UNIVERSITÀ POLITECNICA DELLE MARCHE
FACOLTÀ DI INGEGNERIA
Dipartimento di Ingegneria dell'Informazione
Corso di Laurea Triennale in Ingegneria Informatica e dell'Automazione

TESI DI LAUREA

Progettazione di un sistema distribuito per la pianificazione e l'esecuzione di exploits e la sottomissione di flags nelle competizioni CTF "Attack and Defense"

Design of a distributed system for exploit scheduling and execution, and flag submission in "Attack and Defense" CTF competitions

Advisor
Prof. Luca Spalazzi

Candidate
Samuele Perticarari

ACADEMIC YEAR 2021-2022

I computer sono incredibilmente veloci, accurati e stupidi.
Gli uomini sono incredibilmente lenti, inaccurati e intelligenti.
L'insieme dei due costituisce una forza incalcolabile.

Albert Einstein

Abstract - Italiano

In un'epoca in cui la connessione e la dipendenza dalle tecnologie digitali sono in costante aumento, la sicurezza informatica sta acquisendo sempre più importanza. Le competizioni CTF "Attack and Defense" sono una valida occasione per verificare e perfezionare le proprie abilità in materia di sicurezza informatica. In queste competizioni le capacità di pianificare ed eseguire rapidamente gli exploit e di sottomettere i flag sono cruciali per la vittoria del team. Con un numero sempre crescente di partecipanti e macchine da attaccare, diventa fondamentale disporre di un sistema distribuito efficiente e scalabile per gestire tali attività. Questa tesi si concentra sull'analisi e la progettazione di un sistema distribuito per la pianificazione e l'esecuzione di exploits e la sottomissione di flag nelle competizioni CTF "Attack and Defense". Verranno esplorate le sfide specifiche delle competizioni CTF e si presenterà un approccio per la progettazione di un sistema distribuito che sia in grado di affrontare simili sfide. Verrà, infine, valutato l'impatto delle scelte progettuali sulle prestazioni e sulla disponibilità del sistema.

Abstract - English

In an era where connection and dependence on digital technologies are constantly increasing, cyber security is gaining more and more importance. The CTF "Attack and Defense" competitions are a valid opportunity to test and perfect your cybersecurity skills. In these competitions, the ability to quickly schedule and execute exploits and to submit flags are crucial to team victory. With the ever-increasing number of participants and servers to attack, it becomes essential to have an efficient and scalable distributed system to manage these activities. This thesis focuses on analyzing and designing a distributed system for scheduling and executing exploits and flag submissions in "Attack and Defense" CTF competitions. It will be explored the specific challenges of CTF competitions and it will be presented an approach for designing a distributed system that can address these challenges. Finally, it will be evaluated the impact of the design choices on system performance and availability.

Keywords: Distributed Systems, Microservices, Kubernetes, Docker, Container, Cloud-Native, Serverless, Cassandra, Knative, RabbitMQ, Rook, TiKV, Event-Driven Architecture, Message Broker, Scalability, High Availability, Performance, Capture The Flag.

Indice

Introduzione **1**

1 Panoramica sui sistemi distribuiti **3**
1.1 Sistemi distribuiti . . . 3
1.1.1 Vantaggi . . . 4
1.1.2 Svantaggi . . . 4
1.2 Teorema CAP . . . 5
1.3 Scalabilità . . . 5
1.3.1 Scalabilità dell'applicazione . . . 5
1.3.2 Scalabilità di un sistema di computazione . . . 8
1.4 Architettura monolitica . . . 9
1.4.1 Vantaggi . . . 10
1.4.2 Svantaggi . . . 10
1.4.3 Dal monolite ai microservizi . . . 11
1.5 Microservizi . . . 11
1.5.1 Vantaggi . . . 12
1.5.2 Svantaggi . . . 14
1.6 Event-Driven Architecture . . . 15

2 Analisi del sistema **17**
2.1 Introduzione e contesto . . . 17
2.1.1 Le competizioni CTF . . . 17
2.1.2 Le competizioni CTF "Attack and Defense" di CyberChallenge.IT . . . 18
2.1.3 Lo scopo e le caratteristiche del sistema . . . 19
2.1.4 Contesto di esecuzione del sistema . . . 19
2.2 Approccio alla progettazione del sistema distribuito . . . 20
2.3 Analisi dei requisiti . . . 20
2.4 Requisiti funzionali . . . 21
2.4.1 Requisiti funzionali utente non autenticato . . . 21
2.4.2 Requisiti funzionali utente autenticato . . . 22
2.4.3 Requisiti funzionali sistema automatizzato . . . 23
2.5 Requisiti non funzionali . . . 23
2.5.1 Performance . . . 24
2.5.2 Scalabilità . . . 24
2.5.3 Affidabilità . . . 25
2.5.4 Manutenibilità . . . 25

2.5.5 Disponibilità . . . 25
2.5.6 Sicurezza . . . 25
2.5.7 Portabilità . . . 25
2.5.8 Compatibilità . . . 26
2.6 Analisi dei volumi . . . 26
2.6.1 Requisiti di traffico . . . 26
2.6.2 Conclusioni . . . 29

3 Progettazione ad alto livello **32**
3.1 Vista concettuale del sistema . . . 32
3.2 Architettura ad alto livello . . . 33
3.2.1 API Gateway . . . 36
3.2.2 Authentication Service . . . 37
3.2.3 Frontend Service . . . 38
3.2.4 Exploits Service . . . 39
3.2.5 Services Service . . . 39
3.2.6 Vulnboxes Service . . . 39
3.2.7 Jobs Service . . . 39
3.2.8 Job Scheduler Service . . . 40
3.2.9 Job Executor Service . . . 40
3.2.10 Flags Service . . . 41
3.3 Scelta delle tecnologie . . . 41
3.3.1 Kubernetes . . . 42
3.3.2 Knative . . . 43
3.3.3 Apache Cassandra . . . 44
3.3.4 TiKV . . . 45
3.3.5 RabbitMQ . . . 45
3.3.6 Rook . . . 45
3.3.7 Emissary-Ingress . . . 46

4 Progettazione a basso livello **47**
4.1 Authentication Microservice . . . 49
4.1.1 AuthenticateUser Function . . . 49
4.2 Frontend Microservice . . . 50
4.2.1 FrontendService . . . 51
4.3 Vulnboxes Microservice . . . 52
4.3.1 GetVulnboxes Function . . . 53
4.3.2 GetVulnboxById Function . . . 53
4.3.3 UpdateVulnboxById Function . . . 54
4.3.4 DeleteVulnboxById Function . . . 54
4.3.5 CreateVulnbox Function . . . 54
4.4 Services Microservice . . . 56
4.4.1 GetServices Function . . . 57
4.4.2 GetServiceById Function . . . 57
4.4.3 UpdateServiceById Function . . . 58
4.4.4 DeleteServiceById Function . . . 58
4.4.5 CreateService Function . . . 58
4.5 Exploits Microservice . . . 60

4.5.1 GetExploits Function 61
4.5.2 GetExploitById Function 61
4.5.3 UpdateExploitById Function 62
4.5.4 DeleteExploitById Function 62
4.5.5 CreateExploit Function 62
4.6 Jobs Microservice 65
4.6.1 ConfigurationUpdater Function 66
4.6.2 JobsCreator Service 67
4.6.3 CreateJob Function 67
4.6.4 CustomJobsCreator Function 67
4.6.5 GetJobById Function 67
4.6.6 GetJobByRoundNumber Function 68
4.6.7 UpdateJobById Function 68
4.6.8 GetJobsToScheduleByShard Function 68
4.7 Job Scheduler Microservice 75
4.7.1 JobScheduler Service 76
4.7.2 JobDLQEnqueuer Service 76
4.8 Job Executor Microservice 79
4.8.1 JobExecutor Service 80
4.8.2 FlagsDLQEnqueuer Service 80
4.8.3 JobsWorkerAutoscaler Function 81
4.9 Flags Microservice 83
4.9.1 FlagSubmitter Service 83
4.9.2 FlagSubmitterAutoscaler Function 84
4.10 Emissary-Ingress - API Gateway 85

5 Conclusioni **86**
5.1 Sviluppi futuri 89

Bibliografia **90**

Elenco delle figure

1.1 Il cubo di scala definisce tre modi distinti per ridimensionare un'applicazione: il carico di ridimensionamento dell'asse X bilancia le richieste tra più istanze identiche; Il ridimensionamento dell'asse Z instrada le richieste in base a un attributo della richiesta; L'asse Y scompone funzionalmente un'applicazione in servizi. Fonte: [8] 6
1.2 Il ridimensionamento dell'asse X esegue più istanze identiche dell'applicazione monolitica dietro un sistema di bilanciamento del carico. Fonte: [8] 7
1.3 Il ridimensionamento dell'asse Y suddivide l'applicazione in un insieme di servizi. Ogni servizio è responsabile di una particolare funzione. Un servizio viene ridimensionato utilizzando il ridimensionamento dell'asse X e, possibilmente, il ridimensionamento dell'asse Z. Fonte: [8] 7
1.4 Il ridimensionamento dell'asse Z esegue più istanze identiche dell'applicazione monolitica dietro un router, che instrada le richieste in base a un attributo di richiesta. Ogni istanza è responsabile di un sottoinsieme di dati. Fonte: [8] . . 8
1.5 Scalabilità verticale di un sistema di computazione 8
1.6 Scalabilità orizzontale di un sistema di computazione 9
1.7 Suddivisione di un'applicazione monolitica in microservizi. (Fonte: [5]) 11
1.8 Architettura a microservizi. Fonte: [11] 13
1.9 Un esempio grafico di dipendenze in un'architettura a microservizi. Fonte: [22] 15
1.10 Funzionamento ad alto livello di un architettura guidata ad eventi 15

2.1 Schema di rete della competizione Attack and Defense nella sfida nazionale di CyberChallenge.IT. 19

3.1 Vista concettuale del sistema distribuito. 33
3.2 Architettura ad alto livello del sistema distribuito. 35
3.3 Approccio Low Trust nei sistemi distribuiti. Immagine adattata da [17]. 37
3.4 Approccio Zero Trust nei sistemi distribuiti. Immagine adattata da [17]. 38

4.1 Architettura a basso livello del microservizio: Authentication Service 49
4.2 Architettura a basso livello del microservizio: Frontend Service 50
4.3 Architettura a basso livello del microservizio: Vulnboxes Service 52
4.4 Architettura a basso livello del microservizio: Services Microsevice 56
4.5 Architettura a basso livello del microservizio: Exploits Microsevice 60
4.6 Architettura a basso livello del microservizio: Jobs Microsevice 65
4.7 CQL per creare il Jobs Database 73
4.8 CQL per creare dei job nel Jobs Database 74

4.9 CQL per filtrare i job per numero di round e per shard su Jobs Database . . . 74
4.10 Architettura a basso livello del microservizio: Job Scheduler Microsevice . . . 75
4.11 CQL per creare il Jobs DLQ Database 78
4.12 CQL leggere i job per shard e cancellarne uno su Jobs DLQ Database 78
4.13 Architettura a basso livello del microservizio: Job Executor Microsevice 79
4.14 CQL per creare il Flags DLQ Database 82
4.15 CQL leggere i flag per shard dal Flags DLQ Database 82
4.16 Architettura a basso livello del microservizio: Flags Microsevice 83

Introduzione

La sicurezza informatica è una sfida costante per le aziende e le organizzazioni in tutto il mondo. Con l'evoluzione della tecnologia e della minaccia informatica, è diventato sempre più importante essere preparati a difendere i propri sistemi e dati. Un modo per mettere alla prova le proprie capacità di difesa e attacco informatico è partecipare a competizioni di sicurezza informatica, come le CTF "Attack and Defense". In queste competizioni, i team cercano di pianificare ed eseguire rapidamente gli exploit e di sottomettere i flag per ottenere il massimo punteggio possibile. Per far fronte alla quantità crescente di macchine da attaccare e alla necessità di una gestione efficiente delle attività, diventa cruciale disporre di un sistema distribuito scalabile e affidabile per la pianificazione e l'esecuzione di exploits e la sottomissione di flag.

La progettazione di un sistema distribuito è un processo complesso che richiede un'analisi accurata e attenta dei requisiti del sistema. Il processo di progettazione sarà suddiviso in diverse fasi, tra cui l'analisi dei requisiti, la progettazione ad alto livello, la scelta delle tecnologie e la progettazione a basso livello. Questa tesi si concentra sull'analisi e la progettazione di un sistema distribuito per la gestione di queste attività nelle competizioni CTF "Attack and Defense". La fase di analisi dei requisiti sarà cruciale per comprendere le esigenze del sistema e per determinare come queste possono essere soddisfatte in modo efficiente. L'analisi dei requisiti funzionali e non funzionali fornirà un quadro completo delle funzionalità che il sistema dovrà fornire e delle prestazioni che dovrà raggiungere. La progettazione ad alto livello permetterà di identificare la struttura del sistema e le sue componenti principali, mentre la scelta delle tecnologie sarà fondamentale per garantire che il sistema sia scalabile, performante e altamente disponibile. Infine, la progettazione a basso livello permetterà di definire i dettagli tecnici del sistema e di definire le soluzioni progettuali.

La presente tesi è composta da cinque capitoli strutturati come di seguito specificato:

1. **Panoramica sui sistemi distribuiti**: In questo capitolo si cercherà di descrivere i sistemi distribuiti, con particolare riferimento all'architettura monolitica, ai microservizi e all'architettura guidata ad eventi (EDA[1]). Verrà inoltre enfatizzata la scalabilità delle applicazioni e dei sistemi distribuiti. Dell'architettura monolitica si descriverà la sua struttura e le caratteristiche del monolite, enfatizzando i vantaggi e gli svantaggi. Verrà evidenziata la soluzione alle limitazioni dei sistemi monolitici con l'evoluzione in un'architettura a microservizi. Per quanto riguarda i microservizi verranno evidenziati i molteplici vantaggi di tale architettura e le conseguenze che porta l'implementazione di tale architettura. Infine viene introdotta l'architettura guidata ad eventi, definendo le sue caratteristiche e la sua struttura.

[1] EDA: Event-Driven Architecture (architettura guidata ad eventi).

2. **Analisi del sistema**: In questo capitolo verranno introdotte le competizioni CTF, con particolare riferimento alle CTF Attack and Defense. Verranno inoltre descritti il contesto in cui dovrà essere eseguito il sistema e le caratteristiche che dovrà possedere. Successivamente verrà presentato l'approccio utilizzato per la progettazione del sistema. Verranno poi analizzati i requisiti, con particolare riferimento all'analisi dei requisiti funzionali, non funzionali e dei volumi. Nell'analisi dei requisiti funzionali verranno analizzate le funzionalità che il sistema dovrà fornire. Nell'analisi dei requisiti non funzionali, invece, verranno analizzate le caratteristiche che il sistema dovrà avere. Tramite l'analisi dei volumi, si trarranno delle conclusioni che verranno utilizzate in fase di progettazione del sistema.

3. **Progettazione ad alto livello**: In questo capitolo si cercherà di descrivere l'architettura ad alto livello del sistema. Tramite lo studio della vista concettuale saranno poi analizzate le tre tipologie di servizi del sistema: i servizi pubblici, privati e interni. Per ogni tipologia di servizio saranno studiati alto livello i componenti principali del sistema. Verranno introdotte le funzionalità e le caratteristiche dei componenti principali del sistema. Infine verranno introdotte brevemente le varie soluzioni in termini di orchestrazione, computazione, storicizzazione e messaggistica che verranno utilizzate nel sistema.

4. **Progettazione a basso livello**: Nel capitolo in questione, viene trattata la progettazione a basso livello del sistema distribuito. si analizzano tutti i microservizi del sistema, tra cui: Authentication Service, Frontend Service, Exploits Service, Services Service, Vulnboxes Service, Jobs Service, Job Scheduler Service, Job Executor Service, Flags Service e API Gateway. Per ogni microservizio saranno analizzate le varie componenti che ne fanno parte, le caratteristiche delle componenti, le strategie di progettazione, gli eventi prodotti e consumati e la modellazione dei dati del database.

5. **Conclusioni**: In questo ultimo capitolo vengono introdotte le conclusioni che si possono trarre dallo sviluppo della tesi. Verrà ricordato il processo seguito per analizzare e progettare il sistema distribuito, con particolare riferimento alle soluzioni tecniche adottate in fase di progettazione per soddisfare i requisiti non funzionali descritti, nella fase di analisi, al capitolo 2.3. Vengono infine forniti degli spunti per possibili sviluppi futuri.

CAPITOLO 1

Panoramica sui sistemi distribuiti

In questo capitolo si cercherà di descrivere i sistemi distribuiti, con particolare riferimento all'architettura monolitica, ai microservizi e all'architettura guidata ad eventi (EDA[1]). Verrà inoltre enfatizzata la scalabilità delle applicazioni e dei sistemi distribuiti. Dell'architettura monolitica si descriverà la sua struttura e le caratteristiche del monolite, enfatizzando i vantaggi e gli svantaggi. Verrà evidenziata la soluzione alle limitazioni dei sistemi monolitici con l'evoluzione in un'architettura a microservizi. Per quanto riguarda i microservizi verranno evidenziati i molteplici vantaggi di tale architettura e le conseguenze che porta l'implementazione di tale architettura. Infine viene introdotta l'architettura guidata ad eventi, definendo le sue caratteristiche e la sua struttura.

