiderato, evitando di scoprire eventuali

errori al termine dell'installazione della piattaforma.

3.3.3 Memorizzazione

I dati prodotti in seguito alla fase di acquisizione ed elaborazione, contengono tutte le informazioni necessarie che, grazie ad una analisi approfondita, permettono di comprendere lo stato interno dell'applicazione Online Boutique. Sono decisamente importanti per garantire il corretto monitoraggio e di conseguenza devono essere archiviati in delle strutture a loro dedicate.

Tuttavia, cercando applicazioni che siano in grado di offrire strumenti per la memorizzazione di logs, metriche e tracciature, non è stato trovato un servizio che permettesse l'archiviazione unificata di questi tre tipi di dato. Purtroppo infatti, ogni pilastro viene elaborato in un formato che, a causa delle diverse informazioni contenute, risulta completamente diverso da quello degli altri. La differenza di sintassi dei dati elaborati, spesso proprietaria dell'applicazione che li ha generati, non permette quindi di aggregarli tutti insieme. La soluzione è stata implementata installando applicazioni con strutture dedicate al singolo pilastro e separando quindi tra loro i dati di logs, metriche e tracciature.

Un'avanzata piattaforma di osservabilità non si limita a descrivere lo stato interno del sistema in tempo reale, ma è in grado di analizzare sopratutto lo stato passato. In questo modo, attraverso dei confronti tra il comportamento del presente ed uno simile del passato, è possibile prevedere probabili errori futuri. Inoltre, essendo quest'analisi fatta sulle informazioni offerte dai dati elaborati, è chiaro quanto quest'ultimi risultino di assoluta importanza per il monitoraggio del sistema.

Le applicazioni specializzate nell'acquisizione ed elaborazione di logs, metriche e tracciature non implementano strutture dedicate alla memorizzazione. I dati, infatti, sono conservati temporaneamente nella RAM della macchina, e con un semplice riavvio dell'intera piattaforma si otterrebbe la perdita di tutte le informazioni.

È necessario integrare delle strutture di memorizzazione a lungo termine, rendendo affidabile l'intera piattaforma di monitoraggio. Per fare ciò, si affianca ad ogni applicazione di elaborazione un tool specializzato nella conservazione di dati e capace di sfruttare tecniche ad alta affidabilità, in grado di evitare perdite di informazioni in caso di eventuali errori o malfunzionamenti.

3.3.4 Rappresentazione

L'ultima fase dello sviluppo della piattaforma di osservabilità è quella di rappresentazione di tutte le informazioni raccolte, elaborate e memorizzate. Tuttavia, i dati conservati, essendo completamente in formato di testo, non sono molto comprensibili per un operatore umano. È necessario, quindi, usare un'applicazione

che sia in grado di capirli e successivamente di creare delle rappresentazioni grafiche che semplifichino la comprensione dello stato interno per il team di sviluppo. Inoltre, l'applicazione deve essere necessariamente compatibile con le strutture di memorizzazione dove sono archiviati i dati dei tre pilastri, così da poter estrarre tutte informazioni necessarie.

Le rappresentazioni grafiche non sono le stesse per logs, metriche e tracciature poiché ognuna mostra informazioni differenti sullo stato interno del sistema, e il singolo pilastro deve essere visualizzato con la rappresentazione più adatta ad esso.

Per i logs, è sufficiente uno strumento di visualizzazione con filtri per microservizio, namespace, pod o nodo del cluster. Così, andare a studiare gli eventi di un singolo processo risulta molto più rapido, grazie anche alla possibilità di analizzare tutta l'applicazione Online Boutique da un unico strumento.

Le metriche, invece, essendo composte da numeri, si approcciano bene a rappresentazioni grafiche come diagrammi, grafici nel tempo e valori numerici. Permettono uno studio molto ampio dello stato del sistema, grazie sopratutto alla possibilità di trasformare i loro dati con modelli matematici, calcolando medie, differenze, valori massimo e minimo, ecc.

Infine le tracciature offrono una rappresentazione grafica standard, con elenco degli spans che la compongono e i relativi tempi di elaborazione. Un supporto per il team di sviluppo è quello di poter filtrare le tracciature per microservizio e di avere in evidenza le tracciature con errori all'interno.

Una funzionalità molto utile per il monitoraggio del sistema è la possibilità di poter impostare degli avvisi che scattano in presenza di determinate condizioni, per esempio un valore di una metrica che supera una soglia precedentemente impostata. Si semplifica così il lavoro dell'operatore, che dovrà solo intervenire ad allarme scattato.

La rappresentazione grafica è la fase finale dello sviluppo della piattaforma, attraverso la quale è possibile monitorare l'intero sistema. È importante, quindi, che includa molti elementi grafici personalizzabili, offrendo la miglior soluzione per qualsiasi tipo di analisi necessaria.

3.4 Strumenti per il monitoraggio del sistema

Completato il disegno della strategia per lo sviluppo della piattaforma di osservabilità, si passa alla ricerca e alla selezione di tutto l'insieme di applicazioni e strumenti più adatti per raggiungere l'obiettivo del progetto. Grazie alla letteratura

informatica nel campo dell'osservabilità e all'analisi delle specifiche tecniche delle diverse applicazioni, è stato possibile confrontare ed individuare gli strumenti open source migliori tra tutti quelli offerti dalla comunità del settore del monitoraggio.

Per ogni singolo pilastro, la strategia disegnata identifica quattro fasi di sviluppo distinte, collegate tra loro. In ognuna di queste quindi, è necessario installare e configurare nel cluster una o più applicazioni, così da garantire la corretta esecuzione per poter passare alla fase successiva.

Pertanto sono analizzate le applicazioni candidate per sviluppare la piattaforma di osservabilità, studiando singolarmente le fasi dei distinti monitoraggi di logs, metriche e tracciature. Si mostrano i vantaggi e svantaggi di ogni strumento preso in considerazione, per poi motivare attraverso un confronto diretto la scelta del servizio che meglio si integra allo sviluppo del progetto, garantendo il raggiungimento degli obiettivi desiderati.

3.4.1 Monitoraggio dei logs

In questo paragrafo si studiano le applicazioni più usate per il monitoraggio dei logs, distinguendo le quattro fasi del flusso di elaborazione della strategia disegnata. Nel caso in cui ci fossero più applicazioni candidate per l'esecuzione di una fase, sono mostrati i punti di forza e le funzionalità di ognuna, per poi confrontarle e individuare lo strumento più adatto da implementare nella piattaforma.

Generazione

I microservizi che sono stati scelti per il monitoraggio dell'applicazione Online Boutique, sono scritti in linguaggi di programmazione molto diffusi. Le librerie standard di Java, Golang e NodeJs definiscono diverse classi di logs e offrono molti metodi per generarli e gestirli. Infatti, analizzando il codice di Online Boutique si trovano già diverse righe di codice che definiscono logs per i processi di debug, usati dal team di sviluppo per verificare il corretto funzionamento dell'applicazione.

Tutti questi logs sono esportati all'esterno di Online Boutique attraverso la stampa a video effettuata dal pod che esegue il microservizio. Infatti, all'interno di Kubernetes lanciando il comando bash "*kubectl logs*" seguito dal nome del pod e dal namespaces dove quest'ultimo è in esecuzione, si possono visualizzare tutti i logs stampati dal microservizio.

Essendo quindi, già garantita la generazione all'interno dei microservizi, non è necessario instrumentare il codice sorgente con nuove librerie esterne.

Acquisizione ed elaborazione

Una volta che i logs sono stati stampati a video, è necessario installare degli strumenti che siano in grado di catturarli, per poi successivamente elaborarli. In seguito alla ricerca e allo studio delle applicazioni etichettate per la gestione dei logs, sono stati individuati tre tools molto popolari tra la comunità informatica dell'osservabilità: FluentD, Logstash e FluentBit.

FluentD è un data collector open source, che permette di unificare la collezione e l'elaborazione dei dati per un miglior uso e comprensione delle informazioni. In particolare, le sue funzioni chiave sono:

- *Unificazione dei logs in JSON*: FluentD prova a costruire le nuove strutture dati in formato JSON il più possibile: questo permette a FluentD di unificare tutti i diversi aspetti dell'elaborazione dei logs: raccolta, filtraggio, buffering e inoltrare i logs tra più sorgenti e destinazioni. Il flusso dell'elaborazione dei dati è molto più facile in JSON, dal momento che questo offre una struttura accessibile pur mantenendo schemi flessibili.
- *Architettura a plugin*: FluentD ha un sistema flessibile a plugins che permette alla comunità di estendere le sue funzionalità. Al momento ci sono più di 500 plugins disponibili.
- *Minime risorse richieste*: FluentD è scritto in C e Ruby, e richiede un consumo veramente minimo di risorse. Una singola istanza pesa circa 40 MB di memoria e può processare circa 13,000 eventi/secondo/core.
- *Affidabilità integrata*: FluentD supporta il buffering basato su file e memoria per prevenire la perdita di dati tra i nodi. FluentD implementa anche un failover robusto e può essere configurato per l'alta affidabilità.

FluentD può essere configurato con più sorgenti da cui ricevere i logs, che nel momento stesso in cui sono letti dall'applicazione, vengono etichettati. La gestione delle etichette permette a FluentD di instradare i logs verso plugins distinti. L'instradamento, e di conseguenza il percorso di elaborazione a cui sono sottoposti i logs, sono inerenti al modo in cui è stato configurato FluentD. In Kubernetes, questo comportamento è definito in una ConfigMap, che può essere modificata secondo le necessità in qualsiasi momento.

Logstash è uno dei tre progetti open source alla base dell' Elastic Stack, un piattaforma che offre la possibilità di aggregare e studiare i logs catturati dal sistema e dai servizi installati, e di creare rappresentazioni grafiche per l'analisi delle applicazioni, per controlli di sicurezza e altro. Elastic Stack, o ELK Stack,

include tre progetti diventati popolari tra la comunità informatica: Elasticsearch, Logstash e Kibana.

Logstash è un motore open source di raccolta dati con funzionalità di elaborazione a catena di montaggio in tempo reale. Logstash non è un semplice aggregatore di logs. Può infatti unificare diversi tipi di dati da più sorgenti simultaneamente, e normalizzarli in destinazioni scelte dal team di sviluppo. Logstash è compatibile con tanti formati di logs come log4j per Java, syslog, logs di rete e firewall.

I dati, una volta entrati nell'applicazione, possono essere arricchiti e trasformati grazie ad un'ampia gamma di plugins di input, filtri e output. Il team di sviluppo può quindi configurare Logstash per semplificare e trasformare i dati durante l'importazione, così da avere le informazioni più importanti direttamente in tempo reale.

FluentBit è un'applicazione specializzata nell'estrarre ed inoltrare i logs permettendo di collezionare queste informazioni, arricchendole con filtri e inoltrandole a più destinazioni.

FluentBit è pensata e sviluppata per essere eseguita all'interno dell'ambiente Kubernetes, grazie alla sua struttura estremamente leggera perfetta per lavorare con i container: garantisce infatti alte prestazioni con un bassissimo consumo di CPU e memoria. È scritta in C ed è progettata secondo un'architettura a plugins, che implementa al momento più di 70 estensioni di input, filtri e output.

Grazie ai suoi strumenti integrati, FluentBit è in grado di comprendere i dati non strutturati in entrata e di convertirli immediatamente in formati più semplici da processare come JSON, Regex, LTSV e Logfmt. Inoltre, memorizza temporaneamente i dati in memoria e sul file system fin quando non sono inoltrati in output. Così facendo, nessun logs viene perso in caso di malfunzionamenti di rete o di sistema.

Confrontando le tre applicazioni candidate, è evidente che FluentBit non offre un insieme di funzionalità al pari di FluentD e Logstash. In un sistema di media-alta complessità, questo vincolo limita l'analisi dei logs, non garantendo un monitoraggio di alto livello. FluentD e Logstash, invece, grazie alla vasta gamma di funzionalità implementate, offrono una fase di elaborazione flessibile, permettendo un monitoraggio dettagliato dello stato interno del sistema. Tuttavia, installare entrambe le applicazioni all'interno del cluster è una scelta ridondante, poiché il loro comportamento è molto simile.

Analizzando i vantaggi e svantaggi di FluentD e Logstash, il primo risulta essere lo strumento più indicato per lo sviluppo della piattaforma di osservabilità. FluentD, infatti, offre al team di sviluppo un insieme di plugins molto più numeroso di quello di Logstash, così da permettere ogni tipo di elaborazione e trasformazione dei logs. Inoltre, la sua architettura è perfettamente compatibile con Kubernetes e

la possibilità di salvare i logs localmente mentre la rete non funziona, consente di non perdere nessun tipo di informazione.

Logstash è un'applicazione più pesante rispetto a FluentD e non implementa una struttura ad alta affidabilità. Il carico lavorativo del team di sviluppo, quindi, sarà maggiore poiché si dovrà costantemente controllare la sua corretta esecuzione. Inoltre, le funzionalità di Logstash non sono completamente indipendenti ed isolate, ma sono molto legate a quelle degli altri progetti dell'ELK Stack, essendo le tre applicazioni sviluppate dallo stesso fornitore.

Poichè FluentBit e FluentD usano entrambi il formato JSON come struttura dei logs, per incrementare le performance della piattaforma di osservabilità si è scelto di implementare l'acquisizione dei logs dai microservizi con FluentBit, che inoltra tutte le informazioni a FluentD per la loro elaborazione. Questa soluzione permette di sfruttare i principali vantaggi delle due applicazioni per ottimizzare le risorse del sistema. FluentBit offre una fase di acquisizione molto più veloce rispetto a FluentD, garantendo una lettura ed inoltro dei logs a basso consumo, FluentD invece assume il ruolo di aggregatore ed elaboratore, che grazie alle centinaia di plugins offerti semplifica il processo di trasformazione dei logs agli sviluppatori.