1.1 Sistemi distribuiti

Inizialmente, le applicazioni erano gestite da sistemi centralizzati con un solo server che gestiva tutte le richieste e i dati, approccio che, tuttavia, ha rapidamente mostrato i suoi limiti; ben presto ci si rese conto che tali sistemi avevano delle limitazioni in termini di capacità computazionale, memoria e difficoltà nella gestione del carico di lavoro. Con l'aumento delle esigenze dei clienti e la crescita dei dati, è diventato sempre più evidente che questi sistemi centralizzati non erano più in grado di gestire la quantità di richieste e di dati in modo efficiente. Di conseguenza, si è passati a sistemi distribuiti in cui le risorse e i dati sono distribuiti su più nodi, permettendo una maggiore scalabilità e resilienza.

Un simile cambiamento ha portato a una maggiore flessibilità nell'architettura delle applicazioni e ha permesso di soddisfare le crescenti esigenze dei clienti. Infatti, i sistemi distribuiti sono dei sistemi in cui i componenti hardware o software, situati nei computer in rete, comunicano e coordinano le loro azioni attraverso uno scambio di messaggi [20]. Le componenti del sistema distribuito, pur essendo fisicamente distribuite su più dispositivi o nodi, sembrano funzionare come un'unica entità.

La computazione di un workload[2] che sfrutta un sistema distribuito per operazioni di elaborazione parallela su più nodi è detta computazione distribuita. Il calcolo distribuito è il metodo per far lavorare insieme più computer per risolvere un problema comune; esso fa apparire una rete di computer come un singolo potente computer che fornisce risorse su larga scala per affrontare sfide complesse. [6]

In tale maniera è possibile condividere le risorse tra più dispositivi per offrire una maggiore scalabilità e tolleranza ai guasti, migliorando le performance e la resilienza del sistema.

[1] EDA: Event-Driven Architecture (architettura guidata ad eventi).

[2] Il termine workload è utilizzato per indicare un carico di lavoro, composto da diverse operazioni.

Le caratteristiche di un sistema distribuito includono, quindi, la condivisione delle risorse, la tolleranza ai guasti e la scalabilità.

1.1.1 Vantaggi

I sistemi distribuiti offrono molti vantaggi [6] rispetto all'elaborazione a sistema centralizzato, come per esempio:

- **Scalabilità**: i sistemi distribuiti possono crescere in termini di capacità di computazione e memoria con il carico di lavoro e i requisiti di business. È possibile aggiungere, quando necessario, più dispositivi di elaborazione al sistema.
- **Elevata disponibilità e tolleranza ai guasti**: i sistemi distribuiti possono garantire un'alta disponibilità delle risorse e dei servizi che offrono; quindi, attraverso un'architettura progettata correttamente, non andrà in crash se uno dei nodi si guasterà, ma continuerà a funzionare.
- **Consistenza**: i computer in un sistema distribuito condividono informazioni e possono replicare i dati tra di loro, pertanto, è possibile garantire la tolleranza ai guasti senza compromettere la coerenza dei dati.
- **Ottime performance**: i sistemi distribuiti possono aumentare le prestazioni del sistema, in quanto utilizzano più risorse per eseguire le operazioni di elaborazione parallela.
- **Efficienza**: I sistemi distribuiti offrono prestazioni migliori con un utilizzo ottimale delle risorse dell'hardware sottostante, di conseguenza, si può gestire un qualsiasi carico di lavoro senza preoccuparsi dei guasti del sistema dovuti a picchi di volume.
- **Economie di scala**: i sistemi distribuiti consentono di raggiungere economie di scala, in quanto le risorse possono essere condivise tra più applicazioni o utenti con una riduzione dei costi di elaborazione e di memoria.
- **Trasparenza**: i sistemi di elaborazione distribuiti forniscono una separazione logica tra l'utente e i dispositivi fisici. Si può interagire con il sistema come se fosse un singolo computer senza dover installare e configurare le singole macchine. È possibile disporre di hardware, middleware, software e sistemi operativi diversi che lavorano insieme per far funzionare il sistema senza problemi.

1.1.2 Svantaggi

I sistemi distribuiti, seppur molto vantaggiosi sotto numerosi punti di vista, hanno delle caratteristiche [20] svantaggiose:

- **Complessità**: i sistemi distribuiti sono più complessi da progettare, implementare e gestire rispetto ai sistemi centralizzati. Ciò può aumentare i costi e il tempo necessario per la loro manutenzione.
- **Latenza**: la comunicazione tra i nodi del sistema avviene tramite una rete di comunicazione, la quale può causare latenze più elevate rispetto ai sistemi centralizzati, influendo sulle prestazioni dell'applicazione.

- **Sincronizzazione dei dati**: garantire la sincronizzazione dei dati tra i nodi del sistema può essere complesso e richiedere un'attenta progettazione.
- **Affidabilità**: poiché i sistemi distribuiti sono composti da più componenti, la loro affidabilità dipende dalla disponibilità di tutti i componenti. Se un componente non è disponibile, il sistema potrebbe non funzionare correttamente.
- **Costi**: i sistemi distribuiti possono essere più costosi da progettare, implementare e gestire rispetto ai sistemi centralizzati.
- **Debugging e testing complesso**: il testing di funzionalità e il debugging dei problemi in un sistema distribuito è più complesso rispetto ad un sistema centralizzato, poiché i problemi possono verificarsi in qualsiasi punto del sistema.

1.2 Teorema CAP

Il teorema CAP, noto anche come congettura di Brewer nell'informatica teorica [21], sostiene che un sistema informatico distribuito non può garantire contemporaneamente tutte e tre le seguenti proprietà:

- **Coerenza (Consistency)**: tutti i nodi del sistema hanno la stessa visione dei dati.
- **Disponibilità (Availability)**: tutte le richieste ricevono una risposta, anche in caso di guasti.
- **Tolleranza di partizione (Partition tolerance)**: il sistema continua a funzionare anche quando alcuni nodi non sono raggiungibili a causa di problemi di rete.

In sintesi, il teorema CAP, sostiene che in un sistema distribuito si possono scegliere al più 2 delle proprietà sopra citate. Nei sistemi distribuiti, in particolare, non è possibile garantire la partition tolerance a causa dell'impossibilità di poter escludere guasti a livello di rete; quindi, si è costretti a scegliere tra Coerenza e Disponibilità. Si avrà quindi un sistema distribuito di tipo CP, cioè coerente, ma non sempre disponibile oppure un sistema AP, cioè un sistema altamente disponibile, ma non sempre consistente. Sarà quindi necessario fare un compromesso e scegliere se avere un servizio altamente disponibile o fortemente consistente.

1.3 Scalabilità

La scalabilità è la capacità di un sistema di gestire un aumento del carico senza compromettere le prestazioni. E' possibile dividere la scalabilità in due categorie: la scalabilità dell'applicazione [8] e la scalabilità di un sistema di computazione.

1.3.1 Scalabilità dell'applicazione

Chris Richardson, in "Microservices Patterns" [8], tramite l'immagine 1.1, descrive in quanti modi è possibile scalare un'applicazione.

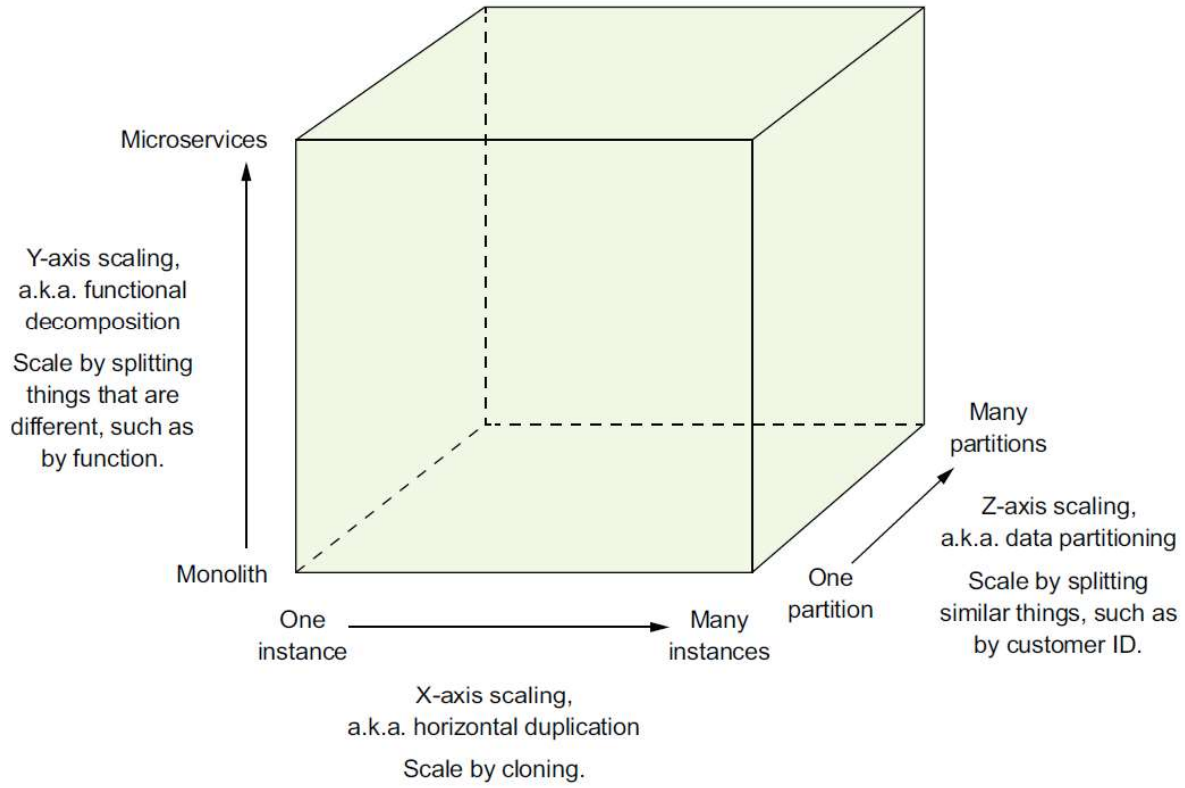

Figura 1.1: Il cubo di scala definisce tre modi distinti per ridimensionare un'applicazione: il carico di ridimensionamento dell'asse X bilancia le richieste tra più istanze identiche; Il ridimensionamento dell'asse Z instrada le richieste in base a un attributo della richiesta; L'asse Y scompone funzionalmente un'applicazione in servizi. Fonte: [8]

Richardson spiega che è possibile scalare un'applicazione in 3 modi distinti:

1. **Scalare il numero di istanze dell'applicazione**: è un modo comune per ridimensionare un'applicazione monolitica. Eseguendo più istanze dell'applicazione c'è la necessità di un sistema di bilanciamento del carico capace di inoltrare le richieste all'istanza desiderata. Il bilanciamento del carico distribuisce le richieste tra le N istanze identiche dell'applicazione. Questo è un ottimo modo per migliorare la capacità e la disponibilità di un'applicazione (Vedi Fig. 1.2).

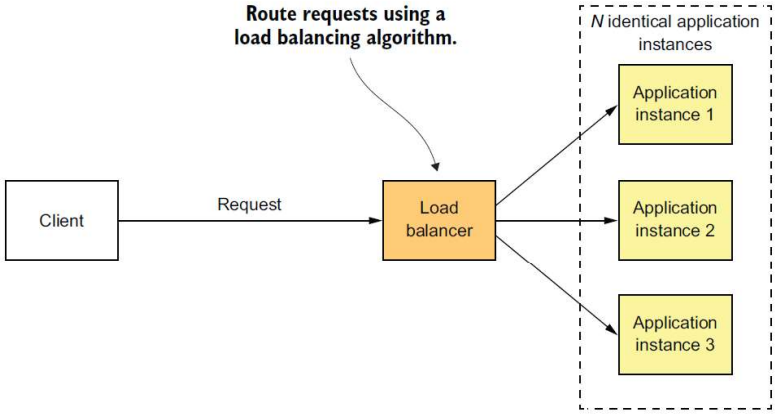

Figura 1.2: Il ridimensionamento dell'asse X esegue più istanze identiche dell'applicazione monolitica dietro un sistema di bilanciamento del carico. Fonte: [8]

2. **Scalare per decomposizione funzionale**: suddividendo l'applicazione in un insieme di servizi. Ogni servizio sarà una "mini" applicazione che implementa funzionalità strettamente mirate (Vedi Fig. 1.3).

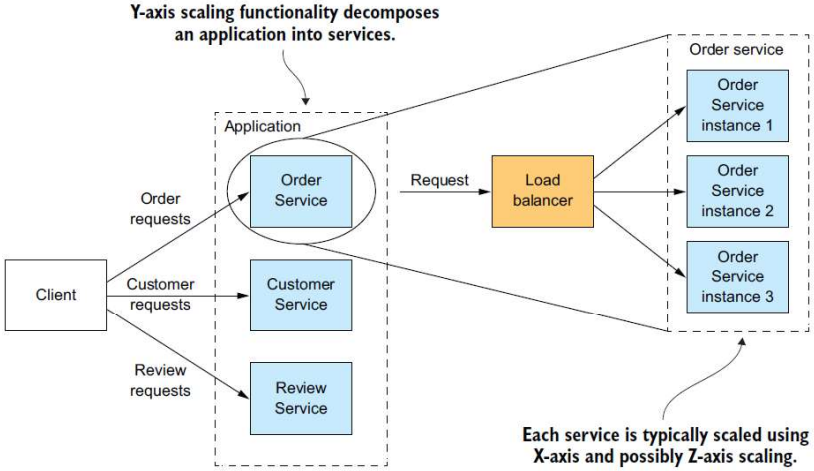

Figura 1.3: Il ridimensionamento dell'asse Y suddivide l'applicazione in un insieme di servizi. Ogni servizio è responsabile di una particolare funzione. Un servizio viene ridimensionato utilizzando il ridimensionamento dell'asse X e, possibilmente, il ridimensionamento dell'asse Z. Fonte: [8]

3. **Scalare il numero di partizioni dei dati**: partizionando i dati in un numero maggiore di partizioni è possibile assegnare una porzione di dati ad ogni istanza, rendendo l'applicazione più performante e tollerante ai guasti. E' necessario un router che intercetta le richieste e che, tramite un attributo di richiesta, lo inoltri all'istanza appropriata. Scalare il numero di partizioni dei dati è un ottimo modo per ridimensionare un'applicazione per gestire l'aumento dei volumi di transazioni e dei dati (Vedi Fig. 1.4).

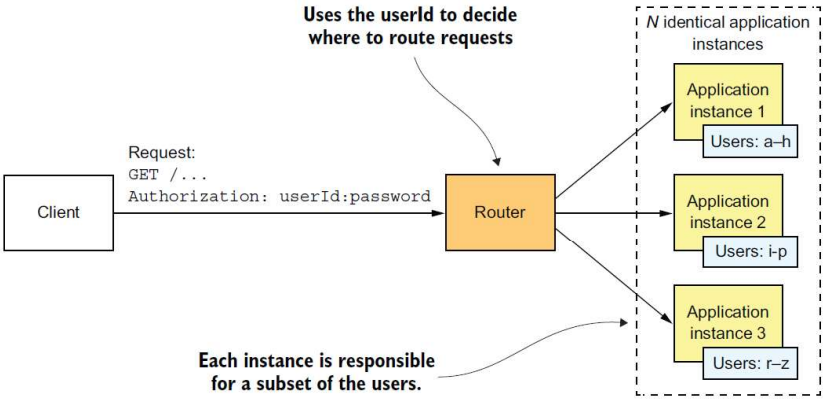

Figura 1.4: Il ridimensionamento dell'asse Z esegue più istanze identiche dell'applicazione monolitica dietro un router, che instrada le richieste in base a un attributo di richiesta. Ogni istanza è responsabile di un sottoinsieme di dati. Fonte: [8]

1.3.2 Scalabilità di un sistema di computazione

Un sistema di computazione è composto da nodi computazionali con delle risorse limitate in termini di memoria, CPU[3] e spazio di archiviazione. Il sistema, se dovrà servire un numero crescente di utenti e carico, avrà bisogno di più risorse per soddisfare le richieste. Tali necessità possono essere soddisfatte scalando il sistema di computazione in due modi:

1. **Scalabilità verticale**: la scalabilità verticale si ottiene aumentando le risorse (ad esempio la memoria e le performance del processore) di un singolo nodo; ciò consente di aumentare la capacità di elaborazione di un singolo nodo, ma può limitare la capacità di gestire un carico crescente. La scalabilità verticale è utile per sistemi ad alto carico di elaborazione e consente di gestire un carico crescente, ma limitato, su un singolo nodo senza compromettere le prestazioni. [3]

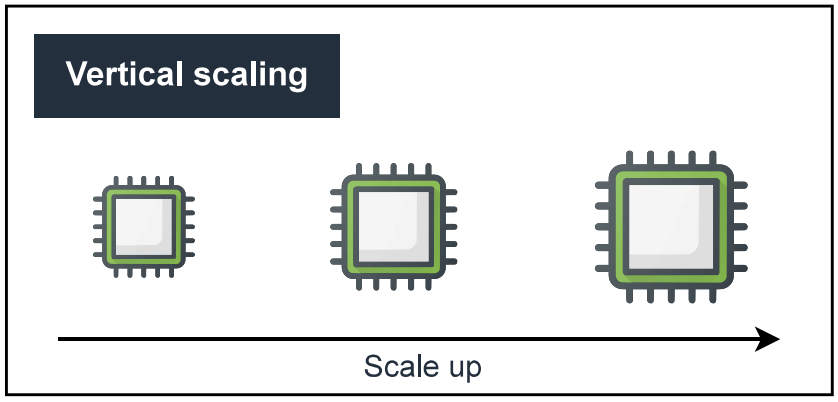

Figura 1.5: Scalabilità verticale di un sistema di computazione

2. **Scalabilità orizzontale**: la scalabilità orizzontale si ottiene aggiungendo più nodi al sistema; ciò consente di distribuire il carico tra più macchine e aumentare la capacità del sistema. La scalabilità orizzontale è utile per sistemi ad alto traffico e consente

[3]CPU (Central Processing Unit): unità centrale di elaborazione o processore centrale.

di gestire un carico crescente senza compromettere le prestazioni. I sistemi scalabili orizzontalmente sono spesso in grado di superare, in termini di prestazioni, i sistemi scalati esclusivamente in maniera verticale consentendo l'esecuzione parallela dei carichi di lavoro e distribuendoli su molti computer diversi. [3]

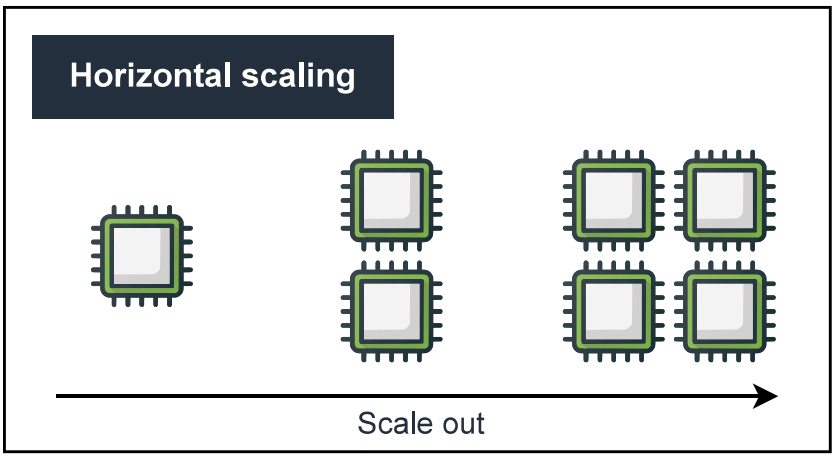

Figura 1.6: Scalabilità orizzontale di un sistema di computazione

Esempio di scalabilità orizzontale e verticale

E' bene evidenziare che le due tipologie di scalabilità non sono mutuamente esclusive, ma possono essere combinate per ottenere dei vantaggi migliori in termini di performance. Si pensi ad un sistema distribuito che permette agli utilizzatori di caricare immagini in alta qualità e di scaricare la thumbnail dell'immagine caricata. Il sistema dovrà elaborare delle immagini in alta qualità tramite un software di processamento delle immagini, ricavare una thumbnail e inviarla all'utente per il download. Il sistema avrà quindi bisogno di nodi di elaborazione con una CPU performante e una memoria volatile di discrete dimensioni, in modo tale da velocizzare l'elaborazione di una singola immagine; così è possibile velocizzare ogni singola operazione, ma per un numero crescente di utenti sarà necessario aumentare il numero di nodi che elaborano le immagini.

In questo esempio il sistema dovrà scalare verticalmente per velocizzare l'elaborazione di una singola immagine e scalare orizzontalmente per elaborare le richieste di più utenti contemporaneamente, senza un degrado delle performance.

1.4 Architettura monolitica

L'architettura monolitica è un pattern di progettazione software in cui tutte le funzionalità e i componenti di un'applicazione sono integrati all'interno di un unico pacchetto o modulo, spesso chiamato *"monolite"*. Il monolite è il sistema software che include l'interfaccia utente, il database e la business-logic; è solitamente caratterizzato da un codice sorgente di grandi dimensioni e mantenuto da uno stesso team di sviluppatori.

Tale struttura è considerata come la più semplice tra le varie architetture software, presente sin dai primi programmi su mainframe; per alcuni decenni è stata l'unica architettura software utilizzata. Questo tipo di architettura, infatti, è spesso utilizzata nei sistemi legacy[4]

[4]Un sistema legacy è un sistema informativo che utilizza tecnologie e processi obsoleti.

e nei sistemi di piccole dimensioni, poiché è più semplice da sviluppare rispetto ai sistemi distribuiti.

Tuttavia, i sistemi monolitici possono diventare complessi e difficili da scalare quando le esigenze del sistema cambiano; infatti per apportare una modifica all'applicazione, occorre aggiornare l'intero modulo, compilando e distribuendo una versione aggiornata del servizio. Tale procedura rende gli aggiornamenti restrittivi e dispendiosi in termini di tempo.

1.4.1 Vantaggi

Le applicazioni monolitiche hanno delle caratteristiche che le rendono delle buone candidate per certi casi d'uso. I vantaggi [8, 12] dei sistemi monolitici sono:

- **Facilità di sviluppo**: i sistemi monolitici possono essere più semplici da sviluppare poiché tutto il codice è contenuto in un unico luogo.
- **Facilità di debugging e testing**: il test e debug dell'applicazione può essere più semplice in un sistema monolitico, poiché le chiamate tra le varie parti del sistema sono semplici da verificare e da testare.
- **Minore complessità nella gestione delle dipendenze**: in un sistema monolitico, le dipendenze tra le varie parti del sistema sono più facili da gestire poiché tutto il codice è contenuto in un unico progetto.
- **Minori costi di infrastruttura**: i sistemi monolitici possono richiedere meno risorse di sistemi distribuiti poiché non è necessario gestire un gran numero di servizi indipendenti.
- **Maggiore sicurezza**: i sistemi monolitici possono essere più sicuri poiché le chiamate tra le varie parti del sistema sono più facili da controllare.

1.4.2 Svantaggi

L'architettura monolitica è stata per decenni il pattern di riferimento per lo sviluppo di applicazioni software, ma le sue caratteristiche possono essere problematiche quando lo scopo dell'applicazione cambia, sia in termini di volume di traffico, sia in termini di requisiti di business. I sistemi monolitici hanno diversi svantaggi [8, 12], che rendono l'architettura monolitica non adatta allo sviluppo di applicazioni moderne:

- **Grande quantità di codice**: tutte le funzionalità e i componenti dell'applicazione sono integrati in un unico pacchetto di codice, che può diventare molto grande e complesso.
- **Dipendenza tra componenti**: tutti i componenti dell'applicazione sono strettamente interconnessi, il che significa che un cambiamento in una parte dell'applicazione può avere un impatto sull'intero sistema.
- **Difficoltà nella scalabilità**: poiché tutti i componenti sono integrati in un unico pacchetto, può essere difficile scalare singoli componenti per gestire un aumento del traffico.
- **Difficoltà nella manutenzione**: la grande dimensione e la complessità del codice possono rendere difficile la manutenzione e l'aggiornamento dell'applicazione.

- **Stack di tecnologie fissato**: un'architettura monolitica rende difficoltoso aggiornare lo stack di tecnologie dell'applicazione che si è scelto all'inizio dello sviluppo. Può quindi essere difficile adottare in modo incrementale una tecnologia più recente.
- **Poca flessibilità**: i monoliti sono poco flessibili, in quanto ogni modifica richiede un aggiornamento completo dell'applicazione.
- **Poca tolleranza ai fallimenti**: un errore in un componente del sistema monolitico può causare l'interruzione dell'intero sistema.
- **Difficoltà nel test**: poiché tutti i componenti sono integrati, può essere difficile testare singoli componenti senza influire sull'intero sistema.

1.4.3 Dal monolite ai microservizi

Negli anni la necessità di creare sistemi software più flessibili, scalabili e facili da mantenere ha fatto evolvere l'architettura monolitica, in quanto i sistemi monolitici possono diventare complessi e difficili da gestire man mano che crescono e si evolvono. Si è cercata una soluzione architetturale che fosse in grado di risolvere le limitazioni dei sistemi monolitici. [12]

L'architettura si è evoluta in un sistema con componenti indipendenti tra loro e più semplici: l'architettura a microservizi (Cap. 1.5).

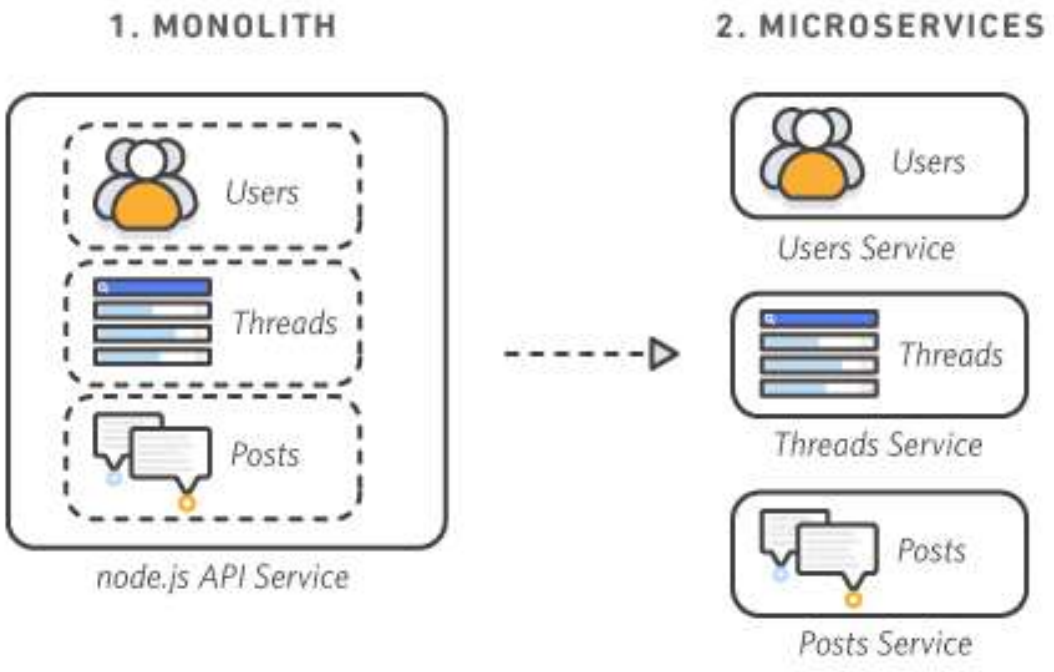

Figura 1.7: Suddivisione di un'applicazione monolitica in microservizi. (Fonte: [5])

1.5 Microservizi

I microservizi sono un approccio architetturale e organizzativo allo sviluppo del software in cui, il software, è composto da piccoli servizi indipendenti che comunicano tramite API ben

definite. Questi servizi sono in genere di proprietà di piccoli team autonomi. Le architetture di microservizi rendono le applicazioni più facili da scalare e più veloci da sviluppare, consentendo l'innovazione e accelerando il time-to-market[5] per le nuove funzionalità. [5]

I microservizi possono essere sviluppati, testati e distribuiti in modo indipendente, il che li rende più facili da gestire, testare e scalare; ciò consentirà quindi uno sviluppo e una distribuzione rapida e frequente. I microservizi, tra loro, non sono accoppiati strettamente, ma in modo lasco. Due servizi sono accoppiati strettamente se c'è un'invocazione sincrona tra essi; sono invece accoppiati in modo lasco se interagiscono in maniera asincrona.

Tale caratteristica permette a un team di lavorare in modo indipendente su un microservizio, senza essere influenzato dalle modifiche degli altri servizi. L'accoppiamento lasco tra i microservizi consente una distribuzione indipendente di codice. Ciò permette ad un team di distribuire il proprio servizio senza doversi coordinare con altri team; se ci sono dipendenze strette tra microservizi è necessario, invece, un coordinamento per il rilascio dei servizi interessati quando si modifica il codice che interagisce con l'altro servizio. [11]

I microservizi, avendo uno scopo ben definito ed essendo di dimensioni ridotte, consentono di essere sviluppati da un piccolo team, il che riduce i tempi di comunicazione dei grandi team di sviluppo, riducendo i tempi di sviluppo. Tramite l'implementazione di un'architettura a microservizi, è possibile utilizzare tecnologie diverse per ogni servizio, ciò consente di fronteggiare l'eterogeneità delle capacità tecniche dei team di sviluppo e di sfruttare al meglio le caratteristiche di ogni tecnologia per lo scopo. L'architettura a microservizi, inoltre, rende le applicazioni più facili da scalare e più resistenti ai guasti, poiché i singoli servizi possono essere aggiornati o sostituiti senza interferire con il funzionamento dell'intera applicazione [11].

Ogni servizio ha un proprio database, ciò consente di utilizzare un database con caratteristiche differenti in ogni microservizio. Durante lo sviluppo, gli sviluppatori, possono modificare lo schema di un servizio senza doversi coordinare con gli sviluppatori che lavorano su altri servizi. In fase di esecuzione, i servizi sono isolati l'uno dall'altro, ad esempio un servizio non verrà mai bloccato a causa di un altro servizio che tiene un blocco in una o più righe del database.

La comunicazione tra i servizi avviene utilizzando protocolli sincroni come REST/SOAP o protocolli asincroni come AMQP. L'architettura REST (REpresentational State Transfer) si basa su HTTP, il suo funzionamento prevede l'utilizzo dei metodi HTTP specifici per il recupero di informazioni, per la modifica e per altri scopi. SOAP (Simple Object Access Protocol) è un protocollo per lo scambio di messaggi tra componenti software. SOAP può operare su differenti protocolli di rete, ma HTTP è il più comunemente utilizzato, si basa sul metalinguaggio XML. AMQP (Advanced Message Queuing Protocol), invece, è uno standard open che definisce un protocollo a livello applicativo per il message-oriented middleware; sostanzialmente permette il passaggio di messaggi aziendali tra applicazioni o organizzazioni.

1.5.1 Vantaggi

L'architettura a microservizi presenta numerosi vantaggi, i quali riguardano principalmente la distribuzione e l'implementazione continua di applicazioni grandi e complesse, la dimensione del microservizio, l'isolamento tra microservizi e l'eterogeneità dello stack tecnologico:

[5]Il time-to-market (TTM) è il periodo di tempo che intercorre tra il momento in cui un prodotto o servizio viene sviluppato e il momento in cui viene messo in vendita sul mercato.

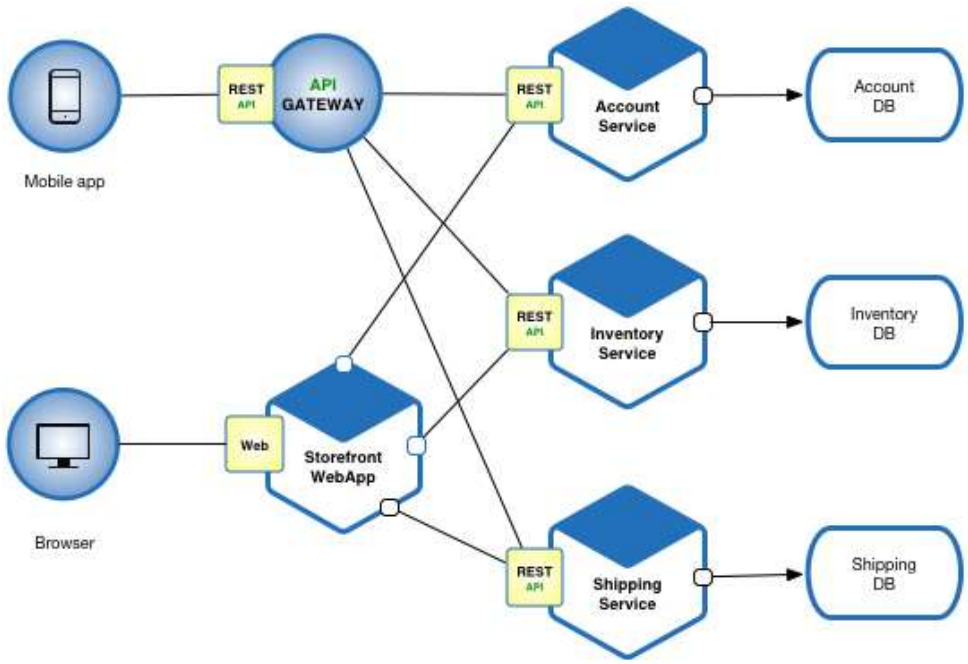

Figura 1.8: Architettura a microservizi. Fonte: [11]

- **Scalabilità**: i singoli servizi possono essere scalati indipendentemente l'uno dall'altro, rendendo più semplice gestire il carico di lavoro e aumentare la capacità del sistema.
- **Agilità**: la dimensione ridotta del team che sviluppa un microservizio può migliorare l'efficienza e la qualità dello sviluppo. Il team, essendo composto da pochi individui, avrà una comunicazione efficace e si vedrà aumentata la flessibilità del progetto.
- **Resilienza**: i singoli servizi possono essere aggiornati o sostituiti senza interferire con il funzionamento dell'intera applicazione, rendendo il sistema più resistente ai guasti. Se si verifica un guasto in un servizio, sarà interessato solo quel servizio; gli altri servizi continueranno a gestire le richieste.
- **Manutenibilità migliorata**: ogni servizio è relativamente piccolo, quindi più facile da comprendere e da modificare dai nuovi sviluppatori.
- **Testabilità del servizio**: i servizi, essendo più piccoli, sono veloci e semplici da testare; ciò può facilitare la verifica della qualità del codice.
- **Riduzione dell'accoppiamento**: il fatto che i microservizi sono meno dipendenti gli uni dagli altri, permette di modificarne o sostituirne uno senza dover modificare gli altri.
- **Migliore implementabilità**: i servizi accoppiati in modo lasco possono quindi essere implementati in modo indipendente.
- **Migliore velocità di rilascio**: l'indipendenza tra i microservizi permette agli sviluppatori di implementare e ridimensionare i propri servizi in maniera indipendente da tutti gli altri team. I microservizi possono essere distribuiti in modo indipendente, rendendo più semplice il deploy di nuove versioni.

- **Stack di tecnologie variabile**: quando si sviluppa un nuovo servizio si può scegliere un nuovo stack tecnologico; allo stesso modo, quando si apportano modifiche importanti a un servizio esistente, è possibile riscrivere tale servizio utilizzando un nuovo stack tecnologico.