Memorizzazione

Una volta filtrati ed elaborati tutti i logs utili per il monitoraggio dello stato interno di Online Boutique, è necessario memorizzarli localmente, così che sia possibile per le applicazioni di rappresentazione grafica, estrarre tutte le informazioni necessarie. Tuttavia, FluentD non offre nessuna funzionalità di memorizzazione e non può essere configurato come un sorgente di dati. È necessario quindi, conservare i logs in strutture disegnata specificatamente per la loro conservazione ed aggregazione. Gli strumenti più completi e popolari tra la comunità informatica, che garantisco la corretta memorizzazione dei logs sono Elasticsearch e Grafana Loki.

Elasticsearch è una delle tre applicazioni che compone l'ELK Stack ed è un motore distribuito, open source e analitico per tutti i tipi di dati, inclusi quelli testuali, numerici, geospaziali, strutturati e non-strutturati. Elasticsearch è sviluppata in Apache Lucene ed è diventata molto popolare tra la comunità, grazie alle sue semplici REST APIs, alla sua natura distribuita, e alla sua velocità e scalabilità.

Elasticsearch memorizza tutte le informazioni come documento JSON, ognuno associato ad un insieme di chiavi (nomi dei campi o proprietà) con i loro corrispondenti valori (stringhe, numeri, booleani, dati, vettori di valori, geolocalizzazioni, o altri tipi di dato). I logs in entrata sono prima soggetti ad un processo definito *data*

ingestion, durante il quale le informazioni sono parsificate, normalizzate ed arricchite, per poi successivamente essere completamente indicizzate in Elasticsearch. Così, gli utenti possono lanciare query complesse per interrogare i loro dati e utilizzare le aggregazioni per recuperare riepiloghi complessi delle informazioni memorizzate.

Grafana Loki è un sistema di aggregazione dei logs che offre scalabilità orizzontale, alta affidabilità e un'architettura multi-tenant. È stato disegnato per essere molto conveniente e facile da usare. Non indicizza il contenuto dei logs, ma piuttosto affianca un insieme di etichette a ciascuno di essi. Così, memorizzando i logs compressi e non-strutturati, ed indicizzando solo i metadati, Loki è più semplice da configurare e più veloce nell'elaborazione. Inoltre, la struttura progettata di Loki, lo rende perfetto per memorizzare i logs stampati dai pods di Kubernetes. Infatti, i metadati come le etichette dei pods sono automaticamente identificati ed indicizzati.

Data la perfetta integrazione con Kubernetes, Loki risulta essere la scelta migliore tra i due strumenti analizzati. Infatti, memorizzare i logs come testo in chiaro, taggato con un insieme indicizzato di etichette di nomi e valori, offre prestazioni più elevate rispetto ad un indice completo di tutti i dati, implementato invece da Elasticsearch.

Visualizzazione

L'ultima fase della strategia di monitoraggio, è quella di visualizzare da una singola applicazione, tutti logs necessari allo studio dello stato interno di Online Boutique, aggregati uniformemente. Per lo sviluppo della piattaforma di osservabilità, è necessario che questa stessa applicazione, offra anche rappresentazioni grafiche per le successive analisi delle metriche e tracciature. Gli strumenti principali offerti dalla comunità informatica, specializzati nella visualizzazione di logs, metriche e tracciature sono Kibana e Grafana.

Kibana è un'applicazione frontend gratuita ed open source che si trova in cima all'Elastic Stack. Comunemente noto come lo strumento di creazione di grafici per l'Elastic Stack, Kibana funge anche da interfaccia utente per il monitoraggio, e per la gestione e la protezione di un cluster ELK.

La stretta integrazione di Kibana con Elasticsearch lo rendono ideale per la ricerca e visualizzazione dei dati indicizzati in Elasticsearch e per l'analisi dei dati tramite la creazione di grafici a barre, grafici a torta, tabelle, istogrammi e mappe. Infatti, si possono disegnare delle dashboard combinando questi elementi visivi, per poi essere condivise tramite browser fornendo visualizzazioni analitiche in tempo reale, che mostrano l'analisi dei logs, il monitoraggio delle metriche e di container, e le tracciature delle applicazioni installate.

Kibana fornisce anche uno strumento di presentazione, denominato Canvas, che consente agli utenti di creare presentazioni di diapositive che estraggono dati in tempo reale direttamente da Elasticsearch.

Grafana è la tecnologia più popolare usata per la creazione di dashboard di osservabilità e monitoraggio. Attraverso un'interfacia web, Grafana consente di interrogare, visualizzare, avvisare e comprendere le metriche, i logs e le tracciature, indipendentemente da dove sono archiviate.

La creazione dei grafici lato client risulta veloce e flessibile, grazie ad una moltitudine di opzioni disponibili e ai diversi plugins dei panelli che offrono molti modi diversi per visualizzare i dati. Le dashboard sono completamente dinamiche e riutilizzabili con variabili del sistema che possono essere selezionate da un menù a discesa nella parte superiore della dashboard.

Inoltre, Grafana è compatibile con diversi tipi di database, rendendosi completamente indipendente dalle elaborazioni precedenti e offrendo la possibilità di unire diverse origini di dati nello stesso grafico.

Sebbene Kibana offra molti strumenti per il monitoraggio dello stato interno del sistema, la sua stretta dipendenza da Elasticsearch non lo rende una scelta implementabile per lo sviluppo della piattaforma di osservabilità. Grafana, invece, grazie alla compatibilità con i diversi tipi di sorgenti dati si integra perfettamente con tutte le applicazioni del sistema di monitoraggio disegnato.

3.4.2 Monitoraggio delle metriche

Terminato lo studio dei logs, si passa al monitoraggio delle metriche, analizzando le quattro fasi della strategia disegnata.

Generazione

Contrariamente alla generazione dei logs, i linguaggi di programmazione per lo sviluppo software non implementano nessuna libreria standard per la creazione delle metriche, limitando così l'osservabilità delle applicazioni. È necessario, quindi, instrumentare il codice sorgente dei microservizi selezionati di Online Boutique con librerie open source offerte e sviluppate da teams della comunità informatica. Il progetto più ambizioso nel campo dell'osservabilità delle applicazioni è OpenTelemetry.

OpenTelemetry è un insieme di strumenti, APIs e SDKs, e membro della Cloud Native Computing Foundation, un progetto della Linux Foundation per aiutare a far progredire la tecnologia dei container e allineare il settore tecnologico alla sua evoluzione. OpenTelemetry è nato come fusione dei due progetti open

source OpenCensus e OpenTracing, che avendo standard diversi obbligavano gli utenti a scegliere un solo ecosistema. Inoltre, sebbene i fornitori di tecnologia abbiano creato agenti per raccogliere i dati di telemetria, l'utilizzo di questi agenti può vincolare gli sviluppatori a tali fornitori. OpenTelemetry definisce un unico standard open source per instrumentare e generare le metriche e tracciature, e offre la tecnologia per raccogliere ed esportare i dati di telemetria dalle applicazioni cloud-native per poterli monitorare e analizzare.

OpenTelemetry definisce un singolo set di APIs, specifiche per il linguaggio di programmazione, che acquisiscono i dati da framework web, client di archiviazione e sistemi RPC. Possono essere utilizzate per creare metriche personalizzate e tracciature. Inoltre, semplifica la gestione e l'esportazione dei dati di telemetria acquisiti, configurando specifici SDK per inviare le informazioni verso agenti collettori esterni come Prometheus, Zipkin o Jaeger.

Acquisizione ed elaborazione

Per esportare le metriche generate, fuori da Online Boutique, è necessario dichiarare all'interno del codice sorgente il collettore verso il quale inoltrare tutte le informazioni necessarie, incaricato di svolgere la fase di acquisizione ed elaborazione delle metriche, aggregandole uniformemente secondo il loro formato. La dichiarazione del collettore viene implementata all'interno dei singoli microservizi, grazie ai SDKs offerti da OpenTelemetry. Per l'elaborazione delle metriche, OpenTelemetry è compatibile con un solo agente collettore, Prometheus.

Prometheus è un toolkit open source di monitoraggio e allarmistica dei sistemi, originariamente sviluppato da SoundCloud. Sin dagli inizi (2012), molte aziende e organizzazioni hanno adottato Prometheus, costituendo al giorno d'oggi una grande comunità di sviluppatori e utenti molto attiva. Ora è un progetto open source autonomo e gestito indipendentemente da qualsiasi azienda. Per enfatizzare questa peculiarità e chiarire la struttura di *governance* del progetto, Prometheus è entrato a far parte della Cloud Native Computing Foundation nel 2016 come secondo progetto selezionato, dopo Kubernetes.

Prometheus implementa un modello dati multi-dimensionale con dati di serie temporali identificati dal nome della metrica e da un insieme di coppie chiave-valore, usato per arricchire le metriche con informazioni aggiuntive. Inoltre, l'acquisizione delle metriche segue un approccio *pull* attraverso protocollo HTTP. Infatti, è Prometheus che, assumendo il ruolo di *client*, interroga i microservizi per estrarre le metriche generate. I servizi configurati per essere letti da Promethues sono definiti *target*, e fungendo da *server*, permettono a Prometheus di acquisire correttamente i dati necessari per il monitoraggio. Per garantire la corretta acquisizione, Prometheus

implementa una funzionalità automatica di scoperta dei servizi target, in alternativa alla configurazione statica.

Inoltre, Prometheus mette a disposizione degli sviluppatori una propria interfaccia web, grazie alla quale è possibile monitorare il sistema analizzando i grafici delle metriche acquisite. Per consentire agli utenti di selezionare e aggregare i dati delle metriche in tempo reale, Promethues fornisce un linguaggio di query funzionale chiamato PromQL (Prometheus Query Language). PromQL integra diversi modelli e funzioni matematiche per la trasformazione dei dati, offrendo una manipolazione delle metriche ad alto livello.

Memorizzazione

Per la fase di memorizzazione delle metriche, non è necessaria l'installazione di altri software nel cluster. Prometheus, infatti, include un database locale su disco delle metriche acquisite, permettendo ad altre applicazioni di estrarre le informazioni archiviate attraverso le query PromQL.

Il database delle serie temporali locali di Prometheus memorizza i dati in un formato personalizzato e altamente efficiente nella memoria locale.

I campioni acquisiti sono raggruppati in blocchi di due ore. Ogni blocco di due ore è costituito da una directory contenente uno o più file di blocchi che contengono tutti i campioni di serie temporali per quella finestra di tempo, nonché un file di metadati e un file di indice (che indicizza i nomi e le etichette delle metriche in serie temporali all'interno dei file di blocchi).

Il blocco corrente per i campioni in ingresso è mantenuto in memoria e non è completamente persistente. È protetto dagli arresti anomali grazie al protocollo Write Ahead Log (WAL), ovvero le modifiche sono registrate in file di logs, prima che vengano scritte nel database. Così, al riavvio del server Prometheus può riprodurre tutte le azioni. I file di log write-ahead sono archiviati nella directory *wal* in segmenti da 128 MB. Questi file contengono dati grezzi che non sono stati ancora compattati; quindi sono significativamente più grandi dei normali file a blocchi. Prometheus conserverà un minimo di tre file di log write-ahead.

Si noti che una limitazione dell'archiviazione locale è che non è clusterizzata o replicata. Pertanto, non è arbitrariamente scalabile o durevole di fronte a interruzioni di unità o nodi e dovrebbe essere gestita come qualsiasi altro database a nodo singolo.

Visualizzazione

Come per i logs, lo strumento della piattaforma di osservabilità usato per l'analisi delle metriche è Grafana. Grafana, infatti, supporta Prometheus come sorgente di

dati ed estrae le serie temporali con l'uso di query PromQL.

Per il monitoraggio e l'analisi di metriche, Grafana offre tantissime tipologie di pannelli grafici, garantendo il miglior tipo di monitoraggio per le diverse componenti del sistema.

3.4.3 Monitoraggio delle tracciature

Ultima e terza fase per concludere lo sviluppo della piattaforma di osservabilità, è il monitoraggio delle tracciature. Si analizzano le applicazioni e gli strumenti necessari per i quattro processi di generazione, acquisizione ed elaborazione, memorizzazione, visualizzazione.

Generazione

Come per le metriche, le librerie standard dei linguaggi di programmazione non implementano alcun tipo di supporto per le tracciature. Si estende quindi il codice sorgente di Online Boutique con le librerie di OpenTelemetry, che grazie all'insieme di APIs e SDKs offerti, permette di generare le tracciature all'interno dell'applicazione e di esportare tutte le informazioni ad un agente esterno.

Acquisizione ed elaborazione

Instrumentata Online Boutique, è necessario installare nel cluster un collettore di tracciature, che riceva dai diversi microservizi tutte le informazioni per il monitoraggio dell'applicazione. Sarà compito del collettore aggregare i dati raccolti per generare le tracciature, così da mostrare le comunicazioni avvenute tra i microservizi di Online Boutique.

OpenTelemetry offre il supporto a i due collettori di tracciature più popolari tra la comunità informatica, Zipkin e Jaeger. **Zipking** è stato sviluppato da Twitter e ora è mantenuto come progetto open source dalla sua comunità dedicata. **Jaeger**, invece, è stato sviluppato da Uber ed è stato condiviso come progetto open source una volta pronto. L'architettura generale delle due applicazioni è molto simile, anche se sfruttano APIs diverse. Entrambi ricevono le tracciature dal sistema instrumentato, registrano i dati e le correlazioni tra le diverse tracciature. Inoltre, offrono un'interfaccia grafica per permettere all'utente di analizzare le tracciature generate dai microservizi monitorati.

Zipkin è un'applicazione scritta in Java ed è stato uno dei primi sistemi di tracciamento sviluppato. Zipkin è eseguito come un singolo processo composto da diversi servizi, ognuno che implementa un insieme di funzionalità differenti. Sono presenti il collettore, il database, le API e l'interfaccia grafica. Questa struttura

rende l'installazione nel cluster semplice e rapida.