1.5.2 Svantaggi

L'architettura a microservizi, sebbene risolva molti dei problemi presentati nell'architettura monolitica (vedi Cap. 1.4.2), introduce alcuni lati negativi che vanno tenuti in considerazione quando si decide di adottare l'architettura a microservizi come modello di sviluppo dell'applicazione. L'architettura a microservizi avrà quindi dei fattori svantaggiosi relativi alla maggiore complessità rispetto ad un sistema monolitico:

- **Complessità**: gestire un gran numero di servizi tra loro indipendenti può essere complesso, e richiede un maggior pianificazione del progetto e coordinamento tra team per la comunicazione tra servizi.
- **Integrazione tra servizi**: la comunicazione tra servizi, tramite API ben definite, può essere complessa e richiede una maggiore attenzione alla qualità di implementazione delle interfacce. I servizi devono fornire un'interfaccia ben definita per poter consentire un'interazione da altri servizi.
- **Latenza**: la comunicazione tra servizi avviene tramite rete, quindi la latenza di alcune applicazioni può crescere rispetto ad un'implementazione monolitica della stessa.
- **Testing end-to-end**: la testabilità del servizio è semplice grazie alle dimensioni ridotte; risulta invece difficoltoso testare le funzionalità end-to-end. Per testare un'interazione tra microservizi è necessario fare il deploy in un ambiente di test dell'architettura interessata ed eseguire i test sull'intera architettura.
- **Debug di codice**: la distribuzione dei servizi rende complesso il debug dei problemi, soprattutto quando si hanno degli errori nella comunicazione tra servizi. Tramite un sistema di logging centralizzato è possibile facilitare la risoluzione di tale problema.
- **Costo**: in alcuni casi il costo dell'architettura a microservizi può essere maggiore in termini di costi di sviluppo e gestione, rispetto a soluzioni monolitiche.
- **Competenze**: gli sviluppatori dei team dovranno possedere competenze riguardanti l'architettura a microservizi, la comunicazione in rete tra servizi e la gestione dei dati per implementare correttamente tale architettura.

Nell'architettura a microservizi, le dipendenze che legano i servizi tra loro, possono portare al "dependency hell". Il termine "dependency hell" è un termine colloquiale utilizzato per descrivere la difficoltà nella gestione delle dipendenze tra i servizi che compongono l'applicazione. Per funzionare correttamente, ciascun servizio, dipende da altri e la gestione delle dipendenze può diventare complessa [22].

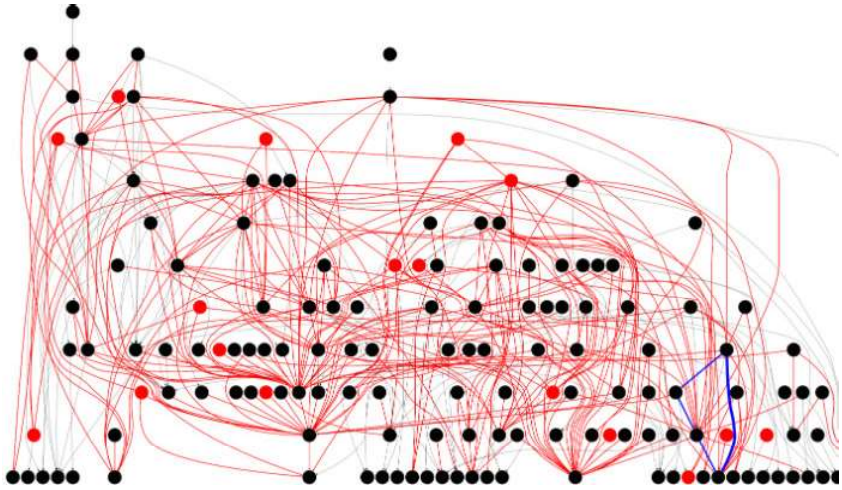

Figura 1.9: Un esempio grafico di dipendenze in un'architettura a microservizi. Fonte: [22]

1.6 Event-Driven Architecture

L'architettura guidata ad eventi (o event-driven architecture) è un pattern comune nelle applicazioni moderne create con microservizi. L'event-driven architecture utilizza gli eventi per comunicare tra servizi accoppiati in modo lasco.

Un evento è un cambiamento di stato o un aggiornamento, come per esempio, il pagamento di un ordine su Amazon o la creazione di un file in uno storage condiviso. Gli eventi possono riportare uno stato (per esempio: la notifica di una lista di articoli acquistati, con il loro prezzo e l'indirizzo di consegna) oppure possono agire da identificatori (per esempio: la notifica che l'ordine è stato spedito).

Le architetture guidate da eventi hanno tre componenti chiave: i produttori di eventi, il router di eventi e i consumatori di eventi. Un produttore pubblica un evento al router, che filtra gli eventi e li invia ai consumatori. I servizi del produttore e i servizi del consumatore sono disaccoppiati, il che consente loro di essere ridimensionati, aggiornati e distribuiti in modo indipendente. [2]

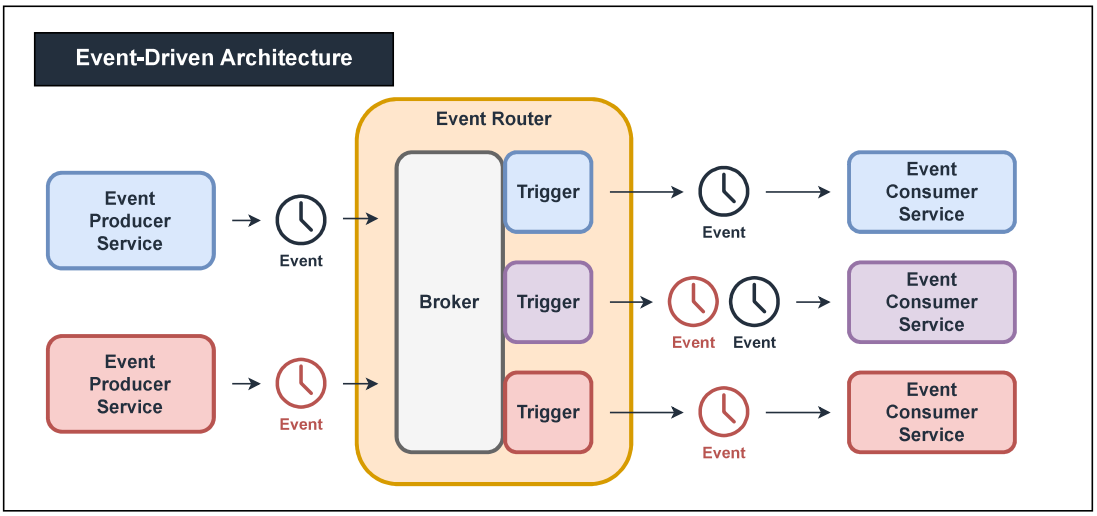

Figura 1.10: Funzionamento ad alto livello di un architettura guidata ad eventi

Tramite il disaccoppiamento tra produttori e consumatori, tramite un router di eventi, si può ottenere un'architettura più resiliente ai guasti. Infatti, tale proprietà consente di pubblicare eventi quando i consumatori non sono disponibili e consente di registrarsi come consumatori di un tipo di evento anche quando non c'è ancora nessun produttore di tale evento.

CAPITOLO 2

Analisi del sistema

In questo capitolo verranno introdotte le competizioni CTF, con particolare riferimento alle CTF Attack and Defense. Verranno inoltre descritti il contesto in cui dovrà essere eseguito il sistema e le caratteristiche che dovrà possedere. Successivamente verrà presentato l'approccio utilizzato per la progettazione del sistema. Verranno poi analizzati i requisiti, con particolare riferimento all'analisi dei requisiti funzionali, non funzionali e dei volumi. Nell'analisi dei requisiti funzionali verranno analizzate le funzionalità che il sistema dovrà fornire. Nell'analisi dei requisiti non funzionali, invece, verranno analizzate le caratteristiche che il sistema dovrà avere. Tramite l'analisi dei volumi, si trarranno delle conclusioni che verranno utilizzate in fase di progettazione del sistema.

2.1 Introduzione e contesto

Prima di studiare i requisiti del sistema, è bene definire lo scopo del sistema e fornire alcune informazioni sul contesto in cui il sistema andrà eseguito. Si introducono, quindi, le competizioni CTF, le CTF "Attack and Defense" di CyberChallenge.IT, lo scopo e le caratteristiche del sistema e il contesto in cui dovrà essere eseguito.

2.1.1 Le competizioni CTF

Le competizioni CTF (Capture The Flag) consistono in una serie di sfide di sicurezza informatica, dove i partecipanti devono trovare e sfruttare vulnerabilità in sistemi e applicazioni per raccogliere *"bandiere"* (flag) che rappresentano una prova di conquista del servizio. Le CTF sono un'occasione per gli esperti di sicurezza e gli appassionati di mettere alla prova le proprie abilità e acquisire nuove conoscenze. Le sfide possono coprire una vasta gamma di argomenti, tra cui la crittografia, i sistemi, le reti, il web, il reverse engineering e l'exploitation. Possono essere organizzati in formato online o in presenza, durare un paio d'ore o diversi giorni, e possono essere aperte a tutti o riservate ai membri di un gruppo specifico.

La varietà e la complessità delle sfide nelle competizioni CTF sono stimolanti per migliorare in maniera divertente le proprie capacità di sicurezza informatica e di lavoro di squadra. Il carico e la responsabilità, infatti, devono essere condivise tra i membri della squadra; collaborando per risolvere sfide complesse e per apprendere nuove abilità che possono essere utili in futuro. Le competizioni CTF stimolano anche lo sviluppo delle capacità di problem solving, perché tutti i concorrenti affrontano diverse sfide e devono proporre soluzioni innovative per risolverle. [26]

Principalmente, ci sono due tipi di competizioni CTF: le CTF Jeopardy e le CTF Attack and defense.

CTF Jeopardy

Le CTF Jeopardy consistono in una competizione caratterizzata dall'assenza di interazione tra i team. Le sfide della CTF sono divise in categorie e livello di difficoltà; maggiore è la difficoltà e maggiore sarà il punteggio attribuito alla risoluzione della sfida. Le squadre, risolvendo le varie sfide, acquisiranno dei punti. La squadra che riesce a ottenere il maggior numero di punti vince la competizione.

CTF Attack and defense

Le CTF Attack and Defense (A&D) sono un tipo di competizione più complessa e interattiva. Nelle competizioni A&D tutti i team hanno una macchina virtuale a loro dedicata, chiamata *vulnbox*, preconfigurata dagli organizzatori della competizione. La macchina virtuale di ogni team è identica, ed espone gli stessi servizi.

I team dovranno quindi scoprire le vulnerabilità dei servizi, correggerle nei propri servizi e attaccare i servizi degli altri team competitor. Tramite la correzione delle vulnerabilità si può prevenire di essere attaccati dalle altre squadre; proteggendosi dagli attacchi si guadagnano punti difesa. Correggendo le vulnerabilità si rischia di rendere il servizio non funzionale o non disponibile. Tenendo i servizi raggiungibili e funzionali si ottengono dei punti SLA (Service Level Agreement points); tali punti indicano la disponibilità e il corretto funzionamento dei servizi.

Sfruttando delle vulnerabilità, si possono attaccare i servizi degli altri team; in tale modo si possono sottrarre le flag che i servizi possiedono, guadagnando punti attacco. Ogni flag sottratta, dai servizi attaccati, deve essere inviata al server di gioco per aggiungere dei punti al punteggio della squadra. Il server di gioco, inoltre, ha il compito di inviare le nuove flag alle vulnbox, di controllare l'integrità dei servizi e di mostrare la scoreboard dei punteggi.

2.1.2 Le competizioni CTF "Attack and Defense" di CyberChallenge.IT

CyberChallenge.IT è un programma di formazione in sicurezza informatica per studenti universitari e delle scuole superiori, gestito dal National Cybersecurity Lab; è un'iniziativa italiana per formare giovani talenti dai 16 ai 24 anni, con l'obiettivo di identificare, attrarre e collocare i futuri professionisti nel campo della sicurezza informatica. [16] CyberChallenge.IT organizza un periodo di formazione della durata di circa tre mesi su vari argomenti di sicurezza informatica. Alla fine del percorso formativo vengono organizzate 2 competizioni CTF: una competizione CTF Jeopardy locale ad ogni università e una competizione CTF nazionale di tipo "Attack and Defense". La competizione locale serve a formare la squadra universitaria che competerà alla competizione nazionale. Nella competizione nazionale si sfideranno tutte le squadre delle varie università che aderiscono al progetto. La competizione "Attack and Defense" è strutturata nella forma standard, vista nel capitolo 2.1.1.

In particolare, la competizione è divisa in round. In ogni round i flag dei servizi vengono aggiornati dal server di gioco; quindi è necessario attaccare i team competitor ogni round per poter ottenere il massimo punteggio. Ogni round, chiamato anche tick, dura 120 secondi. Durante il tick, il server di gioco controlla l'integrità dei servizi interagendo con loro e controllando i flag tramite un accesso legittimo.

Le flag della competizione rispettano la regex[1] `[A-Z0-9]{31}=`, cioè sono composte da 31 caratteri alfanumerici maiuscoli e numeri, conclusi da un segno di uguale.

[1] Una regex (regular expression) è una sequenza di caratteri che specifica un modello di ricerca nel testo.

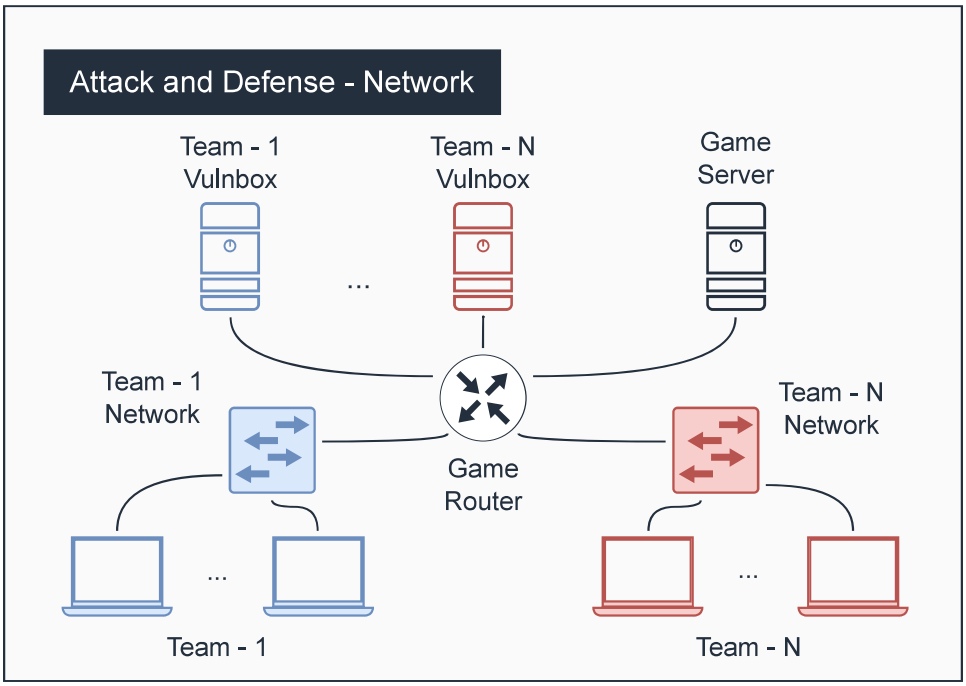

Figura 2.1: Schema di rete della competizione Attack and Defense nella sfida nazionale di CyberChallenge.IT.

È possibile inviare le flag al server di gioco tramite una form presente nel portale CTF messo a disposizione dallo stesso, oppure automatizzare il processo interagendo con un'API tramite una richiesta POST. I flag non inviati entro 5 tick sono considerati scaduti e non contribuiranno al punteggio. La squadra che riesce a ottenere il maggior numero di punti vince la competizione.

2.1.3 Lo scopo e le caratteristiche del sistema

Lo scopo della tesi è proprio quello di progettare un sistema capace di automatizzare il processo di lancio degli exploit e l'invio di flag. Tramite l'automazione dell'esecuzione di tali task è possibile concentrarsi sulla competizione, non dovendo gestire manualmente il lancio degli exploit e l'invio di flag.

Per massimizzare il punteggio, il sistema dovrà essere disponibile per l'intera durata della competizione. È quindi importante che il sistema sia altamente disponibile e resiliente ai guasti di un singolo nodo. Il sistema, in caso di guasti, deve essere capace di tornare in uno stato tale per cui possa riprendere il corso dell'esecuzione in un periodo di tempo breve.

2.1.4 Contesto di esecuzione del sistema

Il sistema distribuito che si intende progettare deve esclusivamente girare nei computer dei componenti del team, in quanto il regolamento vieta l'utilizzo di qualsiasi risorsa computazionale esterna per tale scopo. Non è possibile quindi eseguire il sistema né nell'architettura di un cloud-provider, né in un server esterno.

Le uniche risorse utilizzabili sono quindi i computer dei componenti del team. È bene tenere in considerazione che i computer dei componenti possono rompersi e/o perdere connettività tra di loro; quindi il sistema dovrà essere resiliente ai guasti di un nodo. C'è

bisogno quindi di un orchestratore capace di organizzare i vari servizi del sistema distribuito in maniera uniforme e altamente disponibile tra i computer dei componenti della squadra. Si tratterà della scelta delle tecnologie nel Cap. 3.3.

2.2 Approccio alla progettazione del sistema distribuito

In questa tesi si cercherà di progettare il sistema distribuito, seguendo l'approccio sistematico descritto nel libro "System Design Interview – An Insider's Guide" di Alex Xu [1]. Alex Xu, nel libro, definisce il processo da seguire quando si progetta un sistema distribuito. Nella progettazione di un sistema, non c'è una soluzione che sia valida per qualsiasi caso, ma ci sono dei passi e degli argomenti comuni da coprire in ogni progettazione. Analizziamo i quattro step generali dell'approccio:

1. **Comprendere il problema e stabilire lo scopo del design**: innanzitutto è necessario acquisire informazioni riguardo il sistema che si desidera progettare. Andranno quindi analizzati i vari aspetti del sistema, comprendendo i requisiti funzionali (vedi Cap. 2.4), non funzionali (vedi Cap. 2.5) e il volume di traffico che il sistema deve sopportare (vedi Cap. 2.6).

2. **Progettare il sistema ad alto livello**: dai requisiti ottenuti dalla fase precedente, va progettato il sistema ad alto livello; andranno pertanto spiegati i vari componenti del sistema, le loro funzionalità e come interagiscono tra loro (vedi Cap. 3).

3. **Progettare il sistema a basso livello**: in questa fase si dovrà sviluppare ogni componente dell'architettura ad alto livello, definendo le API del sistema, la modellazione dei database e i vari compromessi del design suggerito (vedi Cap. 4).

4. **Riassunto**: andranno riassunte infine le varie caratteristiche del sistema.

2.3 Analisi dei requisiti

L'analisi dei requisiti è un passo fondamentale per comprendere i requisiti del sistema, come per esempio quali sono le funzionalità che dovrà implementare e quali caratteristiche dovrà avere. Il sistema verrà utilizzato dai componenti del team, i quali avranno accesso all'intero sistema. E' bene precisare che gli utenti potranno interagire con le varie API del sistema dopo essersi autenticati.

Gli utenti autenticati al sistema devono poter creare delle vulnbox da attaccare per poi inserire i metadati dei servizi presenti in ogni macchina virtuale dei team competitor. Gli utenti leggono la lista delle vulnbox da attaccare e la lista dei metadati dei servizi. Gli utenti possono creare exploit, che attaccano un servizio specifico oppure generici e leggere la lista dei metadati degli exploit. Gli exploit caricati potrebbero contenere bug o errori logici che non garantiranno una corretta esecuzione; è quindi necessario poter rendere inattivo lo scheduling di un exploit, oppure deve essere possibile eliminare tale exploit.

Il sistema dovrà essere in grado di schedulare in automatico (ogni `tick_duration` secondi) il lancio di ogni exploit contro un servizio specifico, contro ogni vulnbox da attaccare. Il tempo di ogni round è definito da `tick_duration`, solitamente dura 120 secondi.

Le impostazioni della competizione, come la durata del tick, l'orario di inizio e di fine della competizione, saranno impostate dall'utilizzatore all'avvio del sistema.

Gli utilizzatori possono avere il bisogno di eseguire degli exploit una volta, oppure schedulare il lancio di essi ogni round. Gli utenti possono leggere le informazioni sullo stato di un particolare job oppure un sommario dei job, filtrati per round. I job possono essere filtrati per `round_number` e per tipo (se schedulato dal sistema o schedulato manualmente).

E' possibile caratterizzare i task che andranno eseguiti dal sistema I task sono idempotenti[2], cioè un'esecuzione multipla di un task fornisce lo stesso risultato. I task non hanno necessità di essere eseguiti in ordine; un job schedulato può concludersi dopo di uno pianificato successivamente. I task saranno pianificati per essere eseguiti nel presente o nel futuro, ma non nel passato; Se un nodo worker diventa non raggiungibile, si devono rischedulare i job che sono in esecuzione in tale istanza. E' necessario rilanciare i nodi worker che vanno offline[3] in un tempo relativamente ridotto; si può ottenere ciò riducendo al minimo il tempo di cold start di tali nodi e introducendo un controllo periodico frequente sullo stato di salute dei nodi.

I task saranno eseguiti in maniera asincrona da dei nodi worker. Dopo ogni esecuzione, dovranno aggiornare lo stato dell'esecuzione.

I flag sottratti vengono estrapolati dall'output delle esecuzioni degli exploit per essere inviati al server di gioco. Il sistema dovrà automaticamente inviare i flag al server di gioco entro 5 round.

Dopo aver definito le specifiche del sistema nella descrizione delle competizioni Attack and Defense (Cap 2.1.1), nella sezione che descrive il contesto (Cap 2.1.4) e nel capitolo corrente, si può passare alla fase di analisi dei requisiti funzionali.

2.4 Requisiti funzionali

I requisiti funzionali descrivono in modo formale, le funzionalità e i servizi che il sistema metterà a disposizione dei componenti del team e quelli che eseguirà in maniera autonoma. Il sistema avrà delle funzionalità che esporrà a tutti gli utenti e altre che saranno esposte ai soli utenti autenticati; inoltre il sistema avrà delle necessità interne per l'automazione di alcuni processi, è bene quindi specificare anche questi ultimi.

2.4.1 Requisiti funzionali utente non autenticato

(RF-1) Accesso al frontend dell'applicazione.

- **(RF-1.1)** L'utente deve poter accedere al frontend dell'applicazione per poter interagire tramite API con il sistema.

(RF-2) Autenticazione al sistema tramite login.

- **(RF-2.1)** L'utente che intende autenticarsi deve fornire un username ed una password per potersi autenticare al sistema.

[2] In informatica, in matematica, e in particolare in algebra, l'idempotenza è una proprietà delle funzioni per la quale applicando molteplici volte una funzione data, il risultato ottenuto è uguale a quello derivante dall'applicazione della funzione un'unica volta.

[3] Offline: non disponibile o non raggiungibile.

2.4.2 Requisiti funzionali utente autenticato

(RF-1) Gestione delle vulnbox dei team competitor da attaccare.

- **(RF-1.1)** L'utente deve poter creare una vulnbox di un team.
- **(RF-1.2)** L'utente può leggere la lista delle vulnbox.
- **(RF-1.3)** L'utente può leggere i dati di una vulnbox tramite l'identificativo.
- **(RF-1.4)** L'utente deve poter modificare una vulnbox da attaccare tramite l'identificativo.
- **(RF-1.5)** L'utente deve poter cancellare una vulnbox tramite l'identificativo.

(RF-2) Gestione dei metadati dei servizi da attaccare per ogni vulnbox.

- **(RF-2.1)** L'utente deve poter creare i metadati dei servizi da attaccare.
- **(RF-2.2)** L'utente può leggere la lista dei metadati dei servizi.
- **(RF-2.3)** L'utente può leggere i metadati di un servizio tramite l'identificativo.
- **(RF-2.4)** L'utente deve poter modificare i metadati di un servizio tramite l'identificativo.
- **(RF-2.5)** L'utente deve poter cancellare i metadati di un servizio tramite l'identificativo.

(RF-3) Gestione degli exploit.

- **(RF-3.1)** L'utente deve poter creare i metadati e caricare il file di exploit.
- **(RF-3.2)** L'utente deve poter scaricare il file di exploit precedentemente caricato.
- **(RF-3.3)** L'utente può leggere la lista dei metadati degli exploit caricati filtrati per servizio e ordinati per versione.
- **(RF-3.4)** L'utente può leggere i metadati di un exploit tramite l'identificativo.
- **(RF-3.5)** L'utente deve poter modificare i metadati di un exploit.
- **(RF-3.6)** L'utente deve poter impostare un exploit come attivo o inattivo.
- **(RF-3.7)** L'utente deve poter cancellare i metadati di un exploit, ma non il file caricato.

(RF-4) Creazione di job personalizzati.

- **(RF-4.1)** L'utente deve poter creare dei job personalizzati che vengono eseguiti una sola volta.
  - **(RF-4.1.1)** L'utente deve poter creare dei job personalizzati che attaccano un team specifico, una sola volta.
  - **(RF-4.1.2)** L'utente deve poter creare dei job personalizzati che attaccano tutti i team, una sola volta.
  - **(RF-4.1.3)** L'utente deve poter creare dei job personalizzati generici che vengono eseguiti una sola volta.
- **(RF-4.2)** L'utente deve poter creare dei job personalizzati che vengono eseguiti ogni tick.
  - **(RF-4.2.1)** L'utente deve poter creare dei job personalizzati che attaccano un team specifico, una volta per tick.

(RF-4.2.2) L'utente deve poter creare dei job personalizzati che attaccano tutti i team, una volta per tick.

(RF-4.2.3) L'utente deve poter creare dei job personalizzati generici che vengono eseguiti una volta per tick.

(RF-5) Visualizzazione dello stato dei job.

(RF-5.1) L'utente deve poter controllare lo stato dei job filtrati per round.

(RF-5.2) L'utente deve poter controllare lo stato dei job filtrati per round e per tipo.

2.4.3 Requisiti funzionali sistema automatizzato

(RF-1) Calcolare la configurazione desiderata dei job (quali vulnbox da attaccare, con quali exploit, riferiti a quali servizi).

(RF-2) Modificare la configurazione dei job.

(RF-3) Creare, in modo automatico, i job in base alla configurazione attuale.

(RF-4) Leggere la lista dei job da schedulare.

(RF-5) Schedulare, in modo automatico, i job in base alla configurazione attuale.

(RF-6) Eseguire un job almeno una volta. L'esecuzione multipla di un task è tollerata; l'importante è che ogni task sia eseguito.

(RF-7) Cancellare un job occasionale dalla configurazione.

(RF-8) Aggiornare lo stato di un job.

(RF-9) Inviare i flag sottratti, tramite l'esecuzione di exploit, al server di gioco.

(RF-10) Leggere il file di exploit.

(RF-11) Il sistema deve generare le credenziali degli utenti per accedere al sistema.

2.5 Requisiti non funzionali

I requisiti non funzionali sono specifiche riguardanti le prestazioni, la sicurezza, la scalabilità, l'usabilità, la compatibilità, la manutenibilità, la disponibilità e altri aspetti non relativi alle funzionalità specifiche di un sistema software o prodotto. Tramite lo studio di tali non requisiti è possibile definire le caratteristiche principali che il sistema dovrà possedere.

2.5.1 Performance

Definisce la velocità con cui un il sistema software deve rispondere alle azioni di determinati utenti e di processi in background del sistema con un determinato carico di lavoro.

(RN-1) Le API per gestire le risorse, quali vulnbox, exploits e servizi, devono fornire un tempo di risposta di 2 secondi o meno in un browser desktop Chrome.

(RN-2) L'API per creare job personalizzati deve essere in grado di fornire un tempo di risposta di 2 secondi o meno per ogni richiesta.

(RN-3) Il sistema deve calcolare la configurazione desiderata dei job in meno di 300ms. Il sistema deve modificare la configurazione dei job entro 2 secondi.

(RN-4) Il sistema deve essere in grado di creare tutti i job da schedulare in meno di 10 secondi ogni round. La creazione dei job deve avvenire 15 secondi prima dell'inizio del round successivo.

(RN-5) Il sistema deve essere in grado di schedulare tutti i job creati in meno di 35 secondi ogni round. Il sistema deve schedulare almeno i primi 100 job in 5 secondi.

(RN-6) Il sistema deve cancellare un job occasionale dalla configurazione in meno di 300ms.

(RN-7) Il sistema deve aggiornare lo stato di un job in meno di 300ms.

(RN-8) Il sistema deve inviare i flag, sottratti dalle vulnbox, al server di gioco entro 4 tick.

(RN-9) Il sistema deve leggere il file di exploit in meno di 2 secondi.

(RN-10) Il sistema deve eseguire un job in meno di 2 secondi.

(RN-11) Il sistema deve leggere i job da mettere in coda in meno di 3 secondi.

(RN-12) Il sistema deve accodare 100 job per essere eseguiti in meno di 3 secondi.

(RN-13) Il sistema deve eseguire la totalità dei job del round corrente entro la scadenza del tick.

(RN-14) Il sistema deve inviare i flag al server di gioco entro 4 tick.

(RN-15) I flag sottratti non devono essere persi, devono essere inviati al server di gioco almeno una volta.

2.5.2 Scalabilità

La scalabilità cerca di stimare quanto è il carico di lavoro in base al quale il sistema soddisferà ancora i requisiti prestazionali. Esistono due modi per abilitare la scalabilità del sistema man mano che i carichi di lavoro aumentano: scalabilità orizzontale e verticale.

(RN-1) Il sistema deve scalare abbastanza da supportare la creazione, la pianificazione e l'esecuzione di tutti i job desiderati per ogni tick.

(RN-2) Il sistema deve scalare abbastanza da supportare l'invio di tutti i flag per ogni round.

2.5.3 Affidabilità

L'affidabilità specifica la probabilità che il sistema o il suo elemento funzionino senza guasti per un determinato periodo di tempo in condizioni predefinite; tradizionalmente, questa probabilità è espressa in percentuale.

(RN-1) Il sistema deve funzionare senza guasti nel 95% dei casi d'uso durante la competizione.

2.5.4 Manutenibilità

La manutenibilità definisce il tempo necessario affinché una soluzione o un suo componente venga riparato, modificato per aumentare le prestazioni o altre qualità o adattato a un ambiente in evoluzione. Come l'affidabilità, può essere espressa come probabilità di riparazione nel tempo.

(RN-1) Il tempo medio di ripristino del sistema (MTTRS[4]) a seguito di un guasto del sistema non deve essere superiore a 10 minuti.

2.5.5 Disponibilità

La disponibilità descrive la probabilità che il sistema sia accessibile a un utente in un determinato momento. La disponibilità può essere espressa come percentuale prevista di richieste riuscite o come percentuale di tempo in cui il sistema è accessibile per il funzionamento.

(RN-1) Il sistema deve essere disponibile per il 95% della durata della competizione.

2.5.6 Sicurezza

La sicurezza è un requisito non funzionale che garantisce che tutti i dati all'interno del sistema o parte di esso siano protetti nella fase di storicizzazione e di trasporto.

(RN-1) Il sistema, essendo eseguito in un ambiente controllato, non ha necessità di fornire una protezione a livello di trasporto dei dati.

(RN-2) Il sistema deve proteggere l'interazione con le API tramite un'autenticazione.

(RN-3) Le credenziali degli utenti dovranno essere criptate nel database degli utenti.

2.5.7 Portabilità

La portabilità determina come un sistema o un suo elemento può essere lanciato all'interno di un ambiente o di un altro.

(RN-1) I vincoli imposti dalla struttura e dal regolamento della competizione, non impongono dei limiti stretti sulla portabilità. Il sistema dovrà essere eseguito su un cluster di computer. Non sarà quindi necessario rendere eseguibile il sistema su altre piattaforme.

[4]MTTRS: Mean Time To Restore the System, cioè il tempo medio di ripristino del sistema in caso di guasti

2.5.8 Compatibilità

La compatibilità, come aspetto aggiuntivo della portabilità, definisce come un sistema, o una parte di esso, può coesistere con un altro sistema nello stesso ambiente.

(RN-1) Il sistema subirà aggiornamenti in un periodo precedente o successivo alla competizione. Gli studi sulla compatibilità tra versioni andranno effettuati in fase di aggiornamento dei microservizi. E' bene coordinare l'aggiornamento tra microservizi strettamente accoppiati per evitare problemi logici.

2.6 Analisi dei volumi

L'analisi dei volumi di traffico è un processo che consiste nell'esaminare la quantità di dati o informazioni che transitano attraverso un sistema, al fine di comprendere le esigenze di capacità e le prestazioni richieste. Questa analisi aiuta a progettare e dimensionare correttamente il sistema, a identificare eventuali colli di bottiglia e a prevedere future esigenze di capacità.

2.6.1 Requisiti di traffico

Il traffico che deve sopportare il sistema varia a seconda del numero di richieste che dovrà soddisfare. Il traffico destinato alle API esposte per l'utente è ridotto, in quanto il numero di utenti è limitato alle dimensioni del team.

Gli schedulatori e gli esecutori di task, invece, possono dover servire un grande numero di richieste, in quanto il numero di task è proporzionale al prodotto tra il numero di vulnbox da attaccare, il numero di exploit per ogni servizio e il numero di servizi esposti. Per esempio, se si aggiunge un team da attaccare, il numero di task aumenterà notevolmente. Conseguentemente un numero maggiore di exploit lanciati avrà come effetto un incremento del numero di flag da inviare al server di gioco. Si prevede che la competizione duri 4 ore e che la durata di un tick sia di 2 minuti, per un totale di 120 round.

Si assume che il traffico sia maggiore rispetto alle esigenze per una migliore generalizzazione dei requisiti di volume:

(RV-1) Gli utenti del sistema saranno i componenti del proprio team, quindi ci saranno 5 utilizzatori.

(RV-2) Si prevede di attaccare 50 vulnbox.

(RV-3) Si prevede di attaccare 5 servizi per vulnbox.

(RV-4) Si prevede di creare una media di 1.5 exploit per ogni servizio di ogni vulnbox.

(RV-5) Si prevede di creare un job ricorrente per un servizio di ogni vulnbox.

(RV-6) Si prevede di creare una media di 6 job ricorrenti per un servizio di una singola vulnbox.

(RV-7) Si prevede di creare una media di 3 job ricorrenti generici.

(RV-8) Si prevede di creare 10 job non ricorrenti per un servizio di ogni vulnbox.

(RV-9) Si prevede di creare 15 job non ricorrenti per un servizio di una singola vulnbox.

(RV-10) Si prevede di creare 15 job non ricorrenti generici.

(RV-11) Si stima che per ogni esecuzione dei task si catturino in media 4 flag.

(RV-12) Si stima che ogni utente accederà al frontend del sistema circa 50 volte.