Jaeger è scritta in Go ed è molto simile a Zipkin, essendo un'applicazione più recente, ma presente comunque qualche differenza. In aggiunta alle funzionalità di Zipkin, Jaeger fornisce anche il campionamento dinamico, API Rest, un interfaccia grafica basata su ReactJS ed un supporto integrato per Elasticsearch e Cassandra come database. Per implementare tutte queste funzionalità, Jaeger usa un approccio più distribuito rispetto a Zipkin.

Jaeger, inoltre, è un progetto che fa parte del Cloud Native Computing Foundation, quindi è perfettamente sviluppato per girare su Kubernetes. Sono infatti offerti un modello Kubernetes ufficiale e un pacchetto Helm che distribuiscono l'agente, il collettore, l'API di query e l'interfaccia utente.

Per la piattaforma di osservabilità, si è preferito installare Jaeger. Tra i due, sicuramente Zipkin è un prodotto più maturo, essendo stato ideato prima, ma la sua struttura poco modulare e simile ad un'architettura centralizzata, lo rende più lento e meno flessibile rispetto a Jaeger. Questa differenza non incide molto nei sistemi piccoli, ma man mano che il sistema tende a crescere e a scalare, questo limite potrebbe diventare un problema. La comunità di Jaeger compensa la relativa mancanza di maturità di Jaeger fornendo una buona documentazione e una gamma di opzioni di implementazione.

Inoltre, la piattaforma di osservabilità, essendo sviluppata all'interno di un cluster Kubernetes con l'orchestrazione dei container Docker, risulta perfettamente compatibile con tutte le funzionalità di Jaeger, fornendo migliori prestazioni del sistema. Zipkin, infatti, si adatta meglio in un ambiente meno distribuito, poiché ci sono meno componenti in movimento.

Memorizzazione

Jaeger, ricevute le tracciature dai microservizi instrumentati, memorizza i dati su un suo semplice database memorizzato in RAM, utile per testare le configurazioni dell'agente. Inoltre, grazie al suo servizio di Query, permette ad altre applicazioni compatibili, di estrarre le informazioni archiviate per il monitoraggio delle tracciature raccolte.

Jaeger supporta due popolari database NoSQL open source come backend per l'archiviazione delle tracciature: Cassandra ed Elasticsearch. In aggiunta, sono in corso esperimenti della comunità per integrare altri database, come ScyllaDB, InfluxDB e Amazon DynamoDB.

Visualizzazione

La rappresentazione grafica delle tracciature è un'operazione fondamentale per il monitoraggio di un'applicazione a microservizi. Aiuta a comprendere il funzionamento interno dell'applicazione e mette in evidenza le comunicazioni avvenute tra i diversi componenti coinvolti nell'elaborazione di una richiesta. Si analizzano, pertanto, i due strumenti che offrono questo tipo di funzionalità, Grafana e Jaeger.

Grafana ha implementato nelle sue ultime versioni software un visualizzatore di tracciature e la compatibilità di datasource per i popolari sistemi di tracciamento distribuito Jaeger e Zipkin.

Il pannello di visualizzazione implementato in Grafana, permette all'operatore di analizzare nel dettaglio le singole tracciature raccolte, potendo espandere ogni span per accedere a vari dettagli come tags o log associati alla tracciature. Inoltre, entrambi i datasource Jaeger e Zipkin sono dotati di funzionalità di query di base in cui è possibile la ricerca per ID di tracciatura o la selezione di una tracciatura da un selettore in base al nome del servizio, al nome dell'operazione e alla sua durata.

Grazie a queste funzionalità di Grafana, è possibile implementare la visualizzazione grafica del monitoraggio di logs, metriche e tracciature generate da Online Boutique, in un'unica applicazione. Concentrando tutti e tre i pilasti dell'osservabilità in un solo strumento si ha un monitoraggio più semplice che permette di analizzare facilmente ogni tipo di informazione del sistema osservato.

Tuttavia, sebbene il visualizzatore di tracciature implementato da Grafana offra diverse funzionalità per l'analisi delle tracciature raccolte, non è ancora del tutto maturo. Infatti, la navigazione tra le tracciature risulta molto macchinosa e poco intuitiva. L'assenza più grande è la visualizzazione di un elenco delle tracciature con una loro anteprima dettaglia, che mostri le informazioni importanti evitando di aprire una per una le singole tracciature. In presenza di applicazioni distribuite, con molte operazioni effettuate all'interno, l'assenza di queste funzionalità rende complicato lo studio delle comunicazioni avvenute tra i diversi microservizi.

L'interfaccia grafica di **Jaeger**, invece, mostra le tracciature all'utente con un'organizzazione più intuitiva, fornendo una migliore rappresentazione delle comunicazioni interne di Online Boutique. È possibile infatti, raggruppare le tracciature per microservizio, per operazione e per intervallo di tempo. Inoltre, sono presenti le anteprime delle singole tracciature che mostrano diversi dettagli utili per il monitoraggio, come il numero di spans presenti nella tracciature, i microservizi coinvolti, la durata totale e, in caso di malfunzionamenti, il numero di errori generati. La rappresentazione della singole tracciature, invece, è completamente uguale a quella offerta da Grafana, con gli stessi dettagli e rappresentazioni temporali

delle operazioni.

Offrire all'utente tutte queste funzionalità è molto utile e semplifica lo studio del sistema distribuito, aiutando il team di sviluppo in fase di monitoraggio.

Il visualizzatore di tracciature di Grafana e la Jaeger UI fornisco diversi vantaggi per la piattaforma di osservabilità. Con Grafana si concentrano i monitoraggi dei tre pilastri in un unico strumento, con Jaeger, invece, si ha uno studio più semplice e dettagliato delle comunicazioni di Online Boutique. Dati i diversi punti di forza, è utile integrarli entrambi nella piattaforma di osservabilità sviluppata, poiché la collaborazione di questi due strumenti fornisce un monitoraggio completo e avanzato dell'applicazione Online Boutique.

3.5 Piattaforma unica

Nel capitolo precedente, a seguito di un'attenta analisi degli obiettivi fissati e degli strumenti offerti dalla comunità informatica, si è individuato l'insieme di software da installare all'interno del cluster per garantire i tre monitoraggi di log, metriche e tracciature. L'unione di questi tre processi genera la piattaforma unica di osservabilità divisa per le quattro fasi della strategia, mostrata in figura.

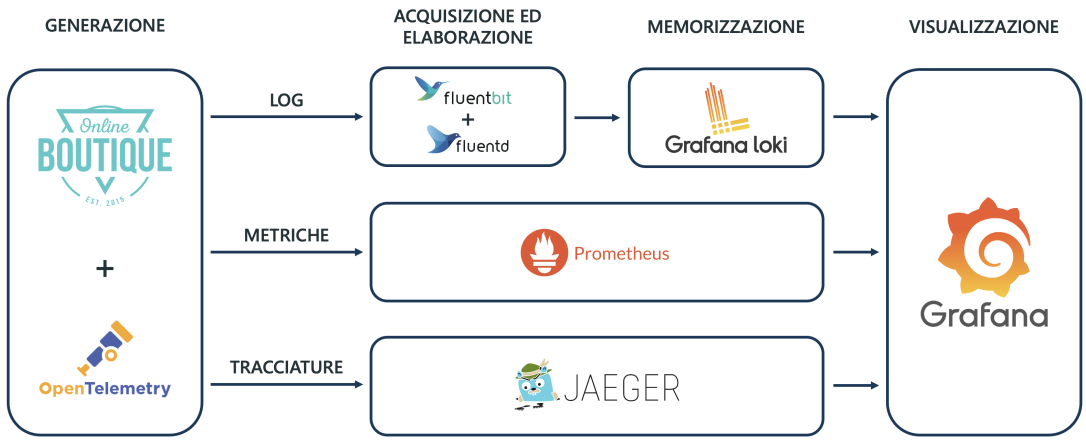

Figura 3.2: Disegno della piattaforma di osservabilità

Come è possibile notare dallo schema della piattaforma, gli strumenti selezionati per le fasi di *acquisizione ed elaborazione* e *memorizzazione* sono completamente dedicati ad un solo pilastro dell'osservabilità. Questa indipendenza dello sviluppo dei tre tipi di monitoraggi implementati semplifica il processo di installazione della piattaforma, garantendo un buon livello di flessibilità allo sviluppatore. Inoltre, in ambienti di media-alta complessità è possibile dedicare lo sviluppo di ciascun monitoraggio a diversi team specializzati, così da lavorare in parallelo sulla piattaforma

riducendo il tempo necessario.

Lo scambio di informazioni tra le applicazioni individuate è un processo fondamentale per garantire il corretto monitoraggio dello stato interno di Online Boutique. È necessario, quindi, configurare correttamente tutti gli strumenti selezionati, al fine di far arrivare a destinazione le informazioni desiderate.

Oggetti risorsa di Kubernetes

Essendo la piattaforma di osservabilità sviluppata su Kubernetes, è importante comprendere come questo sistema gestisca tutte le sue risorse, così da usarle correttamente nell'installazione e configurazione delle applicazioni individuate nella strategia di sviluppo della piattaforma.

Kubernetes usa gli *oggetti risorsa*, entità persistenti nel sistema, per rappresentare lo stato del cluster. Nello specifico, possono descrivere quali sono le applicazioni *containerizzate* in esecuzione e su quali nodi, le risorse disponibili per quelle applicazioni e le politiche relative al comportamento di tali applicazioni, come le politiche di riavvio, gli aggiornamenti e la tolleranza agli errori.

Una volta creato l'oggetto risorsa, il sistema Kubernetes lavorerà costantemente per garantire che l'oggetto esista. Creando un oggetto, si dice effettivamente al sistema Kubernetes come si vuole che sia il carico di lavoro del cluster, definendo quindi lo *stato desiderato*.

Per lavorare con gli oggetti risorsa Kubernetes, per crearli, modificarli o eliminarli, si devono utilizzare l'*API Kubernetes*. Al momento della creazione di un oggetto in Kubernetes, si devono fornire le specifiche dell'oggetto che descrive lo stato desiderato, oltre ad alcune informazioni di base (come il nome). Quando si utilizzano l'API Kubernetes per creare l'oggetto, la richiesta API deve includere tali informazioni in formato JSON. Molto spesso, si usa un file YAML, le cui informazioni sono convertite in JSON da Kubernetes al momento della richiesta.

Per installare e configurare le applicazioni che compongono la piattaforma di osservabilità, si utilizzano alcuni dei più importanti oggetti risorsa di Kubernetes, nello specifico:

- **Deployment**: fornisce aggiornamenti dichiarativi alle applicazioni e consente di descriverne il ciclo di vita, con aspetti quali le immagini da utilizzare, il numero di pod necessari e le modalità di aggiornamento;

- **Service**: è un'astrazione che definisce un insieme logico di pod e le regole per accedervi, garantendo la comunicazione tra applicazioni;

- **DaemonSet**: gestisce le repliche dei pod, assicurandosi che un determinato pod sia eseguito in tutti i nodi del cluster. Man mano che i nodi vengono aggiunti al cluster, vengono aggiunti i pod. Quando i nodi sono rimossi, tali pod vengono eliminati;
- **ConfigMap**: memorizza dati non riservati in coppie chiave-valore. I pod possono utilizzare la ConfigMap come variabile di ambiente, argomento della riga di comando o come file di configurazione in un volume. Inoltre, consente di separare la configurazione specifica dell'ambiente dalle immagini del container, in modo che le applicazioni siano facilmente trasportabili.

Si procede alla descrizione tecnica dei passaggi necessari per installare e configurare la piattaforma di osservabilità all'interno del cluster Kubernetes. Si mostrano le impostazioni più importanti per ciascuno strumento e alcune porzioni di codice delle configurazioni implementate.

3.5.1 Configurazione per i logs

Come anticipato nei capitoli precedenti, la generazione dei logs è già integrata nel codice sorgente dei microservizi di Online Boutique. L'esportazione dei log all'esterno, invece, è garantita dal Pod che gestisce l'esecuzione del singolo microservizio, stampando le informazioni a video.

Per acquisire ed elaborare tutti i logs generati dai Pod di Online Boutique, si passano a Kubernetes i file YAML con le specifiche necessarie per l'installazione di FluentBit e FluentD.

FluentBit è installato come *DaemonSet*, così che anche eventuali futuri nodi del cluster siano monitorati dalla piattaforma di osservabilità. Una volta catturati i logs, FluentBit è configurato per inoltrare tutti i dati alla singola istanza di FluentD. Per questo motivo, dovendo aggregare ed elaborare tutte le informazioni in un unico punto, **FluentD** è installato come *Deployment* associato al suo *Service*. La risorsa Service consente a FluentBit di individuare il pod di FluentD all'interno del cluster, così da inoltrare correttamente i logs letti.

Le istruzioni da eseguire che definiscono il comportamento di FluentBit e FluentD sono scritte in due distinte ConfigMap, una per applicazione. Entrambe usano lo stesso schema per stabilire le operazioni che i due software devono eseguire e quale ordine rispettare. Nel codice delle istruzioni si possono individuare tre parti principali:

- **Input**: è la prima parte di codice delle istruzioni e definisce le sorgenti dalle quali ricevere i logs. Si configura, quindi, FluentBit per l'acquisizione dei logs dalla stampa a video dei pods in esecuzione e FluentD per ricevere tutti i dati da FluentBit, mettendosi in ascolto su una porta del suo Service.