Dai requisiti appena elencati, è possibile studiare i volumi effettivi di traffico che il sistema deve essere in grado di supportare. Si studiano i requisiti dei volumi per ogni requisito funzionale; tramite qualche calcolo, è possibile ricavare i volumi per ogni tipologia di requisito funzionale, tra cui quelli degli utenti non autenticati, degli utenti autenticati e del sistema automatizzato. Le stime dei volumi sono espresse in numero di richieste per round (*req/round*) e in numero di richieste totali (*req*)

Analisi dei volumi per i requisiti funzionali degli utenti non autenticati

Requisito funzionale	Tipo operazione	N° richieste per round	N° richieste totali
(RF-1.1)	Lettura frontend	10 req/round	250 req
(RF-2.1)	Richiesta di autenticazione	10 req/round	250 req

Tabella 2.1: Analisi dei volumi dei requisiti funzionali degli utenti non autenticati.

Analisi dei volumi per i requisiti funzionali degli utenti autenticati

Requisito funzionale	Tipo operazione	N° richieste per round	N° richieste totali
(RF-1.1)	Creazione vulnbox	-	55 req
(RF-1.2)	Lista vulnbox	5 req/round	600 req
(RF-1.3)	Ricerca vulnbox per id	3 req/round	360 req
(RF-1.4)	Modifica vulnbox	-	20 req
(RF-1.5)	Cancellazione vulnbox	-	5 req
(RF-2.1)	Creazione metadati servizio	-	9 req
(RF-2.2)	Lista metadati servizi	5 req/round	600 req
(RF-2.3)	Ricerca metadati servizi per id	2 req/round	240 req
(RF-2.4)	Modifica metadati servizio	-	20 req
(RF-2.5)	Cancellazione metadati servizio	-	4 req
(RF-3.1)	Creazione exploit	-	50 req
(RF-3.2)	Download file di exploit	10 req/round	180 req
(RF-3.3)	Lista metadati exploit	5 req/round	600 req
(RF-3.4)	Ricerca metadati exploit per id	2 req/round	240 req
(RF-3.5) **(RF-3.6)**	Modifica metadati exploit	-	20 req
(RF-3.7)	Cancellazione metadati exploit	-	5 req
(RF-4.1.1)	Creazione job non ricorrente contro ogni vulnbox	-	10 req
(RF-4.1.2)	Creazione job non ricorrente contro una vulnbox	-	15 req
(RF-4.1.3)	Creazione job non ricorrente generico	-	15 req
(RF-4.2.1)	Creazione job ricorrente contro ogni vulnbox	-	2 req
(RF-4.2.2)	Creazione job ricorrente contro una vulnbox	-	10 req
(RF-4.2.3)	Creazione job ricorrente generico	-	5 req
(RF-5.1) **(RF-5.2)**	Lettura dello stato dei job di un round	30 req/round	3.600 req

Tabella 2.2: Analisi dei volumi dei requisiti funzionali degli utenti autenticati.

Analisi dei volumi per i requisiti funzionali del sistema automatizzato

Requisito funzionale	Tipo operazione	N° richieste per round	N° richieste totali
(RF-1)	Calcolare la configurazione dei job	-	1.260 req
(RF-2)	Modificare la configurazione dei job	-	1.260 req
(RF-3)	Creare un job	490 req/round	59.330 req
(RF-4)	Leggere lista job da schedulare	18 req/round	2.160 req
(RF-5)	Schedulare un job	490 req/round	59.330 req
(RF-6)	Eseguire un job	490 req/round	59.330 req
(RF-7)	Cancellare un job occasionale dalla configurazione	-	530 req
(RF-8)	Aggiornare lo stato di un job	980 req/round	117.600 req
(RF-9)	Inviare un flag	1.960 req/round	235.200 req
(RF-10)	Leggere file di exploit	10 req/round	800 req
(RF-11)	Generare credenziali utente	-	1 req

Tabella 2.3: Analisi dei volumi dei requisiti funzionali del sistema.

2.6.2 Conclusioni

Sulla base delle stime per eccesso di cui sopra, possiamo trarre le seguenti conclusioni su vari aspetti studiati nei requisiti di traffico.

Il traffico generato dall'invio di flag e dalla creazione, pianificazione e esecuzione di job sono la principale fonte di carico del sistema. Si dovranno quindi adottare soluzioni tecniche in termini di storicizzazione, lettura e modifica dei dati adatte al carico a cui il sistema è sottoposto.

Data la necessità di avere un sistema altamente disponibile, è bene utilizzare delle tecnologie computazionali e di storicizzazione capaci di risolvere in automatico la non raggiungibilità, la terminazione o il crash di uno o più nodi.

Considerazioni sui volumi per l'utente non autenticato

Dalla tabella dei volumi degli utenti non autenticati (Tab. 2.1), si nota che il volume delle operazioni eseguite dagli utenti di tale categoria sono molto limitate.

Considerazioni sui volumi per l'utente autenticato

Dalla tabella 2.2, si nota che le operazioni ricorrenti sono quelle che mirano a leggere le risorse, come per esempio nei requisiti **(RF-1.2)**, **(RF-2.2)** e **(RF-3.3)**. Le letture dei dati sono più frequenti delle scritture di dati.

Considerazioni sui volumi per il sistema automatizzato

Dalla tabella 2.3, si nota che le operazioni ricorrenti sono quelle che interessano l'invio di flag e la creazione, pianificazione, esecuzione e modifica dei task.

Il requisito **(RF-9)** è soddisfatto se si riescono ad inviare circa 2.000 flag a round. E' possibile scalare il servizio di invio dei flag proporzionalmente al numero di flag da inviare al server di gioco.

Per quanto riguarda la creazione **(RF-3)**, la schedulazione **(RF-5)**, l'esecuzione **(RF-6)** e la modifica **(RF-8)** di task, si hanno dei volumi considerevolmente alti. In questo caso, da come è possibile notare dal requisito riguardante la creazione **(RF-3)** e la modifica **(RF-8)** di task, i volumi delle operazioni di lettura e di scrittura dei task sono notevolmente alti; sarà necessario adottare delle soluzioni tecniche particolari per gestire tali dati. Si ricorda che il sistema deve essere il più possibile resiliente ai guasti.

Conclusioni

Sulla base delle stime di cui sopra, possiamo trarre le seguenti conclusioni:

- Il volume delle operazioni degli utenti non autenticati è molto piccolo, non è quindi necessario scalare tali servizi. Un servizio con poca potenza computazionale può soddisfare tutte le richieste senza riscontrare problemi di performance.
- La maggior parte delle operazioni di modifica effettuate dagli utenti autenticati sono eseguite con una bassa frequenza; è possibile ridurre o fermare i processi che espongono tali servizi quando non vengono utilizzati.
- I volumi di scrittura e lettura per il database dei task sono notevolmente alti. Il database dei task dovrebbe essere scalabile orizzontalmente in più nodi per ottimizzare le operazioni di scrittura e lettura dei task.
- Il database dei task dovrebbe essere auto-rigenerante per garantire allo schedulatore dei job una maggiore resilienza ai guasti.
- E' necessario accodare i task prima di eseguirli.
- E' necessario accodare i flag prima di inviarli al server di gioco.
- E' necessario creare i task in parallelo; più repliche di tale servizio si divideranno i task da inserire nel database.
- E' necessario pianificare i task in parallelo; più repliche di tale servizio si divideranno i task da pianificare.
- I task andranno eseguiti in parallelo; le repliche del servizio di esecuzione si divideranno i task da eseguire.

- I processi che pianificano i task devono risiedere su nodi computazionali differenti per garantire una migliore resilienza ai guasti.
- Il numero dei processi che eseguono i task devono essere scalati orizzontalmente in modo proporzionale al numero di task in coda.
- E' necessario inviare i flag in parallelo da processi scalabili orizzontalmente, al variare del numero di flag in coda.

Dopo aver analizzato il sistema nel Cap. 2.3, è possibile passare allo step successivo della metodologia adottata (vedi cap. 2.2): la progettazione ad alto livello.

CAPITOLO 3

Progettazione ad alto livello

In questo capitolo si cercherà di descrivere l'architettura ad alto livello del sistema. Tramite lo studio della vista concettuale saranno poi analizzate le tre tipologie di servizi del sistema: i servizi pubblici, privati e interni. Per ogni tipologia di servizio saranno studiati alto livello i componenti principali del sistema. Verranno introdotte le funzionalità e le caratteristiche dei componenti principali del sistema. Infine verranno introdotte brevemente le varie soluzioni in termini di orchestrazione, computazione, storicizzazione e messaggistica che verranno utilizzate nel sistema.

Ora che sono stati definiti i requisiti del sistema, tramite lo studio dei requisiti funzionali (nel Cap. 2.4), quelli non funzionali (nel Cap. 2.5) e l'analisi dei volumi (nel Cap. 2.6), sono ben definite le caratteristiche e le funzionalità che il sistema deve avere.

Tale studio di analisi, effettuato a prima della progettazione, consente di anticipare le necessità future di funzionalità e di volume, consentendo all'architetto di soluzioni di progettare una soluzione scalabile, performante, affidabile, flessibile, facilmente manutenibile, altamente disponibile, sicura e portabile tra sistemi differenti.

3.1 Vista concettuale del sistema

La vista concettuale è una rappresentazione astratta di un sistema, che descrive i concetti fondamentali e le relazioni tra di essi. La vista concettuale fornisce un punto di vista di alto livello del sistema in questione, evitando i dettagli tecnici e di implementazione. Questa vista può essere utilizzata per la progettazione di altre viste più dettagliate, come il design ad alto livello. La vista concettuale permette di comprendere, ad alto livello, le relazioni e le interazioni tra il sistema e l'ambiente esterno.

E' possibile rappresentare il sistema da un punto di vista concettuale con l'immagine 3.1. Il sistema, come specificato nell'analisi dei requisiti (vedi Cap. 2.3), dovrà interagire con degli utenti tramite un browser. Si può notare, che il sistema fornisce tre diverse tipologie di servizi: i servizi interni, i servizi pubblici e i servizi privati.

- I servizi interni sono l'insieme dei microservizi che non hanno interazione con l'utente, ma che eseguono delle computazioni interne non invocate in modo diretto dagli utenti.
- I servizi pubblici sono quei servizi che sono esposti dal sistema a qualsiasi utente che non si è autenticato nel sistema.
- I servizi privati, invece, sono risorse protette da autenticazione dall'API Gateway; solo gli utenti autenticati al sistema possono accedervi.

Le richieste degli utenti saranno fronteggiate da un componente chiave del sistema, l'API Gateway, il quale verrà trattato nel capitolo 3.2.1. L'API Gateway sarà l'unico endpoint di accesso esterno al sistema; in particolare consentirà l'accesso a tutti gli utilizzatori per i servizi pubblici, limiterà l'accesso ai servizi privati ai soli utenti autenticati e bloccherà le richieste esterne verso i servizi interni del sistema. Gli utenti interagiranno con i servizi di backend tramite delle richieste HTTP di tipo REST con le API esposte dall'API Gateway. Internamente, i servizi privati e i servizi interni, interagiranno tra loro tramite lo scambio di messaggi in maniera asincrona, utilizzando protocolli come l'AMQP, oppure tramite invocazioni sincrone, con API HTTP di tipo REST.

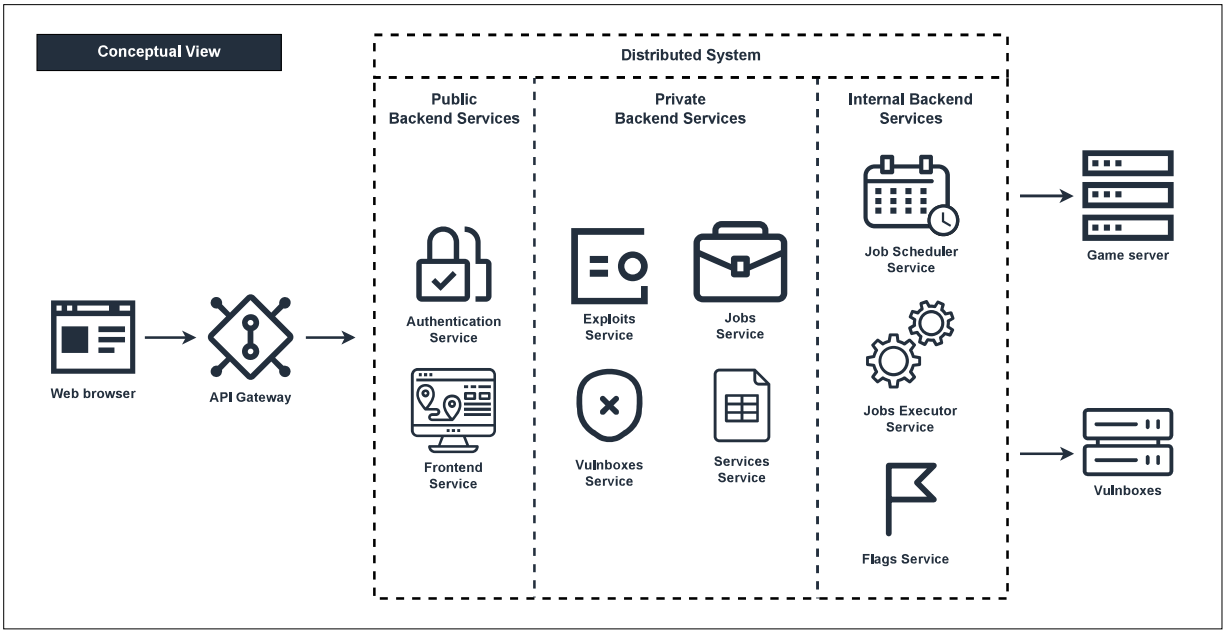

Figura 3.1: Vista concettuale del sistema distribuito.

Gli utenti, tramite il frontend, potranno usufruire delle funzionalità del sistema tramite richieste API. Per autenticarsi al sistema gli utilizzatori interagiranno con il servizio di autenticazione. L'API Gateway sarà poi in grado di riconoscere l'utente autenticato e consentire le sue richieste.

3.2 Architettura ad alto livello

La vista concettuale esprime in modo astratto il sistema. Approfondendo tale vista e analizzando i requisiti funzionali è possibile progettare il sistema ad alto livello.

La progettazione ad alto livello di un sistema distribuito è il processo di definizione e pianificazione delle architetture, delle componenti, delle interazioni e delle politiche di gestione di un sistema distribuito. L'obiettivo è quello di garantire la scalabilità, la disponibilità, l'affidabilità, la sicurezza e la performance del sistema distribuito, tenendo conto delle esigenze specifiche del dominio applicativo, studiate nell'analisi dei requisiti (vedi Cap. 2.3). La progettazione ad alto livello include la definizione della topologia di rete, dei componenti software e dei protocolli di comunicazione. La progettazione del sistema ad alto livello consente di creare l'architettura ad alto livello.

Nell'architettura ad alto livello del sistema, presentata in figura 3.2, si può notare che sono stati divisi per colore le diverse categorie di servizi: i servizi pubblici in verde, quelli privati in giallo e quelli interni in rosso.

Si vuole studiare l'architettura ad alto livello del sistema analizzando le tre categorie di servizi.

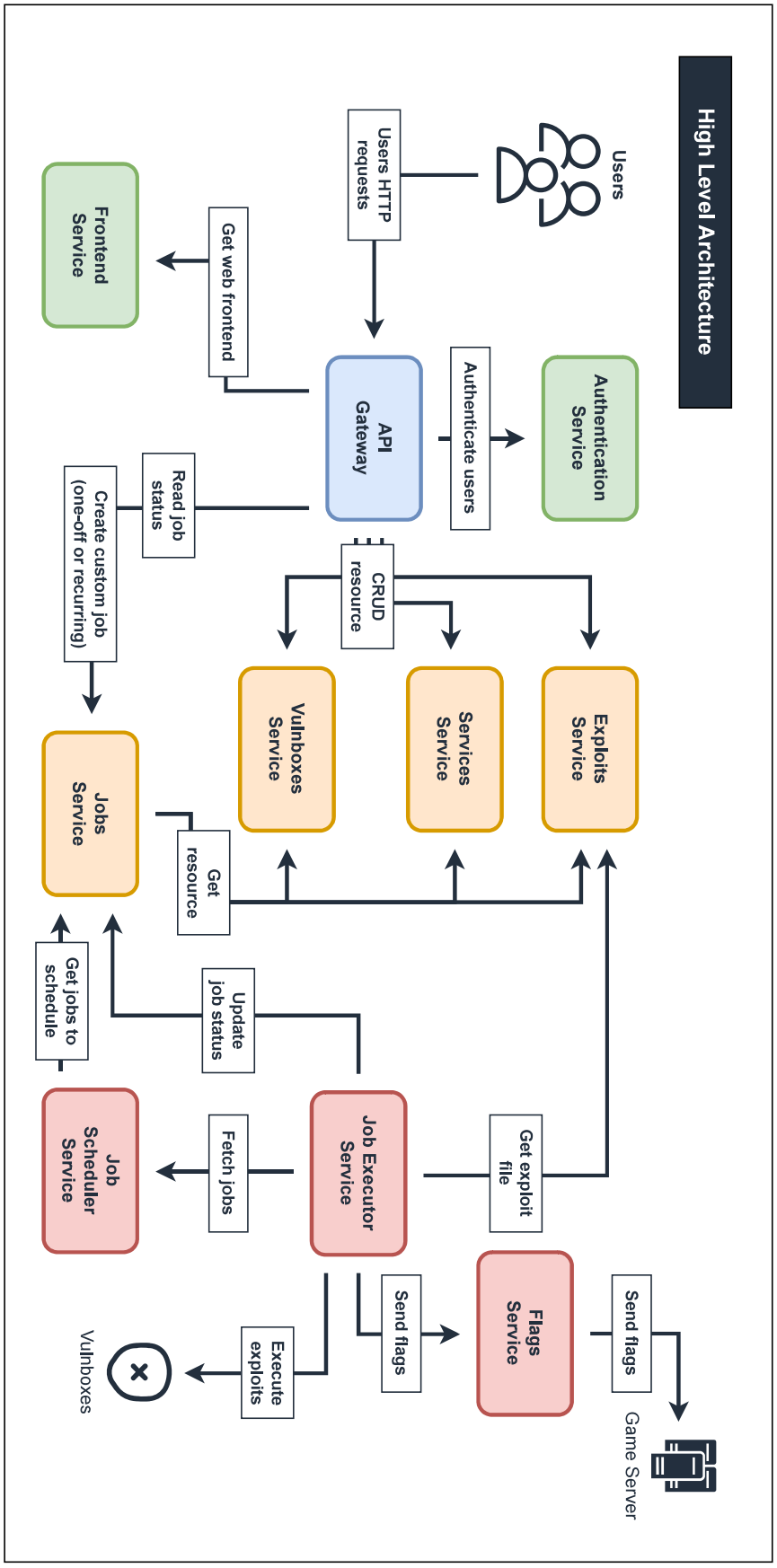

Figura 3.2: Architettura ad alto livello del sistema distribuito.

Servizi pubblici del sistema

I servizi pubblici del sistema sono quei servizi che sono raggiungibili da tutti gli utilizzatori del sistema. Forniscono delle risorse tramite l'API Gateway a chiunque le richiede.

3.2.1 API Gateway

L'API Gateway è uno strumento di gestione delle API che si trova tra un client e un insieme di servizi backend, che funge da reverse proxy. Il reverse proxy è un tipo di proxy che recupera i contenuti per conto dei client da uno o più servizi [33]. Tali contenuti saranno poi trasferiti al client, come se provenissero dallo stesso proxy, il quale appare al client come un server [10].

L'API gateway, infatti, riceve le richieste dei client tramite le chiamate API, aggrega le risposte dei vari servizi richiesti e restituisce un risultato appropriato. L'API Gateway semplifica la gestione delle API fornendo un unico punto di accesso per tutte le richieste HTTP dei client al sistema, rendendo più semplice la gestione e la documentazione delle API. Consente ai client del sistema di accedere ai servizi di un'applicazione basata su microservizi. Infatti, nel sistema in questione, si occuperà di gestire le richieste API inoltrandole/instradandole verso il microservizio di interesse [10].

Una delle particolarità di alcuni API Gateway è che, invece di fornire un'API unica per tutti i client, può esporre API differenti per client differenti. Per esempio, l'API Gateway di Netflix [29], esegue il codice di un adattatore specifico per il client in questione, il quale fornisce a ciascun client un'API più adatta ai suoi requisiti.

L'API Gateway ha la capacità di gestire il traffico in ingresso tramite il bilanciamento del carico. Tale proprietà consente di suddividere il carico tra le varie repliche di ogni servizio, consentendo di sfruttare la proprietà di scalabilità orizzontale dei microservizi (vedi Cap. 1.5.1).

L'API Gateway può essere configurato per autenticare e autorizzare le richieste API, proteggendo i servizi da eventuali minacce esterne; la capacità di verificare se ogni richiesta è stata autenticata o no, quindi, riesce a proteggere i servizi interni al sistema. L'API Gateway fornirà delle API protette da autenticazione per ogni servizio privato che si intende esporre agli utilizzatori. Il sistema progettato in questa tesi, sfrutta sia il pattern dell'architettura a microservizi che quello dell'API Gateway, dove l'API Gateway è l'unico punto di accesso al sistema per le richieste degli utilizzatori. Ciò consente di applicare il pattern dell'Access Token [9]; cioè l'API Gateway autentica le richieste e passa un access token, come per esempio il token JWT (JSON Web Token [23]), che identifica in modo sicuro l'utente che effettua ogni richiesta ai servizi. Un servizio può includere l'access token nella richiesta verso altri servizi. L'identità del richiedente viene trasmessa in modo sicuro al sistema. I servizi possono verificare che il richiedente sia autorizzato a eseguire un'operazione tramite l'analisi dell'access token [9].

L'API Gateway può essere configurato per gestire la cache delle richieste frequenti, migliorando le prestazioni e la velocità del sistema. È possibile salvare in una cache le risorse più richieste per fornire una risposta più veloce ai clienti che richiedono tale risorsa. Non sarà, infatti, necessario interrogare di nuovo il servizio per ottenere la risposta, ma basterà leggerla dalla memoria cache. Andranno tenuti in considerazione degli aspetti aggiuntivi quando ci si interfaccia con una cache, come l'invalidazione e l'aggiornamento. Andranno adottate tecniche di invalidazione e di aggiornamento della cache per consentire, al sistema,

di ottenere una risposta quanto più possibile vicina all'ultima versione del dato che si intende leggere.

L'API Gateway, dovendo gestire tutte le richieste dell'utente, dovrà essere un servizio performante e resiliente. In particolare non dovrà essere un single point of failure (SPOF)[1] del sistema. Un sistema che deve essere altamente disponibile ed affidabile deve essere privo di qualsiasi tipo di single point of failure, che sia un'applicazione software, un sistema industriale o un componente di un sistema distribuito. E' possibile garantire che l'API Gateway non sia un single point of failure per il sistema ridondando le repliche di tale componente.

3.2.2 Authentication Service

L'Autentication Service è il microservizio che permette agli utenti di autenticarsi. Tale servizio fornirà un'API che consentirà agli utenti di ottenere un token di accesso valido dopo essersi autenticati con successo. I servizi del sistema sono quindi protetti dall'API Gateway. E' bene definire come il servizio di autenticazione verificherà le credenziali dell'utente e come fornirà il token di accesso alle risorse.

Un ulteriore aspetto da considerare è come devono essere definiti i limiti di fiducia dei servizi del sistema in fase di identificazione degli utenti all'interno del sistema. E' possibile applicare un livello di fiducia Low Trust o Zero Trust, come mostrato nelle figure 3.3 e 3.4.

Nell'approccio Low Trust, i servizi del sistema si "fidano" di tutti i componenti della rete interna, l'API Gateway può mappare un token di accesso all'identificatore univoco di un utente e passarlo a un qualsiasi servizio interno insieme alle autorizzazioni che tale utente possiede [17] (vedi Fig. 3.3).

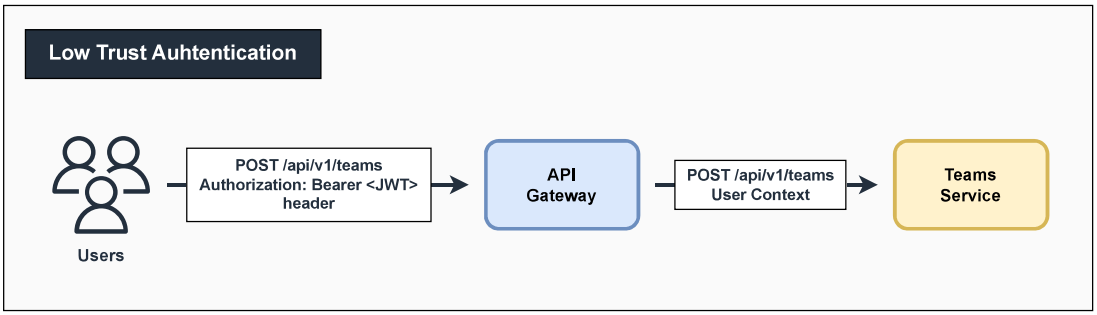

Figura 3.3: Approccio Low Trust nei sistemi distribuiti. Immagine adattata da [17].

Nell'approccio Zero Trust, i servizi del sistema considerano i componenti della rete interna come non fidati, l'API Gateway genererà e mapperà un token di accesso temporaneo all'identificatore univoco di un utente, passerà tale token temporaneo ad un qualsiasi servizio interno insieme alle autorizzazioni che tale utente possiede. Il servizio che riceve il token temporaneo verificherà il token comunicando con il servizio di autorizzazione. Il token temporaneo deve essere distrutto una volta che la richiesta è stata consumata. In tale maniera, i servizi potranno verificare gli utenti controllando dei token differenti da richiesta a richiesta, aumentando la sicurezza del sistema [17] (vedi Fig. 3.3).

Nel contesto del sistema distribuito, che si intende progettare, non sarà necessaria una politica di Zero Trust per le richieste interne, in quanto il sistema sarà eseguito in un ambiente

[1]Single Point Of Failure (SPOF): è una parte di un sistema che, se fallisce, interromperà il funzionamento dell'intero sistema.

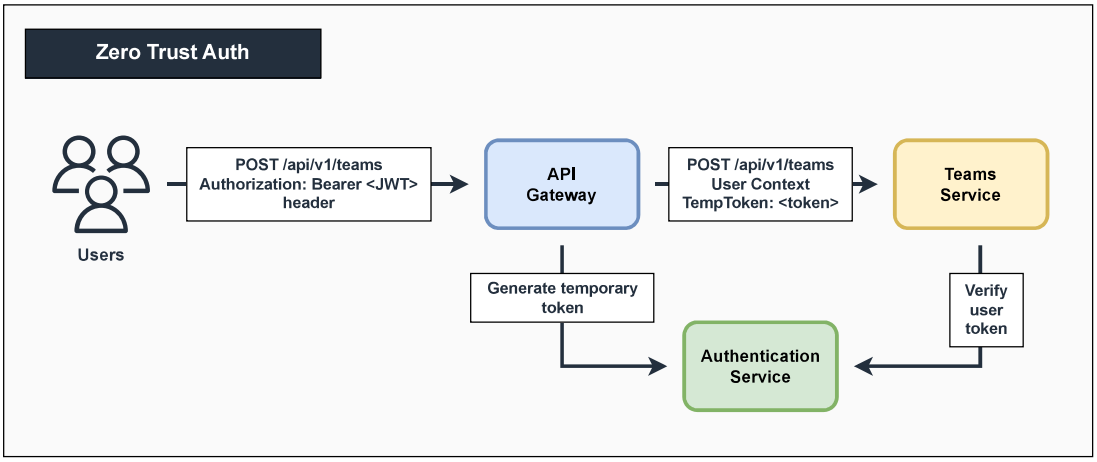

Figura 3.4: Approccio Zero Trust nei sistemi distribuiti. Immagine adattata da [17].

di rete isolato da potenziali attacchi esterni. Il sistema, quindi, adotterà una politica di Low Trust per le richieste autenticate degli utenti.

Gli utenti dovranno utilizzare il servizio di autenticazione solo per autenticarsi all'inizio della competizione; gli utenti, dopo aver ottenuto un token di accesso non dovranno più richiamare tale servizio. Rimarrà quindi inutilizzato per il resto della durata della competizione.

3.2.3 Frontend Service

Il servizio di frontend fornisce all'utente un sito web in di tipo Single Page Application che consentirà all'utente di interagire con le API del sistema tramite un'interfaccia grafica consultabile tramite un browser.

Una Single Page Application (SPA) è un'applicazione web che interagisce con l'utente riscrivendo dinamicamente la pagina web corrente con nuovi dati dal server Web, a differenza delle tradizionali applicazioni web che inviano intere nuove pagine all'utente. In una Single Page Application non si verifica mai un aggiornamento della pagina; invece, tutto il codice HTML, JavaScript e CSS necessario viene recuperato dal browser al primo caricamento della pagina, oppure le risorse appropriate vengono caricate dinamicamente e aggiunte alla pagina in un secondo momento di maniera asincrona [28].

L'utilizzo di una Single Page Application consente di ridurre il carico sul server web, perché il server è sollevato dalla responsabilità di generare contenuto dinamico; il server web dovrà, quindi, fornire solo risorse statiche.

Servizi privati del sistema

I servizi privati del sistema sono quei servizi che sono raggiungibili dai soli utenti autenticati del sistema. L'insieme di tali servizi è protetto dall'API Gateway tramite verifica dell'autenticazione tramite un token di accesso.

3.2.4 Exploits Service

L'Exploits Service è il servizio che si occupa di gestire gli exploit. Tale microservizio fornisce all'utente un insieme di API REST per la creazione, modifica e cancellazione degli exploit che andranno eseguiti dal sistema. Gli exploit creati potranno contenere bug ed errori. La loro esecuzione può portare ad un rallentamento del processo di esecuzione dei task nel sistema; sarà quindi necessario rimuovere tali task dalla pianificazione automatica.

Ogni exploit potrà essere riferito ad un servizio (vedi Cap. 3.2.5), oppure potrà essere un exploit generico. Un exploit è di tipo generico se non è stato studiato per attaccare un servizio specifico, ma come suggerisce il nome, ha uno scopo generale. Questo exploit non sarà pianificato per l'esecuzione in automatico dal sistema, ma sarà lanciato creando un task personalizzato ricorrente o non (vedi Cap. 3.2.9) che ne pianificherà l'esecuzione.

L'utente, in fase di creazione di un exploit, potrà caricare il file di exploit che andrà eseguito nel corso della competizione, file che verrà letto dal microservizio Job Executor (vedi Cap. 3.2.9) per poter poi essere eseguito.

3.2.5 Services Service

Il Services Service è il servizio che si occupa di gestire i servizi che sono esposti in ogni vulnbox di ogni team competitor. Le vulnbox delle squadre sono identiche. I servizi da attaccare, di conseguenza, saranno gli stessi per ogni team. Il servizio in questione fornisce all'utente un insieme di API REST per la creazione, modifica e cancellazione dei servizi da attaccare per ogni vulnbox.

In fase di creazione dei servizi, andranno inserite le porte che tale servizio esporrà. Ogni exploit otterrà le informazioni sul servizio da attaccare tramite una variabile d'ambiente che conterrà le informazioni sul servizio. Il microservizio incaricato di creare i job leggerà le informazioni dei servizi dal microservizio corrente.

3.2.6 Vulnboxes Service

Il Vulnboxes Service è il servizio che si occupa di gestire i dati delle vulnbox dei team competitor che si intendono attaccare. Le vulnbox delle squadre sono identificate da un nome e da un indirizzo IP. Il microservizio incaricato di creare i job leggerà le informazioni delle vulnbox da attaccare dal microservizio corrente.

3.2.7 Jobs Service

Il Jobs Service è quel microservizio che consente agli utenti di creare job personalizzati. Un job personalizzato non è nient'altro che una richiesta di esecuzione di un exploit. I job personalizzati possono essere di due tipi: ricorrenti o non ricorrenti.

I job personalizzati non ricorrenti sono dei task che vengono eseguiti una sola volta e non si ripetono periodicamente. Questi tipi di task richiedono una pianificazione specifica, poiché non possono essere automatizzati o ripetuti in modo regolare. I task ricorrenti, invece, saranno dei job che saranno eseguiti una volta per round di gioco.

In fase di creazione di un job personalizzato si può decidere se pianificare l'esecuzione dell'exploit contro una vulnbox specifica, contro tutte, oppure una esecuzione di tipo generico. Se si creerà un job che punterà ad attaccare tutte le vulnbox, verranno creati tanti job quante sono le vulbox. Le informazioni sul target dell'exploit saranno passate al programma tramite

una variabile d'ambiente. Creare un job che attacca una vulnbox o uno di tipo generico, implicherà la pianificazione di un solo job. Il Jobs Service si occuperà di creare job sia in maniera manuale, cioè tramite l'interazione dei client con le API esposte dal microservizio, sia in modo automatico, tramite la configurazione desiderata dei job.

Esternamente, il servizio, fornirà inoltre delle API per leggere le informazioni dei task relativi ad un round oppure filtrati per identificativo. Ciò consentirà all'utente di verificare se un job è stato pianificato, eseguito oppure no.

Internamente, la creazione di job in maniera automatica sarà dipendente dalla configurazione desiderata dei job. Il servizio si occuperà di aggiornare la configurazione rimanendo in ascolto per le modifiche effettuate sulle vulnbox, sugli exploits e sui servizi. Ogni volta che al servizio verrà notificata una alterazione dello stato di uno dei tre servizi appena nominati, esso modificherà la configurazione desiderata in base alla modifica di stato ricevuta.

In automatico, il servizio, si occuperà di creare i job appena prima dell'inizio del round successivo. In particolare verrà letta la configurazione calcolata in precedenza e verranno creati tanti job quanti specificati nella configurazione. Sarà necessario che la parte del microservizio che creerà i job sia capace di scalare per poter gestire la creazione di tutti i job, non limitando le performance degli altri servizi che interagiscono con i job.

Il servizio esporrà internamente due API per il job scheduler, le quali consentiranno a tale servizio di elencare i job che deve pianificare per l'esecuzione e di modificare il suo stato una volta pianificata.

Servizi interni del sistema

I servizi interni del sistema sono dei servizi che non sono raggiungibili dagli utenti del sistema, ma sono dei servizi che hanno uno scopo interno; tali servizi non espongono funzionalità agli utenti, ma verranno richiamati da altri microservizi o effettueranno chiamate API esterne al sistema.

3.2.8 Job Scheduler Service

Il Job Scheduler Service è il servizio interno che si occupa di accodare i job da eseguire. Tale servizio interroga l'API esposta dal Jobs Service per conoscere la lista dei job che deve mettere in coda. Il Job Scheduler deve assicurarsi di mettere in coda ogni job almeno una volta. Dopo aver messo in coda un task, il servizio, notificherà tale evento al sistema; in particolare il Job Service modificherà lo stato del job in questione. I task messi in coda saranno eseguiti dal servizio Job Executor Service.

Il microservizio può fallire ad accodare un job se la coda dei job non è disponibile nel momento dell'inserimento. Il sistema ritenterà di accodare i job in maniera asincrona, monitorando lo stato della coda. Quando la coda torna raggiungibile il servizio dovrà tentare di accodare nuovamente i job.