- **Filtri ed elaborazione**: si definiscono i filtri per selezionare i logs che si vogliono analizzare e le operazioni di trasformazione per aggiungere nuovi campi, con informazioni utili, all'interno del singolo oggetto. In FluentBit non si imposta nessun tipo di filtro ed elaborazione dei logs, poiché il suo obiettivo principale è quello di inoltrare tutti i logs a FluentD. Quest'ultimo, invece, fa della sua fase di elaborazione il suo punto di forza. Grazie ai moltissimi plugins disponibili, è possibile filtrare ed elaborare i logs secondo ogni tipo di necessità. Nella configurazione per la piattaforma di osservabilità, si definisco le istruzioni per aggiungere all'interno del log i campi del namespace, pod e app di generazione.

  Si presta particolare attenzione all'ordine con cui sono scritte le istruzioni, poiché è quello rispetto in fase di esecuzione delle due applicazioni.

- **Output**: è l'ultima parte delle istruzioni dove si definiscono le destinazioni dei logs. La configurazione di FluentBit fa si che i logs siano inoltrati a FluentD, raggiungibile grazie al suo Service e alla porta su cui è in ascolto. FluentD, invece, ha come destinazione Loki, incaricato di memorizzare i logs, anche questo raggiungibile tramite coppia Service e porta.

Per verificare che le due applicazioni siano state configurate correttamente, si controllano i logs del pod di FluentD con il comando *kubectl logs fluentd*. Nel caso il comando stampasse a video diversi logs, allora la configurazione è avvenuta con successo.

In seguito alla fase di acquisizione ed elaborazione, si passa alla memorizzazione dei logs selezionati. Seguendo la strategia disegnata, si procede con l'installazione di Loki all'interno del cluster. Per la sua installazione si usa Helm, un'alternativa all'esecuzione manuale dei file YAML.

Helm è uno strumento di gestione di *charts*, ovvero pacchetti preconfigurati di risorse oggetto di Kubernetes. È un progetto costantemente aggiornato dalla sua comunità informatica ed è diventato parte della CNCF.

I charts di Helm aiutano la definizione, l'installazione e l'aggiornamento delle applicazioni più complesse, semplificando molto la loro gestione all'interno di Kubernetes. Sono sempre di più i software disponibili tramite Helm, grazie sopratutto alla possibilità di aggiungere molte repository dalle quali selezionare i charts necessari.

Per l'installazione di **Loki**, la comunità di Grafana ha reso disponibile il suo repository ufficiale (*https://grafana.github.io/loki/charts*) da aggiungere ad Helm. Una volta aggiunto, è necessario lanciare il comando *helm install loki* e si installa automaticamente il Deployment e il Service dell'applicazione all'interno del cluster.

Loki non necessita di nessuna configurazione da parte dello sviluppatore, poichè è completamente automatizzato. Infatti, in seguito alla sua installazione è subito

in grado di memorizzare correttamente i logs ricevuti da FluentD.

L'ultima fase di sviluppo del monitoraggio dei logs è la visualizzazione di tutti i dati sulla dashboard di Grafana. Si installa, quindi, **Grafana** tramite Helm, che aggiungere il Deployment e il Service di Grafana al cluster Kubernetes. Tuttavia, il sistema operativo CentOS minimal, eseguito sulla macchina virtuale, non implementa nessun tipo di interfaccia grafica. Per raggiungere il Service di Grafana, è necessario usare un browser installato sul computer principale. Bisogna, quindi, configurare le impostazioni di rete dei vari componenti affinché sia possibile comunicare correttamente con Grafana e visualizzare le sue dashboard.

Si configura una connessione SSH (Secure Shell) dalla macchina principale verso la macchina virtuale in esecuzione. Inoltre, si modificano le impostazioni del browser configurando un Host SOCKS all'indirizzo IP *127.0.0.1* e porta *8080*. In questo modo, tutte le richieste di rete inviate dal browser della macchina principale sono inoltrate e ricevute dalla macchina virtuale di CentOS.

I Service che si voglio interrogare dal browser del computer principale, devono essere configurati in modo tale da essere raggiungibili dall'esterno del cluster Kubernetes. Si distinguono due tipi di configurazioni di un Service Kubernetes:

- **ClusterIP**: espone il Service sull'IP del cluster su cui è in esecuzione, rendendo il Service raggiungibile solo all'interno di tale cluster.

- **NodePort**: espone il Service sull'IP di ogni nodo del cluster, usando una porta statica. Viene creato automaticamente un ClusterIP Service, verso il quale è inoltrato il NodePort Service. I NodePort Service sono raggiungibili dall'esterno del cluster tramite indirizzo *<NodeIP>:<NodePort>*.

Per raggiungere Grafana, quindi, si modifica il codice YAML del suo Service, convertendo il tipo da ClusterIP a NodePort. Così, Kubernetes automaticamente genererà il NodePort Service di Grafana.

In seguito a queste configurazioni di rete, tramite il browser del computer principale è possibile comunicare con Grafana digitando l'indirizzo *<minikube IP:Grafana NodePort>*.

Infine, l'ultimo passaggio necessario per configurare il monitoraggio dei logs, è aggiungere Loki tra i datasouce di Grafana. Si converte il Service di Loki in NodePort e si passa a Grafana l'indirizzo *<minikube IP:Loki NodePort>*. Accedendo alla sezione *Explore* di Grafana, si possono visualizzare tutti i logs filtrati e con le modifiche configurate in FluentD. Grafana semplifica la rappresentazione dei logs, offrendo la possibilità di filtrarli per namespace, service e pod.

3.5.2 Configurazione delle metriche

Come rappresentato dalla strategia di sviluppo disegnata, per generare le metriche all'interno di Online Boutique è necessario instrumentare il codice sorgente dei microservizi con le librerie open source di OpenTelemetry.

Le OpenTelemetry API delle metriche supportano l'acquisizione di misurazioni sullo svolgimento di un programma in fase di esecuzione. Sono disegnate esplicitamente per processare misurazione grezze, generalmente con l'obiettivo di produrre riepiloghi continui di tali misurazioni, offrendo agli sviluppatori visibilità sulle metriche operative della loro applicazione. È estremamente comune usare le metriche per monitorare valori come l'utilizzo della memoria di un processo o il tasso di errore e creare avvisi per indicare quando un servizio sta violando una soglia predeterminata.

Le OpenTelemetry API definiscono più tipi di metriche, ognuna con le proprie caratteristiche che soddisfano diverse necessità. In particolare, per il monitoraggio di Online Boutique si utilizzano:

- **Counter**: implementa una funzione *Add()* che accetta solo valori positivi. È una metrica monotona crescente, ideale per contare dati come le richieste completate e l'incidenza di errori. I Counter sono adatti per le misurazioni della velocità, infatti dividendo la somma per l'intervallo di tempo è possibile ottenere metriche come il tasso di richiesta e il tasso di errore.
- **ValueRecorder**: in modo semantico, questo strumento registra i valori per eventi discreti in una distribuzione, catturando tutte le informazioni necessarie per trovare in modo sincrono velocità, media e intervallo di tempo grazie all'uso della sua funzione *Record()*. Questo lo rende ideale per molti problemi dell'osservabilità, tra cui la misurazione della latenza delle richieste.

Per osservare lo stato interno di Online Boutique si è utilizzato il metodo *RED*, un approccio di monitoraggio che definisce tre metriche chiave da misurare in ogni microservizio analizzato. Queste tre metriche sono:

- ***R**ate*: il numero di richieste al secondo elaborate dai microservizi;
- ***E**rror*: il numero di richieste al secondo che hanno generato un errore;
- ***D**uration*: la durata di tempo impiegata per processare ogni richiesta.

L'utilizzo delle stesse metriche per ogni microservizio aiuta a creare un formato standard e di facile lettura per le dashboard che devono mostrare i risultati.

Per generare i tre tipi di metriche che garantiscono il monitoraggio di Online Boutique, si implementano, all'interno del codice sorgente dei microservizi, le

funzioni offerte dalle librerie open source di OpenTelemetry. Si riporta, come esempio, il codice sorgente di tali funzioni scritto in Java.

```
private final Meter meter =
                OpenTelemetrySdk.getMeterProvider().
                meterSdkProvider.get("onlineboutique", "0.6");

private LongValueRecorder metricRecorder;
private LongCounter errorCounter;
private LongCounter timeoutCounter;

[...]

AdService service = AdService.getInstance();

//Metric for error count
service.errorCounter =
            service.meter
                .longCounterBuilder("error.counter")
                .build();

// Metric for duration method
service.metricRecorder =
            service.meter
                .longValueRecorderBuilder("time")
                .build();

// Metric for timeout counter
service.timeoutCounter =
            service.meter
                .longCounterBuilder("timeout.counter")
                .build();

[...]

long duration = Duration.between(start, end).toMillis();
service.metricRecorder.record(duration, Labels.empty());

Random rand = new Random();
if(rand.nextInt(6) == 2){
    service.errorCounter.add(1, Labels.empty());
}

if(duration > 300){
service.timeoutCounter.add(1, Labels.empty());
}
```

Listing 3.1: Generazione delle metriche OpenTelemetry in Java

Come è possibile notare dall'esempio, si può dividere il codice sorgente che genera e gestisce le metriche in tre parti distinte.

Nella prima parte (righe 1-7) si inizializza l'oggetto *Meter*, incaricato dell'inizializzazione delle metriche da integrare nel codice. Si dichiarano, inoltre, gli oggetti delle metriche che si intendono utilizzare per il monitoraggio del microservizio. Per lo sviluppo della piattaforma di osservabilità si usano le metriche *errorCounter* e *timeoutCounter*, entrambe di tipo Counter, e la metrica *metricRecorder* di tipo ValueRecorder.

Nella seconda parte (righe 11-29) tramite l'oggetto Meter si inizializzano i tre oggetti delle tre metriche dichiarate, definendo il nome di ciascuna metrica.

Infine, nella terza ed ultima parte (righe 33-43) si implementano le funzioni offerte dalle singole metriche per monitorare i microservizi. Per registrare la durata in millisecondi dell'elaborazione del metodo analizzato, si chiama il metodo *record()* di metricRecorder. Per la gestione della metrica errorCounter, si è scelto di simulare la generazione di errori, così da studiare meglio l'ambiente monitorato. Infatti, si chiama il metodo *add()* di errorCounter con una probabilità del 33 %. Infine, per la metrica timeoutCounter si è scelta una soglia sopra la quale la richiesta viene etichettata come timeout. Nelle codice di esempio, viene chiamato il metodo *add()* di timeoutCounter se la durata della richiesta risulta maggiore di 300 millisecondi.

Successivamente alla generazione delle metriche, è necessario implementare nel codice sorgente le funzioni di OpenTelemetry per esportare le informazioni all'esterno dei microservizi. Seguendo la strategia disegnata per il monitoraggio di Online Boutique, la fase di elaborazione ed acquisizione è gestita da Prometheus. Pertanto, si devono instrumentare i microservizi con nuove funzioni così da garantire una corretta esportazione delle metriche.

```
1 private HTTPServer HTTPserver;
2
3 [...]
4
5 PrometheusCollector.newBuilder()
6     .setMetricProducer(service.meterSdkProvider.getMetricProducer())
7     .buildAndRegister();
8
9 service.HTTPserver = new HTTPServer(8081);
```

Listing 3.2: Prometheus Collector di OpenTelemetry in Java

Come anticipato nel capitolo 3.4, dove si sono analizzati gli strumenti per il monitoraggio del sistema, Prometheus acquisisce le metriche dai microservizi instrumentati seguendo un approccio pull tramite protocollo HTTP. Come si può notare dal codice sorgente riportato, il microservizio monitorato offre una

funzionalità di server HTTP impostando una porta per garantire la comunicazione. Tale configurazione, permette a Prometheus di estrarre correttamente i dati relativi alle metriche generate da Online Boutique.

Inoltre, all'interno del codice sorgente, si definisce il Prometheus Collector in modo tale che il microservizio esponga, sulla porta del server HTTP, le informazioni delle metriche nel formato compatibile con Prometheus.

La fase di acquisizione ed elaborazione delle metriche prevede l'installazione e la configurazione di Prometheus all'interno del cluster Kubernetes. La comunità di Prometheus mette a disposizione degli utenti un repository open source (*https://prometheus-community.github.io/helm-charts*) che, tramite Helm, semplifica l'installazione del *kube-prometheus-stack*. Kube-prometheus-stack è una raccolta di oggetti risorsa Kubernetes, dashboard di Grafana e regole di Prometheus combinate con documentazione e script per fornire un facile monitoraggio del cluster Kubernetes.

Tuttavia prima di installare Prometheus è opportuno studiare i dettagli della configurazione necessaria per impostare i target da monitorare. Kube-prometheus-stack permette l'installazione del Prometheus Operator, che offre un modo più semplice per distribuire, configurare e monitorare le istanze di Prometheus eseguite nel cluster Kubernetes. A tal fine, Prometheus Operator introduce tre tipi di CRD (Custom Resource Definition) in Kubernetes: Prometheus, Alertmanager e ServiceMonitor, che configurati correttamente garantiscono l'acquisizione delle metriche indicando a Prometheus quali microservizi monitorare e sopratutto come raggiungerli.

Analizzando l'architettura e il funzionamento del Prometheus Operator, è presente la risorsa Prometheus che descrive in modo dichiarativo lo stato desiderato di una distribuzione di Prometheus, mentre il ServiceMonitor descrive i target che devono essere monitorati da Prometheus.

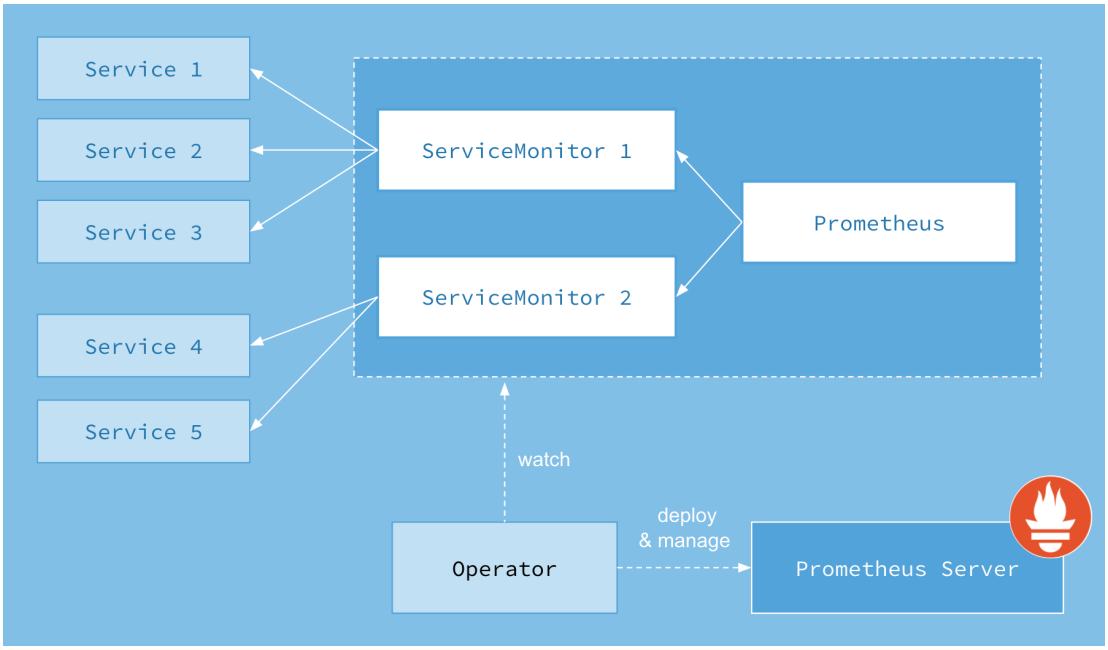

Figura 3.3: Architettura del Prometheus Operator

La risorsa Prometheus include il campo *serviceMonitorSelector* che definisce l'insieme dei ServiceMonitors che devono essere usati. Con le ultime versioni del Prometheus Operator, è possibile selezionare ServiceMonitors presenti fuori dal namespace di Prometheus, usando il campo *serviceMonitorNamespaceSelector* della risorsa Prometheus, dove si possono elencare i namespace dai quali leggere i ServiceMonitors. ServiceMonitor, invece, dispone di un selettore di etichette per selezionare i Service da monitorare.

Per aprire la porta del server HTTP sul Pod del microservizio e per configurare i target di Prometheus, è necessario modificare le risorse oggetto Kubernetes di Online Boutique e creare gli oggetti ServiceMonitor per puntare ai Service dei singoli microservizi monitorati.

La procedura di configurazione è la stessa per tutti i microservizi di Online Boutique, indipendentemente dal linguaggio di programmazione usato. Pertanto, si analizzano tutti i passaggi necessari per garantire a Prometheus una corretta fase di acquisizione delle metriche, si prende come esempio il microservizio AdService.

Come primo passaggio della configurazione, si modifica il codice sorgente YAML del Deployment del microservizio, definendo una nuova porta sulla quale si espongono le metriche generate (righe 19-20).

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: adservice
spec:
  selector:
    matchLabels:
      app: adservice
  template:
    metadata:
      labels:
        app: adservice
    spec:
      containers:
      - name: server
        image: adservice
        ports:
        # Prometheus Scrape
        - name: web
          containerPort: 8081
```

Codice sorgente 1: YAML di AdService Deployment

Aggiunta la nuova porta al Deployment, si modifica il codice sorgente YAML del Service del microservizio associato. Come mostrato nel codice riportato del Service di AdService, si aggiunge una coppia chiave-valore nel campo *metadata.labels* (riga 7), così da rendere il Service individuabile dal ServiceMonitor, e si definisce una nuova porta, coerente a quella implementata nel Deployment (righe 13-15).

```yaml
apiVersion: v1
kind: Service
metadata:
  name: adservice
  # Prometheus Scrape
  labels:
    app: adservice
spec:
  type: ClusterIP
  selector:
    app: adservice
  ports:
  - name: web
    port: 8081
    targetPort: web
```

Codice sorgente 2: YAML di AdService Service

Infine, l'ultimo passaggio per garantire a Prometheus l'acquisizione delle metriche, è la creazione del ServiceMonitor. Si riporta il codice sorgente YAML usato per generare l'oggetto risorsa Kubernetes.

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: adservice
  namespace: default
  labels:
    app.kubernetes: boutique
spec:
  selector:
    matchLabels:
      app: adservice
  namespaceSelector:
    matchNames:
    - default
  endpoints:
    path: /metrics
    port: web
```

Codice sorgente 3: YAML di AdService ServiceMonitor

Analizzando il codice riportato, il ServiceMonitor punta a tutti i Service che hanno le labels indicate nel campo *spec.selector.matchLabels* (righe 8-11). Inoltre, l'etichetta dichiarata nel campo *metadata.labels* (riga 7) permette al ServiceMonitor di essere collegato al Prometheus Operator e i valori del campo *spec.endpoints* (righe 15-17) indicano il percorso del server HTTP che deve essere interrogato da Prometheus.

Configurati tutti i ServiceMonitor dei microservizi di Online Boutique, si installano nel cluster il Prometheus Operator e Grafana, entrambi inclusi in kube-prometheus-stack. Pertanto, si installa l'Helm Charts kube-prometheus-stack passando a linea di comando il file YAML con le specifiche tecniche necessarie per la configurazione del Prometheus Operator.

```
1 prometheus:
2   prometheusSpec:
3     replicas: 1
4     retention: 12h
5     namespaceSelector:
6       any: true
7     serviceMonitorSelector:
8       matchLabels:
9         app.kubernetes: boutique
```

Codice sorgente 4: YAML del Prometheus Operator

La configurazione *namespaceSelector.any: true* (riga 6) permette al Prometheus Operator di raggiungere i ServiceMonitor di tutti i namespaces del cluster. Si usa, invece, il campo *serviceMonitorSelector.matchLabels* per indicare tutte le etichette implementate dai ServiceMonitor, che devono essere lette dal Prometheus Operator.
Inoltre, è possibile impostare nella *ConfigMap* del Prometheus Operator lo *scrape interval*, l'intervallo di tempo che definisce ogni quanto Prometheus deve acquisire le metriche dai targets.

Per verificare che Prometheus funzioni come desiderato e che acquisisca le metriche dei microservizi di Online Boutique, si converte il Service di Prometheus in NodePort così da poter accedere alla sua interfaccia grafica tramite browser. Grazie alla Prometheus UI si possono osservare le metriche catturate e i valori assunti nei diversi intervalli di tempo analizzando la loro rappresentazione grafica. Inoltre, le metriche possono essere interrogate con l'uso di query PromQL, trasformandole con dei modelli matematici per generare nuovi dati da studiare. Prometheus offre una sezione chiamata *Targets*, dove sono elencati tutti i targets configurati e se

sono raggiungibili correttamente.

Essendo un'applicazione molto popolare e di alto livello, la comunità di Prometheus ha implementato all'interno dell'applicazione delle funzionalità di memorizzazione. Pertanto, senza la necessità di installare ulteriori software, è possibile configurare il datasource di Prometheus all'interno di Grafana usando l'indirizzo *<minikube IP:Prometheus NodePort>*.

Si conclude il monitoraggio delle metriche con la rappresentazione grafica in Grafana dei dati raccolti. Grafana, infatti, offre tanti strumenti in grado di analizzare e studiare le metriche acquisite, così da generare grafici e diagrammi che riassumono lo stato interno di Online Boutique.

3.5.3 Configurazione per le tracciature

Data l'assenza di un supporto per la generazione delle tracciature da parte delle librerie standard dei linguaggi di programmazione, anche per questo monitoraggio è necessario instrumentare il codice sorgete dei microservizi di Online Boutique con gli strumenti offerti da OpenTelemetry.

Le OpenTelemetry API sviluppate per le tracciature offrono un ottimo livello di automazione, non richiedendo allo sviluppatore una generazione manuale degli spans. Infatti, OpenTelemetry semplifica l'instrumentazione dei microservizi poiché è in grado autonomamente di tracciare il percorso delle richieste all'interno dell'applicazione.

Si implementa, quindi, nel codice sorgente di Online Boutique, un OpenTelemetry Interceptor che cattura le connessioni in uscita ed in entrata di ogni microservizio, così da garantire la propagazione del contesto per generare la tracciatura distribuita. L'obiettivo principale dell'interceptor è quello di intervenire quando si invia una richiesta ad un altro microservizio e quando un microservizio riceve una richiesta. Nel primo scenario, l'interceptor, prima che la richiesta sia effettivamente inviata, verifica se nel contesto in esecuzione sia presente una tracciatura e nel caso la inserisce nel contesto della richiesta uscente. Nel secondo scenario, invece, l'interceptor controlla se nel contesto della richiesta ricevuta sia presente una tracciatura ed in caso positivo etichetta tutti gli spans generati dal microservizio in esecuzione come suoi figli.

Si mostra il codice dell'OpenTelemetry API, nel caso del microservizio Java AdService, necessario per garantire la generazione automatica delle tracciature.

```
server = ServerBuilder.forPort(port).addService(new AdServiceImpl())
        .addService(healthMgr.getHealthService())
        .intercept(new OpenTelemetryServerInterceptor())
        .build().start();
```

```
5  [...]
6
7  private class OpenTelemetryServerInterceptor implements io.grpc.
       ServerInterceptor {
8      @Override
9      public <ReqT, RespT> ServerCall.Listener<ReqT> interceptCall(
10             ServerCall<ReqT, RespT> call, Metadata headers,
       ServerCallHandler<ReqT, RespT> next) {
11       Context extractedContext = textFormat.extract(Context.current()
       , headers, getter);
12       InetSocketAddress clientInfo =
13               (InetSocketAddress) call.getAttributes().get(Grpc.
       TRANSPORT_ATTR_REMOTE_ADDR);
14       try (Scope scope = ContextUtils.withScopedContext(
       extractedContext)) {
15           Span span =
16                tracer
17                        .spanBuilder("everything")
18                        .setSpanKind(Span.Kind.SERVER)
19                        .startSpan();
20         try {
21           return Contexts.interceptCall(Context.current(), call,
       headers, next);
22         } finally {
23         }
24       }
25     }
26   }
```

Listing 3.3: OpenTelemetry Interceptor in Java per le tracciature

Nella prima parte del codice riportato, si aggiunge l'interceptor nella configurazione del server così da catturare le comunicazioni gestite dal microservizio (riga 3); nella seconda parte è mostrato il codice chiamato dall'interceptor per analizzare il contesto propagato nelle richieste in entrata e in uscita.

Sebbene OpenTelemetry generi in maniera automatica le tracciature delle richieste, offre alcune funzioni per gestirle manualmente. È possibile, infatti, catturare la tracciatura in esecuzione, generare manualmente gli spans e arricchire il contesto con l'aggiunta di nuovi campi. Così, lo sviluppatore può gestire al meglio le tracciature, aggiungendo qualsiasi tipo di informazione necessaria per migliorare il monitoraggio delle richieste.

In seguito alla generazione automatica delle tracciature, per procedere con la fase di acquisizione ed elaborazione come definito dalla strategia di sviluppo della piattaforma, è necessario inviare tutte le informazioni all'agente esterno Jaeger. Si instrumentano, quindi, i microservizi aggiungendo il Jaeger Exporter e si riporta il

codice implementato in Java AdService per inviare le tracciature a Jaeger.

```
1  private void setupJaegerExporter() {
2
3      int jaeger_port = 14250;
4      String jaeger_ip = "simplest-collector.observability.svc.cluster.
       local";
5
6
7      ManagedChannel jaegerChannel =
8              ManagedChannelBuilder.forAddress(jaeger_ip, jaeger_port)
9              .usePlaintext().build();
10
11     // Export traces to Jaeger
12     this.jaegerExporter =
13             JaegerGrpcSpanExporter.newBuilder()
14                     .setServiceName("adservice")
15                     .setChannel(jaegerChannel)
16                     .setDeadlineMs(30000)
17                     .build();
18
19     // Set to process the spans by the Jaeger Exporter
20     OpenTelemetrySdk.getTracerProvider()
21             .addSpanProcessor(SimpleSpanProcessor.newBuilder(this.
       jaegerExporter).build());
22   }
```

Listing 3.4: Jaeger Exporter in Java

La funzione di esportazione è direttamente offerta da OpenTelemetry e permette che le tracciature generate siano inviate correttamente a Jaeger. Per far si che i microservizi possano instaurare una comunicazione con Jaeger, si definisce all'interno della funzione la coppia di coordinate IP:porta di destinazione (righe 3-5). Tuttavia, è possibile notare che l'indirizzo IP usato per raggiungere Jaeger non è composto da quattro campi numerici ma è un indirizzo testuale. Kubernetes, infatti, offre un sistema di DNS che permette di raggiungere i Service tramite URL, aiutando gli sviluppatori nelle configurazioni di rete. Inoltre questa soluzione garantisce la corretta comunicazione nel caso l'IP del Service dovesse cambiare.

Definire il Jaeger Exporter all'interno del codice sorgente, serve anche per impostare il formato con cui sono salvate le informazioni riguardanti le tracciature presenti nel contesto delle richieste. Infatti, ogni applicazione di elaborazione delle tracciature è compatibile con un solo formato diverso da quello delle altre, rendendo impossibile, quindi, la definizione di più Exporter all'interno dei microservizi. Pertanto, è importante definire il formato corretto in base all'agente collettore

implementato affinché sia garantito il monitoraggio.

L'installazione di Jaeger all'interno del cluster è possibile grazie a diversi file YAML, offerti dalla comunità informatica, che generano tutte le risorse Kubernetes necessarie per il suo funzionamento. Jaeger è un'applicazione ad alto livello, pertanto non richiede dopo l'installazione nessun tipo di configurazione.

Per verificare che i microservizi di Online Boutique generino correttamente le tracciature delle richieste elaborate dall'applicazione, si accede alla Jaeger UI tramite browser. Jaeger non espone il proprio Service all'esterno del cluster con il NodePort, bensì usa la risorsa Ingress. **Ingress** è un oggetto API di Kubernetes che gestisce gli accessi esterni ai Service del cluster definendo al suo interno delle regole che gestiscono l'instradamento delle richieste HTTP e HTTPS.

Accedendo all'interfaccia grafica di Jaeger, è possibile navigare tra tutte le tracciature acquisite, osservare i grafici generati e analizzare i tempi di esecuzione di ogni singola richiesta gestita dai microservizi.

Jaeger, inoltre, implementa una propria funzionalità di memorizzazione così che non servano altri strumenti per configurare il datasource. Infatti, sebbene Jaeger già fornisca una interfaccia grafica, è possibile visualizzare le tracciature anche accedendo a Grafana, configurando il datasource di Jaeger raggiungibile all'indirizzo *<minikube IP:Jaeger Ingress>*.

Tuttavia, il pannello di navigazione delle tracciature offerto da Grafana in molte situazioni risulta poco comodo per il monitoraggio delle applicazioni. In questi casi, conviene infatti usare la Jaeger UI, molto più chiara per gli sviluppatori poiché offre un maggior livello di dettaglio delle tracciature semplificando il monitoraggio dei microservizi.

3.6 Estensioni enterprise della piattaforma

La piattaforma sviluppata finora, con gli strumenti definiti dalla strategia disegnata, offre un alto livello di monitoraggio grazie all'instrumentazione dei microservizi per renderli osservabili. Infatti, sono forniti tutti i dettagli dello stato interno di Online Boutique così da analizzare a fondo il suo comportamento. Tuttavia, sono possibili ulteriori modifiche ed implementazioni in modo tale da rendere l'intera infrastruttura molto più stabile e più efficiente.

Estensioni della piattaforma che possono aiutare il team di sviluppo nelle attività di monitoraggio sono sicuramente l'integrazione dell'alta affidabilità dei dati e la generazione delle metriche da determinati logs stampati da Online Boutique.

3.6.1 Alta affidabilità

Garantire l'alta affidabilità dei dati all'interno della piattaforma di osservabilità comporta l'implementazione di strumenti avanzati per la memorizzazione delle informazioni generate. Questi strumenti devono assicurare l'archiviazione dei dati sopratutto in condizioni di poca stabilità, come un riavvio improvviso o un malfunzionamento hardware, così da evitare la perdita di dati significativi.

Per il team di sviluppo incaricato nel monitoraggio, è importante poter fare affidamento su logs, metriche e tracciature memorizzate per lunghi periodi di tempo. In tal modo, infatti, si può confrontare lo stato attuale del sistema con uno simile del passato e, con buona probabilità, prevedere comportamenti futuri del sistema.

Gli strumenti usati dalla piattaforma per memorizzare i tre i pilastri dell'osservabilità sono Loki per i logs, Prometheus per le metriche ed infine Jaeger per le tracciature. La memorizzazione dei logs è gestita diversamente dalle metriche e tracciature, poiché i logs risultano essere molto pesanti per essere conservati eccessivamente a lungo. Loki, inoltre, fornisce un sufficiente livello di affidabilità e pertanto non è necessario installare nuove applicazioni. Al contrario, Prometheus e Jaeger implementano una scarsa affidabilità dei dati, esponendo ad un alto rischio le informazioni conservate da metriche e tracciature.

Si estende, quindi, la piattaforma di osservabilità e si installano due nuovi strumenti entrambi specializzati nell'alta affidabilità dei dati: **Thanos**, affiancato a Prometheus per la memorizzazione delle metriche, ed **Elasticsearch**, in coppia con Jaeger per memorizzare le tracciature.

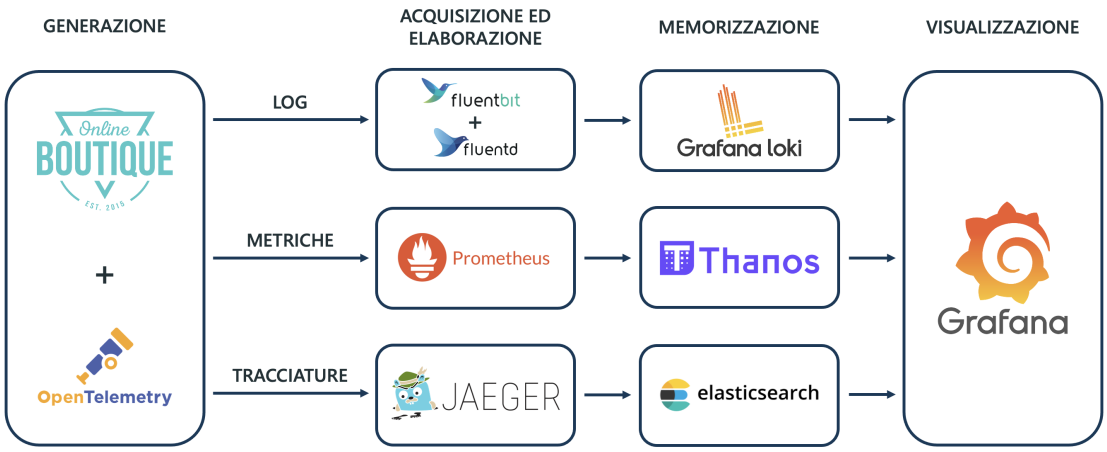

Figura 3.4: Disegno della piattaforma di osservabilità con alta affidabilità

Thanos

Thanos è un'applicazione che offre una visualizzazione globale delle query che interrogano le metriche, alta affidabilità e backup dei dati con cronologia. Tutte questi servizi possono essere installati indipendentemente dagli altri. Così è possibile avere un sottoinsieme delle funzionalità di Thanos immediatamente pronte a migliorare la piattaforma, offrendo anche una flessibilità per delle implementazioni graduali negli ambienti più complessi.

Oltre a garantire una memorizzazione a lungo termine per i dati raccolti, Thanos è in grado di collaborare con più istanze di Prometheus. Infatti, propaga le query a tutti i Prometheus a cui è collegato ed aggrega in un unica visualizzazione le metriche estratte. Questo servizio è fornito grazie alla implementazione del *Thanos Sidecar*, uno dei principali componenti di Thanos. Il Thanos Sidecar è un container che si affianca all'applicazione principale Prometheus, con la quale condivide diverse risorse, permettendo a Thanos di gestire tutte le istanze di Prometheus.

Il Thanos Sidecar è offerto direttamente dal Prometheus Operator e per installarlo si modifica il file YAML con la configurazione di Prometheus, aggiungendo i campi relativi al Thanos Sidecar. Affinché l'installazione di Thanos si completa, si installano nel cluster altre due sue componenti: il *Thanos Store*, che memorizza i dati integrando l'alta affidabilità, e il *Thanos Query*, che gestisce le query per estrarre le metriche e offre l'interfaccia grafica di Thanos, molto simile alla Prometheus UI.

Elasticsearch

Elasticsearch è un motore distribuito, open source ed analitico per tutti i tipi di dati, inclusi quelli testuali, numerici, geospaziali, strutturati e non-strutturati. Elasticsearch è sviluppata in Apache Lucene ed è diventata molto popolare grazie alle sue semplici REST APIs, alla sua natura distribuita, alla sua velocità e scalabilità. Proprio la sua velocità e scalabilità rende Elasticsearch la soluzione perfetta per memorizzare le tracciature, poiché, grazie alla sua capacità di indicizzazione, si integra perfettamente con Jaeger.

Elasticsearch garantisce un altissima affidabilità sui dati conservati. Infatti, dati ricevuti da Jaeger sono prima soggetti ad un processo di *data ingestion*, durante il quale le informazioni sono parsificate, normalizzate ed arricchite, per poi essere indicizzate. Solo in seguito, gli utenti possono eseguire query complesse per interrogare ed analizzare i dati.

Elasticsearch memorizzata tutte le informazioni come documento JSON e conserva i dati sotto forma di *index*. Un index di Elasticsearch è una collezione di documenti correlati tra di loro. Ogni documento, infatti, è legato ad un insieme

di *chiavi* (nomi dei campi o proprietà) con i loro corrispondenti valori (stringhe, numeri, booleani, dati, vettori di valori, geolocalizzazioni, o altri tipi di dato).

Elasticsearch usa una struttura dati chiamata *index invertito*, disegnata per offrire una altissima velocità nella ricerca testuale. Un index invertito elenca ogni singola parola che compare in qualsiasi documento ed identifica tutti i documenti in cui quella parola è presente. Durante il processo di indicizzazione, Elasticsearch memorizza i documenti e crea i suoi index invertiti per garantire una performante ricerca in tempo reale dei dati del documento.

La struttura di memorizzazione di Elasticsearch rende la conservazione dei dati completamente distribuita. I documenti memorizzati, infatti, sono distribuiti su diversi nodi definiti *shards*, che vengono duplicati per fornire copie ridondati dei dati, così da garantire un recupero in caso di qualsiasi tipo di guasto.

Grazie alle API ufficiali, fornite dalla sua comunità, si installa Elasticsearch nel cluster passando a Kubernetes il file YAML con la configurazione richiesta. Per configurare Jaeger con Elasticsearch, così da inoltrargli tutte le tracciature acquisite, si modifica il codice YAML del suo oggetto risorsa, inserendo la coppia di coordinate *IP:porta* per raggiungere Elasticsearch ed indicando il numero di shards e repliche desiderati.

3.6.2 Generazione delle metriche dai logs

I logs contengo informazioni molto importanti, sopratutto in fase di debug delle applicazioni. Avere un elenco con tutte le azioni compiute da tutti i microservizi aiuta a comprendere i processi elaborativi svolti dal sistema monitorato. Tuttavia, a causa della struttura completamente testuale, trasformare ed elaborare i logs al fine di ottenere nuovi tipi di informazioni risulta praticamente impossibile. Inoltre, essendo molto pesanti come dati, la loro memorizzazione per lunghi periodi di tempo è sconsigliata. Una soluzione per colmare questi due limiti, è quella di generare le metriche dai logs generati dal sistema, poiché le metriche implementano una struttura leggere e adatta alla trasformazione con modelli matematici.

Sebbene sia possibile integrare questa funzionalità per generare metriche da tutti i tipi di logs, nel caso del monitoraggio di Online Boutique si sceglie un solo tipo di log così da mostrare le potenzialità di questa estensione della piattaforma e fornire dettagli sulla configurazione implementata.

Si vogliono generare delle metriche per osservare gli acquisti effettuati dagli utenti, studiando gli articoli comprati e la nazionalità dei clienti. Quando viene confermato un ordine dal sito di Online Boutique, il microservizio *checkoutservice* stampa un logs che descrive l'evento accaduto.

```
log="{"country":"Spain","item":"smartphone","message":"Order",
      "scrape":"yes","severity":"info",
      "timestamp":"2021-03-22T11:18:38.669690075Z"}"
stream="stdout" time="2021-03-22T11:18:38.670078494Z
container="server" message="Order"
item="smartphone"country="Spain" scrape="yes"
```

Codice sorgente 5: Log di acquisto generato da *checkoutservice*

Per generare la metrica corrispondente a questo logs, si utilizzano le funzionalità offerte da FluentD. FluentD, infatti, implementa diversi plugin in grado di generare le metriche e di esporle all'esterno dell'applicazione, così Prometheus può acquisirle correttamente.

Si aggiorna, quindi, la ConfigMap di FluentD aggiungendo due porzioni di codice necessarie implementare questa estensione.

```
# input plugin that exports metrics
<source>
  @type prometheus
  port 8081
  metrics_path /metrics
</source>

[...]

@type copy
<store>
  @type prometheus
  <metric>
    name onlineboutique_orders
    type counter
    desc ordini su online boutique
    <labels>
      item ${item}
      country ${country}
      validOrder ${scrape}
    </labels>
  </metric>
</store>
```

Codice sorgente 6: YAML di FluentD per esporre le metriche

Analizzando il codice riportato, nella prima parte si definiscono la porta e il percorso sulla quale si espongono le metriche generate (righe 2-6), così da permettere a Prometheus di acquisirle correttamente. Nella seconda, invece, si costruisce la metrica, configurando il nome della metrica (riga 14), il suo tipo (riga 15), la sua descrizione (riga 16) ed infine tutte le labels con i campi necessari per il monitoraggio (righe 18-20). Nel caso dell'esempio, si è scelto di creare una metrica Counter, incrementata ogni volta che viene effettuato un acquisto sull'ecommerce, con delle etichette aggiuntive che forniscono informazioni sull'oggetto acquistato e la nazione di provenienza del cliente. Questa configurazione, permette di fare molte statistiche sulle abitudini dei clienti, come vedere gli oggetti più venduti e le nazionalità più frequenti.

Come per i microservizi di Online Boutique, anche per FluentD è necessario modificare il codice YAML del Deployment e del Service e creare il ServiceMonitor, in modo tale da garantire la corretta comunicazione tra l'applicazione e Prometheus. Così, FluentD verrà aggiunto alla lista dei target che Prometheus dovrà contattare per acquisire le metriche.

È possibile generare metriche da tutti i tipi di logs, migliorando quindi il sistema di monitoraggio, tuttavia è necessario conoscere bene la struttura del log che si vuole analizzare e tutti i suoi campi, affinché sia possibile implementare la giusta configurazione.

3.6.3 Possibili estensioni

Le piattaforme di osservabilità sono dei sistemi in continua evoluzione e che devono essere sempre al passo con le nuove tecnologie informatiche. Infatti, si possono aggiungere all'interno della piattaforma tante altre estensioni per renderla ancora più completa. Due possibili estensioni che aiutano gli sviluppatori nella fase di sviluppo e monitoraggio, sono l'allarmistica intelligente e l'instrumentazione automatica di OpenTelemetry all'interno dei microservizi.

Allarmistica smart

La memorizzazione e lo studio continuo di logs, metriche e tracciature anche degli stati passati del sistema, sono processi importanti e fondamentali per raggiungere un alto livello di monitoraggio. Inoltre, essendo i sistemi in continua e rapida evoluzione, ci sono comportamenti che non si possono prevedere solo su base teorica, ma richiede un'analisi ben precisa di tutto l'ambiente su cui si lavora.

L'introduzione dell'allarmistica all'interno delle piattaforme, ha fatto si che la dinamicità del sistema fosse sempre sotto controllo da parte degli sviluppatori. Gli allarmi, infatti, permettono di ricevere notifiche dalla piattaforma al verificarsi di di

certe condizioni critiche per l'esecuzione del sistema. L'obiettivo è quello di prevenire errori e malfunzionamenti intervenendo in tempo sul sistema e sull'infrastruttura, così da non compromettere l'erogazione del servizio.

Attraverso le piattaforme di monitoraggio, si possono configurare le soglie degli allarmi soglie su determinati parametri di controllo in continuo cambiamento, come per esempio il numero di utenti connessi ad un sito. Le soglie sono tipicamente impostate su valori utili al monitoraggio e non estremamente al limite di una situazione critica, poiché queste lanciano solo una notifica catturando l'attenzione dell'operatore e non intervengono in prima persona sul sistema.

È chiaro, quindi, che il lavoro degli sviluppatori nella scelta dei parametri su cui impostare un allarme e nell'individuare la soglia migliore per il monitoraggio, è un'operazione fondamentale per prevenire errori e garantire la corretta esecuzione del sistema. Tuttavia può capitare che il team di monitoraggio non si accorga che alcuni parametri richiedevano degli allarmi oppure che la soglia impostata non fosse abbastanza significativa.

Una soluzione che migliora la configurazione e la gestione degli allarmi è l'*allarmistica intelligente*. Strumenti di allarmistica intelligente hanno alla base algoritmi di intelligenza artificiale e machine learnig, che studiando le informazioni dello stato passato, in particolare le metriche che si adattano di più a questo tipo di elaborazioni, riescono a fare confronti con lo stato presente e valutare la presenza di rischi.

Pertanto, integrandoli nella piattaforma di osservabilità, questi strumenti possono configurare i valori delle soglie più indicati per monitorare lo stato attuale, così da essere subito notificati nel caso il sistema stesse tendendo verso una situazione critica. Inoltre, possono gestire autonomamente tutti gli allarmi, decidendo quali paramenti monitorare e cambiando la loro configurazione dinamicamente, a seconda dell'evolversi del sistema.

Introdurre sistemi di allarmistica intelligente è una scelta altamente consigliata poiché spesso, sopratutto in sistemi molto estesi, un solo team non è in grado di monitorare un ambiente così complesso, mettendo a rischio l'intera infrastruttura.

Tuttavia, è importante comprendere che strumenti di allarmistica intelligente, integrati nelle piattaforme di monitoraggio, non risolvono tutti i tipi di problemi e non garantiscono completamente l'assenza di malfunzionamenti. L'allarmistica intelligente, infatti, deve essere considerata come modulo di supporto ai team di sviluppatori incaricati nel monitorare il sistema poiché la componente umana deve essere sempre presente.

Instrumentazione automatica di OpenTelemetry

Un'estensione che può aiutare gli sviluppatori ad integrare l'osservabilità nelle applicazioni sviluppate, è l'instrumentazione automatica di OpenTelemetry, un progetto che la sua comunità ufficiale sta sviluppando ed attualmente è in fase di beta.

Disponibile ora solo per Java, l'instrumentazione automatica di OpenTelemetry permette agli sviluppatori di integrare tutte le funzionalità di OpenTelemetry all'interno dei codici sorgenti dei microservizi, lanciando un solo comando da terminale. Il progetto fornisce un Java Agent JAR che può essere collegato a qualsiasi applicazione scritta in Java 8+ e inietta dinamicamente dei byte per acquisire dati di telemetria da una serie di librerie e framework popolari. È possibile esportare i dati di telemetria in una varietà di formati e si possono configurare l'agente e l'exporter tramite argomenti della riga di comando.

OpenTelemetry mette a disposizione degli utenti un pacchetto che include l'agente di instrumentazione, nonché la strumentazione per tutte le librerie supportate e tutti gli esportatori di dati disponibili, offrendo un'esperienza completamente automatica e pronta all'uso. Il risultato finale dell'instrumentazione automatica è la capacità di raccogliere dati di telemetria da un'applicazione senza modifiche al codice.

3.7 Confronti

Con l'aggiunta dell'alta affidabilità dei dati e della generazione delle metriche dei logs, si conclude lo sviluppo della piattaforma di osservabilità. È utile, a questo punto, confrontare le funzionalità implementate con altre possibili soluzioni alternative. In questo paragrafo si confrontano l'instrumentazione manuale realizzata con quella automatica di OpenTelemetry, e le funzionalità offerte dalla piattaforma con quelle fornite da Istio.

3.7.1 Instrumentazione manuale realizzata e instrumentazione automatica di OpenTelemetry

Per la maggior parte degli utenti, l'instrumentazione automatica di OpenTelemetry, subito pronta all'uso, è completamente sufficiente. A volte, tuttavia, gli utenti desiderano aggiungere attributi agli spans generati, altrimenti automatici, o potrebbero voler creare manualmente spans per il proprio codice personalizzato. È necessario, quindi, confrontare vantaggi e svantaggi dei due tipi di instrumentazione possibili.

La caratteristica principale dell'instrumentazione automatica di OpenTelemetry è proprio quella di offrire all'utente un'esperienza completamente pronta all'uso. Non è richiesta, infatti, una modifica al codice sorgente, ma si lancia solo un comando da

terminale. Si evita, così, di compromette accidentalmente l'applicazione sviluppata e di prestare particolare attenzione alla configurazione necessaria, ma tutto è completamente gestito autonomamente da OpenTelemetry.

Tuttavia, sebbene l'instrumentazione automatica sia configurabile con diverse opzioni, visto che molti aspetti del comportamento del Java Agent possono essere configurati in base alle proprie esigenze, come la scelta e configurazione dell'esportatore (come e dove vengono inviati i dati), le intestazioni di propagazione del contesto di tracciature e altro, non offre livello di personalizzazione elevato tanto quanto l'instrumentazione manuale di OpenTelemetry.

Il punto di forza, infatti, dell'instrumentazione manuale è proprio quello di avere un altissimo livello di gestione delle metriche e tracciature. Grazie alla vasta libreria di OpenTelemetry, si possono implementare nel codice sorgente dei microservizi funzioni per generare e arricchire le tracciature e usare diversi tipi di metriche secondo le diverse necessità, cosi da raggiungere un livello più alto e dettagliato. Tuttavia, andando a modificare il codice sorgente delle applicazioni, è necessario conoscere bene il funzionamento del sistema che si vuole instrumentare e delle funzioni che si vogliono implementare, sopratutto nella configurazione per la propagazione del contesto e per gestire gli esportatori verso i quali inoltrare i dati di telemetria.

Un team di sviluppo incaricato del monitoraggio di un sistema deve obbligatoriamente valutare le due soluzioni per sviluppare la miglior infrastruttura secondo le proprie necessità. Per ambienti di medio-alto livello, per i quali serve un monitoraggio più dettagliato, sicuramente l'instrumentazione manuale è la più indicata, ma richiederebbe tempo e abilità per implementarla correttamente nelle applicazioni.

3.7.2 Istio e livello di funzionalità raggiunto dalla piattaforma sviluppata

Sebbene sia Istio che la piattaforma sviluppata abbiano come obiettivo quello di osservare e controllare il sistema che si vuole monitorare, offrono un insieme di funzionalità differenti. Istio è una piattaforma che consente di controllare in che modo le diverse componenti di un'applicazione a microservizi condividono i dati ed è suddivisa logicamente in una infrastruttura dati e una infrastruttura di controllo:

- l'*infrastruttura dati* è composta da un insieme intelligente di proxy (Envoy) installati come sidecar di tutti i Pod monitorati. Questi proxy mediano e controllano tutte le comunicazioni di rete tra i diversi microservizi, e collezionano e registrano i dati di telemetria del sistema.
- l'*infrastruttura di controllo* configura i proxy affinché gestiscano correttamente il traffico.

Istio usa una versione estesa del proxy Envoy. **Envoy** è un proxy che offre alte prestazioni gestendo tutto il traffico in entrata ed in uscita da tutti i servizi presenti e fornisce un vasto insieme di funzionalità come il bilanciamento del carico di elaborazione e le informazioni sullo stato di salute del microservizio. Tramite l'infrastruttura di controllo, si convertono le regole di routing di alto livello che gestiscono il comportamento del traffico in configurazioni specifiche di Envoy, che le propaga ai sidecar in fase di esecuzione.

Questo tipo di architettura permette di osservare e controllare l'ambiente distribuito da un unico punto di controllo. Istio per gestire al meglio la fase di monitoraggio, integra Kiali, una console di gestione per la rete di servizi. **Kiali** fornisce dashboard, osservabilità e mostra la struttura del sistema a microservizi deducendo la topologia del traffico. Inoltre, consente l'accesso a Grafana e un'integrazione per le tracciature distribuite con Jaeger.

Per quanto questo insieme di strumenti implementi funzionalità di monitoraggio e controllo, non offre una personalizzazione configurabile dall'utente per osservare con diverse granularità i processi elaborati dai microservizi. Come, infatti, per l'instrumentazione automatica di OpenTelemetry, in Istio non è presente la possibilità di generare e gestire manualmente metriche e tracciature, caratteristica molto utile per raggiungere un buon livello di osservabilità a seconda delle necessità.

Istio, quindi, risulta più indicato per la gestione delle risorse e del traffico dei microservizi, ma per per un monitoraggio di alto livello, la piattaforma sviluppata garantisce una migliore osservabilità del sistema.

3.8 Risultati ottenuti

La piattaforma sviluppata, grazie all'osservabilità implementata nei microservizi di Online Boutique, fornisce una fase di monitoraggio molto dettagliata e flessibile. L'obiettivo è quello di fornire strumenti per il monitoraggio di Online Boutique, attraverso rappresentazioni grafiche in seguito a processi di acquisizione ed elaborazione delle informazioni interne all'applicazione. Si mostrano i risultati finali ottenuti in seguito allo sviluppo delle quattro fasi della strategia per i tre pilastri dell'osservabilità.

3.8.1 Rappresentazione grafica dello stato del sistema

Tramite l'acquisizione ed elaborazione delle metriche generate grazie ad OpenTelemetry e gestite da Prometheus, Grafana è in grado di analizzare e studiare tutte le informazioni ricevute. Offre, infatti, una vasta gamma di strumenti per personalizzare al meglio le dashboard, così da monitorare ogni singola componente dell'applicazione.

Seguendo il metodo *RED* per il monitoraggio dei sistemi, sono state create rappresentazioni grafiche utili per osservare le tre categorie Rate, Error e Duration. Si mostra la dashboard di Grafana composta dai pannelli dinamici per lo studio dello stato interno di Online Boutique.

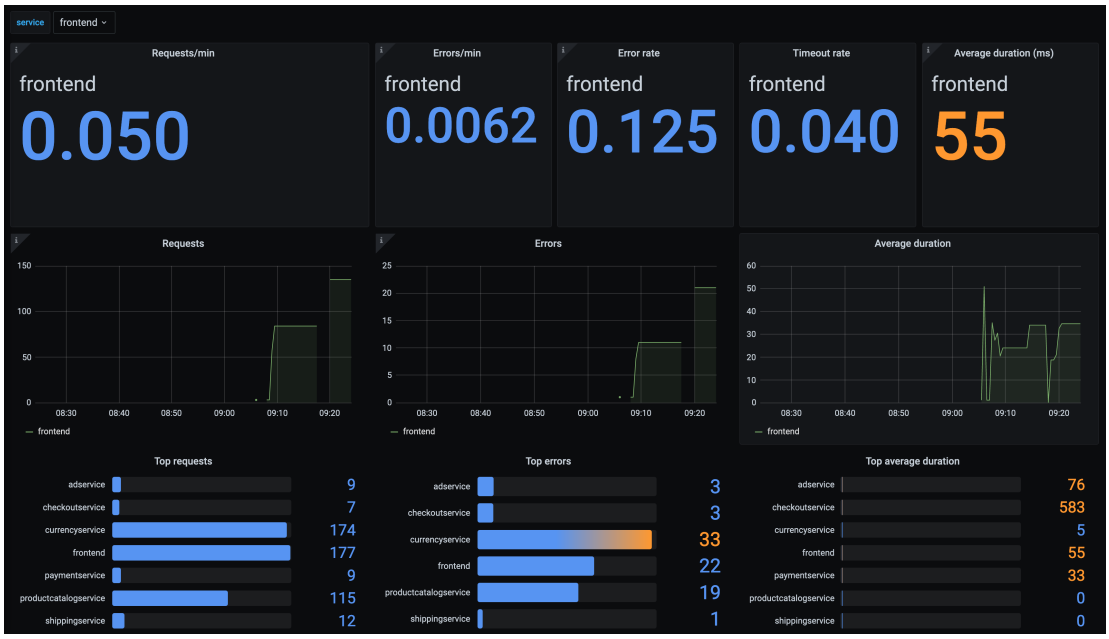

Figura 3.5: Dashboard metodo RED per monitorare lo stato interno di Online Boutique

La dashboard disegnata è suddivisibile in tre colonne, ognuna per la singola categoria di monitoraggio che forniscono informazioni simili per i diversi tipi di metriche. Per la creazione di ciascun pannello si usano query PromQL per estrarre ed aggregare i dati utili alla rappresentazione.

La configurazione implementata in Grafana fornisce l'analisi dell'ultima ora dello stato interno del sistema. Inoltre, essendo possibile definire variabili globali utilizzabili all'interno delle query PromQL per la generazione dei pannelli, si può cambiare il microservizio che si vuole osservare dal menù in alto a sinistra.

Passando alla visualizzazione dei logs, Grafana fornisce un pannello dinamico tramite il quale è possibile impostare filtri su Pod, Service e App. Si mostrano i logs generati dal microservizio *checkoutservice* in seguito all'acquisto di un oggetto tramite Online Boutique.

Grafana, inoltre, fornisce un diagramma temporale dove sono mostrati, per ogni istante, il numero di logs generati ed identifica quei logs che riportano eventi di

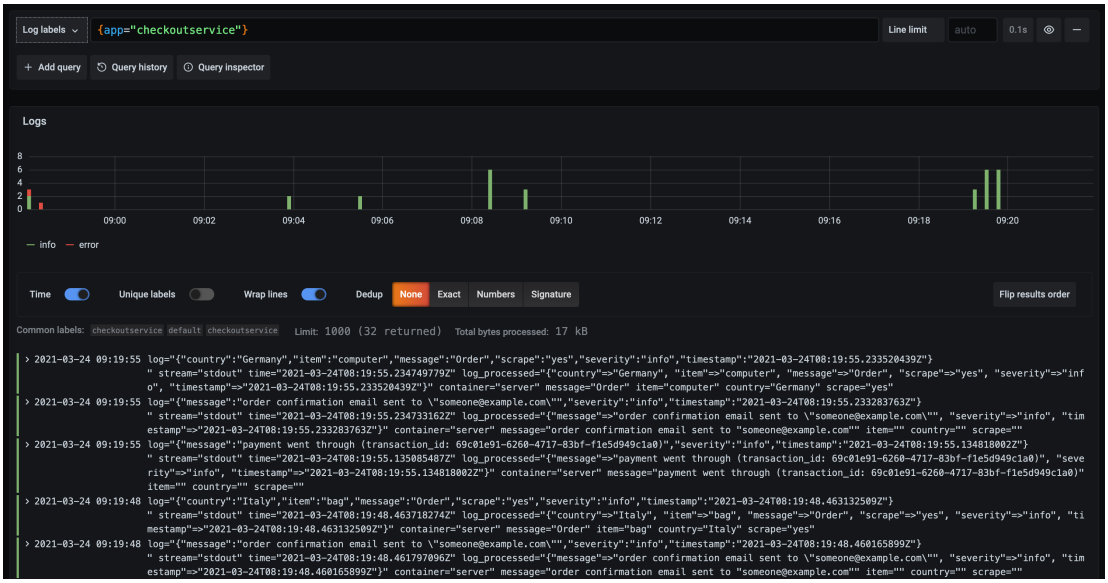

Figura 3.6: Logs di checkoutservice

errore (colorati di rosso). È possibile visualizzare i dettagli dei singoli logs, dove è presente un elenco con tutti i campi già parsificati.

Grazie alla aggregazione in un unico punto di tutti i logs del sistema e alla possibilità di filtrarli per microservizio, è semplice analizzare anche applicazioni a microservizi molto distribuite.

Si può notare che l'ultimo log generato da checkoutservice (il primo dell'elenco) mostra alcuni dettagli riguardo una transazione avvenuta, fornendo informazioni sulla nazionalità del cliente e sull'oggetto acquistato.

Grazie agli strumenti integrati nella piattaforma di osservabilità, è possibile generare da questo tipo di log una metriche Counter, incrementata ad ogni ordine eseguito su Online Boutique, che ha come etichette *nazionalità* ed *oggetto*. Si disegnano, quindi, due pannelli di Grafana che analizzando questa metrica, monitorano gli acquisti in base a queste due informazioni.

Figura 3.7: Pannelli elaborati dalle metriche generate dai logs di checkoutservice

3.8.2 Rappresentazione grafica della propagazione delle richieste nel sistema

Infine, grazie alle tracciature generate tramite OpenTelemetry è possibile analizzare approfonditamente le singole richieste ricevute da Online Boutique ed elaborate dai suoi microservizi. Usando il pannello collegato a Jaeger, si può visualizzare un elenco con tutte le tracciature acquisite ordinate per microservizi. Si mostra un esempio di tracciatura generata al momento della conferma di un acquisto effettuato sull'ecommerce.

Figura 3.8: Tracciatura generata da Online Boutique per la conferma di un acquisto

Si può notare come la tracciatura fornisca molti dettagli riguardo la propagazione di una richiesta all'interno dell'intera rete di microservizi di Online Boutique. Si possono visualizzare, infatti, tutti gli spans che costituiscono la tracciatura, analizzando i microservizi che hanno partecipato alla gestione della richiesta, i tempi di elaborazione delle funzioni eseguite e i loro ordine di esecuzione.

Inoltre, Grafana permette di visualizzare maggiori informazioni dei singoli span della tracciatura, tra cui i campi che questo contiene. Si mostrano i dettagli del secondo span della tracciatura descritta in *Figura 3.8*.

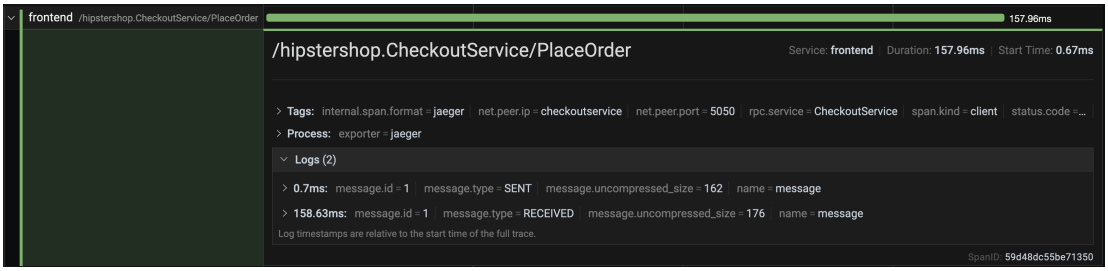

Figura 3.9: Maggiori informazioni fornite da uno span

Grazie a questa visualizzazione, si identifica l'esportatore implementato nei microservizi, in questo caso Jaeger, e la porta usata per comunicare con le altre componenti dell'applicazione. Inoltre, se disponibili ed implementati dallo sviluppatore nel codice sorgente dei microservizi, è possibile visualizzare gli attributi assegnati manualmente allo span.

Capitolo 4

Conclusioni

Si conclude il progetto di tesi con la verifica del raggiungimento degli obiettivi fissati come esigenze iniziali ed eventuali sviluppi futuri.

4.1 Raggiungimenti delle esigenze iniziali

La piattaforma di osservabilità disegnata e sviluppata con il presente progetto fornisce elevate prestazioni per monitorare le applicazioni a microservizi installate all'interno di un cluster Kubernetes. La fase di generazione consente di estrarre informazioni circa l'esecuzione dei processi direttamente dall'interno dell'applicazione, così da analizzare approfonditamente il comportamento del sistema. L'elaborazione e visualizzazione di questi dati, consente di progettare dashboards personalizzate ed interattive, di facile comprensione umana, che rappresentano graficamente un riassunto dettagliato dello stato interno del sistema monitorato.

Inoltre, grazie all'implementazione degli strumenti usati e alle tecniche di altissima affidabilità, la piattaforma risulta solita e stabile per essere eseguita in una infrastruttura aziendale e professionale.

4.2 Eventuali sviluppi futuri

Sebbene la piattaforma sviluppata assicuri un monitoraggio avanzato del sistema analizzato, è possibile continuare il suo sviluppo rendendola ancora più completa. Possibili implementazioni da aggiungere sono l'instrumentazione automatica di OpenTelemetry, l'allarmistica intelligente ed un insieme di funzionalità così da ridurre le differenze con Istio.

4.2.1 Espansione delle funzionalità per colmare le differenze con Istio

Rispetto ad OpenTelemetry, Istio risulta essere uno strumento molto più stabile, essendo sviluppato e disponibile da più tempo. Sebbene OpenTelemetry offre un monitoraggio più dettagliato, con la possibilità di espandere le informazioni manualmente, l'architettura di Istio consente a tutte le sue componenti di automatizzare tutti i processi di configurazione. Così, il team di sviluppo è incaricato solo di gestire dall'infrastruttura di controllo tutti i proxy configurati. Inoltre, grazie a Kiali, Istio mostra chiaramente la mappa della rete di microservizio tra loro in comunicazione.

Nel tentativo di colmare le differenze di OpenTelemetry rispetto ad Istio, l'installazione automatica di quest'ultimo e di tutte le sue componenti è molto simile all'instrumentazione manuale di OpenTelemetry, attualmente in fase di sviluppo. Al momento, invece, è assente un progetto compatibile con OpenTelemetry che fornisca una mappa dei microservizi simile a quella fornita da Kiali, così da semplificare la comprensione globale del sistema.

4.2.2 Instrumentazione automatica di OpenTelemetry

L'instrumentazione automatica fornita dalla comunità di OpenTelemetry è ancora un po' instabile ed è disponibile solo per applicazioni scritte in Java, quindi sarà necessario aspettare futuri aggiornamenti affinché si possa implementare in microservizi scritti con altri linguaggi di programmazione tutti i microservizi. Considerare l'instrumentazione automatica come un'alternativa a quella manuale è una definizione superficiale. Tuttavia il suo utilizzo può semplificare la fase di instrumentazione, evitando di configurare manualmente le librerie e gli esportatori, così da concedere al team di sviluppo di occuparsi solo dell'espansione delle metriche e tracciature generate automaticamente.

4.2.3 Espansione dell'allarmistica

Un'ulteriore possibile funzionalità da implementare nella piattaforma di osservabilità è l'allarmista intelligente. È già stata presentata nei capitoli precedenti e il suo utilizzo sarebbe di supporto al team di sviluppo in fase di monitoraggio. Studiando costantemente il comportamento del sistema grazie alle informazioni raccolte, l'allarmistica intelligente permetterebbe, con buona probabilità, di notificare in anticipo situazioni critiche, così da evitare malfunzionamenti. Sebbene esistano progetti molto piccoli e poco avanzati a riguardo, non è disponibile un software che offra un'esperienza d'uso ad alto livello.

Bibliografia

[1] Wayne Segar. *What is OpenTelemetry? Everything you wanted to know*. 2020. URL: https://www.dynatrace.com/news/blog/what-is-opentelemetry/.

[2] The Kubernetes Authors. *What is Kubernetes?*. 2021.

[3] The Kubernetes Authors. *Kubernetes Workloads*. 2021. URL: https://kubernetes.io/docs/concepts/workloads/.

[4] The Kubernetes Authors. *Services, Load Balancing, and Networking in Kubernetes*. 2021. URL: https://kubernetes.io/docs/concepts/services-networking/.

[5] Yuchen Zhong. *Getting started with Kubernetes and Docker with minikube*. 2019.

[6] Prometheus Authors. *Prometheus Configuration*. 2021. URL: https://prometheus.io/docs/prometheus/latest/configuration/configuration/.

[7] Red Hat. *Cosa sono i microservizi?*. 2021. URL: https://www.redhat.com/it/topics/microservices/what-are-microservices.

[8] Amazon Web Services. *Microservizi*. 202˙URL: https://aws.amazon.com/it/microservices/.

[9] Istio Authors. *What is Istio?*. 2020. URL: https://istio.io/latest/docs/concepts/what-is-istio/.

[10] Istio Authors. *Jaeger Introduction*. 2020. URL: https://www.jaegertracing.io/docs/1.22/.

[11] The OpenTelemetry Authors. *OpenTelemetry for Java*. 2021. URL: https://github.com/open-telemetry/opentelemetry-java.

[12] The OpenTelemetry Authors. *OpenTelemetry Documentation*. 2020. URL: https://opentelemetry.io/docs/.

[13] Fluentd Project. *What is Fluentd?Home*. 2021. URL: https://www.fluentd.org/architecture.

[14] Amazon Web Services. *Cos'è il cloud computing?*. 2021. URL: https://aws.amazon.com/it/what-is-cloud-computing/.

[15] Docker Community. *What is a Container?*. 2020. URL: https://www.docker.com/resources/what-container.

[16] Red Hat. *Come funzionano i container Docker?*. 2021. URL: https://www.redhat.com/it/topics/containers/what-is-docker.

[17] Amazon Web Services. *Cos'è Docker?*. 2021. URL: https://aws.amazon.com/it/docker/.

[18] Keith Rogers. *Black Box vs. White Box Monitoring: What You Need To Know*. 2018. URL: https://devops.com/black-box-vs-white-box-monitoring-what-you-need-to-know/.

[19] Rob Ewaschuk, Betsy Beyer. *Monitoring Distributed Systems*. 2017. URL: https://sre.google/sre-book/monitoring-distributed-systems/.

[20] Thanos Community. *Thanos Getting Started*. 2021. URL: https://thanos.io/tip/thanos/getting-started.md/.

[21] Elasticsearch B.V.. *Elasticsearch*. 2021. URL: https://www.elastic.co/elasticsearch/.

[22] Elasticsearch B.V.. *Kibana*. 2021. URL: https://www.elastic.co/kibana.

[23] Elasticsearch B.V.. *Logstash*. 2021. URL: https://www.elastic.co/logstash.

[24] Google Inc. *Online Boutique*. 2020. URL: `https://github.com/GoogleCloudPlatform/microservices-demo`.

[25] Grafana Labs. *The analytics platform for all your metrics*. 2021. URL: `https://grafana.com/grafana/`.

[26] The Fluent Bit Authors. *Log Processor and Forwarder*. 2021. URL: `https://fluentbit.io`.

[27] Grafana Labs. *Grafana Loki*. 2021. URL: `https://grafana.com/oss/loki/`.