3.2.9 Job Executor Service

Il Job Executor Service è il servizio che si occupa di eseguire i task messi in coda dal Job Scheduler service. Tale microservizio si occuperà, inoltre, di recuperare i flag sottratti dall'esecuzione di ogni task. I flag sottratti andranno, poi, inviati al servizio Flags Service che si occuperà dell'invio al server di gioco (vedi Cap. 3.2.10). Il microservizio Job Executor, dopo

aver eseguito un exploit, notificherà lo stato di esecuzione al Jobs Service, il quale aggiornerà lo stato del task.

Come definito nel requisito funzionale **(RF-6)**, ogni task deve essere eseguito almeno una volta; infatti sarà necessario pianificare l'esecuzione di un job nuovamente se il worker che esegue i task andrà in crash. Se il nodo che esegue un task non riesce a notificare lo stato dell'esecuzione a causa di un crash, il sistema, non saprà se il task è stato eseguito con successo o no, sarà quindi necessario pianificare di nuovo l'esecuzione di tale task.

Il microservizio può fallire ad accodare un flag se la coda dei flag non è disponibile nel momento dell'inserimento. Il sistema ritenterà di accodare i flag in maniera asincrona, monitorando lo stato della coda di interesse. Nel momento in cui la coda torna raggiungibile il servizio dovrà tentare di accodare nuovamente i flag catturati.

3.2.10 Flags Service

Il servizio Flags Service si occupa dell'invio in maniera asincrona dei flag verso il server di gioco. Il microservizio dovrà inviare i flag sottratti dagli exploit precedentemente eseguiti e inviarli al server di gioco. L'invio dei flag verso il server di gioco può fallire, sarà infatti necessario ritentare l'invio dei flag verso il server di gioco nel caso in cui tale server sia non raggiungibile.

3.3 Scelta delle tecnologie

Dopo aver analizzato la topologia di rete dei servizi e i protocolli di comunicazione che utilizzano per scambiarsi informazioni, si ha un'idea chiara di quali tecnologie possono essere utilizzate per il sistema distribuito. In particolare, dal capitolo che analizza il contesto di esecuzione del sistema (vedi Cap. 2.1.4), si nota che esso deve essere eseguito sui computer dei componenti del team e che la disponibilità, la scalabilità e le performance del sistema sono i requisiti chiave del sistema.

Uno degli approcci migliori per sviluppare e progettare i microservizi di un sistema distribuito è l'approccio cloud-native. Le tecnologie cloud-native consentono di creare ed eseguire le applicazioni del sistema in ambienti moderni, capaci di fornire performance, scalabilità, affidabilità, manutenibilità, alta disponibilità e sicurezza. Tale approccio sfrutta appieno il modello di cloud computing, consentendo di creare ed eseguire applicazioni scalabili in ambienti moderni e dinamici [13]. I container, le mesh di servizi, i microservizi, l'infrastruttura non modificabile e le API dichiarative esemplificano questo approccio. Queste tecniche, consentono di progettare e sviluppare sistemi ad accoppiamento debole, resilienti, gestibili ed osservabili.

Nella progettazione del sistema, quindi, verrà utilizzata la tecnologia a microservizi e quella dei container (vedi Cap. 3.3.1). Per poter gestire i container nel sistema distribuito, viene utilizzata la piattaforma open-source Kubernetes (vedi Cap. 3.3.1). Molti dei servizi del sistema dovranno produrre degli eventi per scatenare delle azioni; alcune parti del sistema dovranno reagire a tali eventi per svolgere dei carichi di lavoro computazionali. La soluzione open-source Knative consente di creare applicazioni guidate ad eventi, servizi che saranno capaci di reagire agli eventi del sistema (vedi Cap. 3.3.2).

Dallo studio effettuato nell'analisi dei volumi e dei requisiti funzionali e non (vedi Cap. 2.3), si è dedotto che si ha bisogno di database performanti e scalabili orizzontalmente, con

capacità di auto guarigione. I diversi requisiti di storicizzazione e memorizzazione possono essere soddisfatti utilizzando i database Cassandra (vedi Cap. 3.3.3) e TiKV (vedi Cap. 3.3.4).

Lo storage ad oggetti per storicizzare gli exploit da eseguire dovrà essere altamente disponibile e distribuito su più nodi; Rook.io consentirà di soddisfare tali requisiti (vedi Cap. 3.3.6).

3.3.1 Kubernetes

Kubernetes è una piattaforma open-source per la gestione dei container, progettata per semplificare la distribuzione, l'orchestrazione e la gestione dei sistemi distribuiti basati su tecnologia a container. La piattaforma è stata creata in modo tale da permettere agli sviluppatori di concentrarsi sulla creazione di funzionalità per le loro applicazioni distribuitre, senza preoccuparsi delle complessità sottostanti della gestione di un ambiente cloud.

Kubernetes riesce a rilevare quali container sono andati in crash o sono non funzionali tramite un controllo dello stato definito dall'utente. Kubernetes è capace di sostituire e pianificare nuovamente i container quando i nodi muoiono.

Kubernetes gestisce automaticamente la scoperta dei servizi; assegna ai container degli indirizzi IP e un singolo nome DNS per ogni servizio creato, consentendo di bilanciare il carico tra di essi; è capace di scalare le applicazioni orizzontalmente e verticalmente in maniera automatica o manuale. I container possono scalare in base all'utilizzo della CPU dei container del servizio in questione. Tramite la gestione automatica dei container assicura che le applicazioni siano sempre disponibili e scalabili, facendo in modo che ogni container riceva le risorse di cui ha bisogno per funzionare correttamente.

Kubernetes è progettato per essere facilmente estendibile e personalizzabile, dispone di una comunità attiva che sviluppa continuamente nuove funzionalità e integrazioni. Kubernetes è utilizzato da molte aziende di tutto il mondo per gestire grandi infrastrutture distribuite e applicazioni cloud-native [25].

La risorsa personalizzata costituisce il meccanismo di estensione dell'API in Kubernetes. Una definizione di risorsa personalizzata (CRD: Custom Resource Definitions) definisce la risorsa personalizzata dell'utente ed elenca tutte le configurazioni disponibili a chi utilizza l'operatore. Gli operatori Kubernetes possono introdurre nuovi tipi di oggetti tramite le CRD, che saranno gestite dall'API Kubernetes come oggetti integrati.

Container

Un container è un'unità software standardizzata che impacchetta il codice e tutte le sue dipendenze in modo tale che l'applicazione venga eseguita in maniera rapida e affidabile. Un'immagine del container è un pacchetto di software leggero, autonomo ed eseguibile che include tutto il necessario per eseguire un'applicazione: codice, runtime, strumenti di sistema, librerie di sistema e impostazioni [18].

Le immagini dei container diventano container in fase di esecuzione, quando sono eseguite da un runtime per container. Il software containerizzato sarà, quindi, indipendente dall'infrastruttura su cui viene eseguito. I container isolano il software applicativo dall'ambiente in cui vengono eseguiti, consentendo di assicurare che i container funzionino in modo uniforme nonostante le differenze architetturali.

L'Open Container Initiative [30], è una struttura di governance aperta con il preciso scopo di creare standard di settore aperti attorno a formati di container e runtime. Tale struttura ha

proposto tre standardizzazioni per i container, la specifica per il runtime, delle immagini e quella per la loro distribuzione.

CRI-O

CRI-O è un'implementazione dell'interfaccia di runtime per container di Kubernetes (CRI: Container Runtime Interface) per contentire di utilizzare runtime compatibili con l'Open Container Initiative. CRI-O è un'alternativa leggera all'utilizzo di Docker come runtime per Kubernetes. Consente a Kubernetes di utilizzare qualsiasi runtime conforme all'Open Container Initiative come runtime del contenitore per l'esecuzione dei pod[2]. CRI-O supporta le immagini standardizzate dall'Open Container Initiative e può eseguire il pull da qualsiasi registro di container [15].

3.3.2 Knative

Knative è una soluzione open-source per la gestione di applicazioni serverless (vedi capitolo successivo) su Kubernetes. Fornisce funzionalità per la gestione della scalabilità automatica, delle risorse, della distribuzione e delle comunicazioni tra componenti di un'applicazione. Questo rende più semplice e scalabile l'esecuzione di applicazioni cloud-native su più piattaforme [24].

Utilizzando Knative, gli sviluppatori possono concentrarsi sulla creazione di funzionalità per le loro applicazioni, senza preoccuparsi delle complessità sottostanti della gestione di un ambiente cloud-native. Knative, inoltre, è stato progettato per essere platform-indipendent, cioè le applicazioni create con questo strumento possono essere eseguite su più piattaforme cloud.

Knative è una soluzione composta principalmente da due componenti, che consentono ai team di utilizzare Kubernetes: Knative Serving e Knative Eventing (introdotte nei prossimi capitoli). Queste due componenti lavorano insieme per automatizzare e gestire task e applicazioni tramite lo scambio di messaggi in maniera sincrona e non [24].

Serverless

La computazione serverless consente di creare ed eseguire applicazioni e servizi senza pensare al provisioning, alla scalabilità e alla gestione di server [4]. La computazione serverless è un modello di esecuzione del cloud computing in cui il fornitore di servizi cloud alloca dinamicamente le risorse per eseguire il codice di un'applicazione in risposta a richieste o eventi in arrivo.

Tale modello computazionale offre numerosi vantaggi, tra cui una maggiore flessibilità, una riduzione dei costi e una gestione semplificata delle risorse. E' particolarmente adatto per l'elaborazione di picchi di carico e per le applicazioni che richiedono una capacità di elaborazione dinamica poiché il codice viene eseguito su richiesta [4].

Knative Functions

Knative Functions fornisce un semplice modello di programmazione per l'utilizzo delle funzioni su Knative, senza richiedere una conoscenza approfondita di Knative, di Kubernetes,

[2]I pod sono le unità di calcolo distribuibili più piccole che si può creare e gestire in Kubernetes.

dei container o dei Dockerfile[3]. Knative Functions consente di creare, costruire e distribuire facilmente funzioni stateless[4] e guidate da eventi.

Knative Serving

Knative Serving è un componente di Knative, definita da un set di oggetti come Kubernetes Custom Resource Definitions (CRD, vedi Cap. 3.3.1). Queste risorse vengono utilizzate per definire e controllare il comportamento del carico di lavoro serverless nel cluster. Knative Serving è progettato per gestire la distribuzione e l'esecuzione di funzioni e applicazioni serverless, fornendo un insieme di componenti per la gestione delle risorse, la scalabilità automatica e la gestione della disponibilità.

Con Knative Serving, gli sviluppatori possono caricare e distribuire facilmente le loro funzioni e applicazioni serverless su Kubernetes, senza doversi preoccupare della gestione dell'infrastruttura sottostante. Knative Serving fornisce anche funzionalità avanzate per la gestione delle richieste, che possono aiutare a migliorare la disponibilità e le prestazioni delle applicazioni.

Knative Eventing

Knative Eventing è un insieme di API che consente di utilizzare un'architettura basata sugli eventi nelle applicazioni. Si possono utilizzare queste API per creare componenti che indirizzano gli eventi dai produttori ai consumatori di eventi. I consumatori di eventi possono anche essere configurati per rispondere alle richieste HTTP inviando un evento di risposta.

Knative Eventing utilizza richieste HTTP POST standard per inviare e ricevere eventi tra produttori e consumatori di eventi. Questi eventi sono conformi alle specifiche CloudEvents, che consentono di creare, analizzare, inviare e ricevere eventi in qualsiasi linguaggio di programmazione [14]. I componenti di Knative Eventing sono liberamente accoppiati e possono essere sviluppati e implementati indipendentemente l'uno dall'altro. Qualsiasi producer può generare eventi prima che vi siano consumer di eventi attivi che stanno ascoltando tali eventi. Qualsiasi consumatore di eventi può esprimere interesse per una classe di essi anche prima che ci siano produttori che li stiano creando.

3.3.3 Apache Cassandra

Apache Cassandra è un database distribuito NoSQL[5] progettato per gestire grandi quantità di dati distribuiti in modo affidabile. Il database Cassandra offre elevata disponibilità e scalabilità orizzontale, potendo gestire facilmente l'aumento della quantità di dati e del carico di lavoro [7].

Cassandra utilizza un modello di replicazione peer-to-peer, in cui i dati vengono replicati su più nodi nella rete, garantendo tolleranza ai guasti e disponibilità dei dati. Cassandra permette di scalare orizzontalmente e ridondare i dati; ciò aiuta a garantire la disponibilità dei dati anche in caso di guasti dei nodi. Apache Cassandra ha capacità di auto guarigione se

[3]Un Dockerfile è un documento di testo che contiene tutti i comandi che servono per assemblare un'immagine container [18].

[4]Un'applicazione stateless è definita "senza stato" in quanto non salva i dati generati in una determinata sessione per utilizzarli nel corso di quelle successive.

[5]Un database NoSQL, non relazionale, fornisce un meccanismo per l'archiviazione e il recupero di dati modellati in mezzi diversi dalle relazioni tabulari utilizzate nei database relazionali.

un nodo va offline o se c'è un guasto a livello di hard disk in un nodo. Tali caratteristiche rendono Cassandra il database adatto a soddisfare qualche caso d'uso dell'applicazione. La sua architettura distribuita e scalabile lo rende adatto a molti casi d'uso in cui la disponibilità dei dati è critica e la quantità di dati è in continua crescita [7].

3.3.4 TiKV

TiKV è un database NoSQL open source di tipo chiave-valore distribuito, altamente scalabile, a bassa latenza, facile da usare e transazionale. A differenza di altri database NoSQL tradizionali, TiKV non fornisce solo le classiche API chiave-valore, ma anche API transazionali con conformità ACID [35].

TiKV utilizza l'algoritmo Raft per supportare la replica geografica ed eccellere nella scalabilità orizzontale, potendo scalare facilmente fino a oltre 100 TB di dati. Il database chiave valore, inoltre, supporta transazioni distribuite coerenti dall'esterno. Anche TiKV, come Apache Cassandra, è autorigenerante, consentendo un recupero dei dati automatico [35].

3.3.5 RabbitMQ

RabbitMQ è il broker di messaggi open source più diffuso; fornisce una piattaforma affidabile e scalabile per l'invio e la ricezione di messaggi. RabbitMQ utilizza il modello di messaggistica produttore-consumatore per consentire alle applicazioni di inviare e ricevere messaggi attraverso una rete di broker (middleware) [32].

E' altamente affidabile e disponibile, poiché i messaggi possono essere replicati su più nodi nella rete e possono essere configurati per garantire la consegna affidabile dei messaggi anche in caso di guasti. RabbitMQ supporta inoltre la scalabilità orizzontale, ovvero la capacità di gestire facilmente l'aumento della quantità di messaggi e del carico di lavoro. La sua architettura flessibile e scalabile lo rende adatto a molte use case in cui la gestione affidabile e scalabile dei messaggi è critica [32].

RabbitMQ è una delle opzioni di broker supportate da Knative Eventing. Nella progettazione del sistema distribuito verrà utilizzato RabbitMQ come message broker per gli eventi di Knative Eventing. RabbitMQ offre anche una vasta gamma di funzionalità per la gestione delle code e la persistenza e replica di messaggi, che possono essere utilizzate per rendere il broker altamente disponibile e scalabile. In tale modo è possibile costruire soluzioni di elaborazione degli eventi altamente affidabili con Knative Eventing [24].

3.3.6 Rook

Rook è un orchestratore di storage cloud-native open source, che fornisce la piattaforma, il framework e il supporto per lo storage Ceph per l'integrazione con gli ambienti cloud-native. Ceph è un sistema di archiviazione distribuito che fornisce archiviazione di file, storage a blocchi e ad oggetti. E' un sistema che è distribuito in cluster di produzione su larga scala. Rook automatizza l'implementazione e la gestione dello storage Ceph per fornire servizi di storicizzazione autogestiti, scalabili e con capacità di auto guarigione. L'operatore Rook esegue questa operazione basandosi sulle risorse Kubernetes per distribuire, configurare, eseguire il provisioning[6], ridimensionare, aggiornare e monitorare Ceph [34].

[6]Provisioning: è il processo di configurazione di un'infrastruttura informatica.

3.3.7 Emissary-Ingress

Emissary-Ingress è un API Gateway open-source nativo di Kubernetes per microservizi basato su Envoy Proxy. Emissary consente di gestire il traffico in ingresso bilanciando il carico e integrandosi nativamente con Kubernetes. Protegge i microservizi con un sistema di autenticazione, di limitazione delle richieste (rate limiting) e interruzione del circuito (circuit breaking). Le integrazioni con Grafana, Prometheus e Datadog favoriscono l'osservabilità e il supporto completo delle metrice. Emissary è integrato con Knative Eventing per distribuire il carico a container serverless [19].

CAPITOLO 4

Progettazione a basso livello

Nel capitolo in questione, viene trattata la progettazione a basso livello del sistema distribuito. si analizzano tutti i microservizi del sistema, tra cui: Authentication Service, Frontend Service, Exploits Service, Services Service, Vulnboxes Service, Jobs Service, Job Scheduler Service, Job Executor Service, Flags Service e API Gateway. Per ogni microservizio saranno analizzate le varie componenti che ne fanno parte, le caratteristiche delle componenti, le strategie di progettazione, gli eventi prodotti e consumati e la modellazione dei dati del database.

Dopo aver descritto la progettazione ad alto del sistema che si intende progettare, è necessario studiare la struttura e le funzionalità a basso livello. L'architettura a basso livello è necessaria per comprendere come ogni servizio andrà implementato e come si integrerà con il resto dell'architettura. In questa fase progettuale si dovrà approfondire ogni servizio dell'architettura ad alto livello, definendo le API del sistema e la modellazione dei database.

Considerazioni sui servizi serverless

Si evidenzia che ogni container serverless verrà distribuito con Knative Serving (vedi Cap. 3.3.2) e che ogni funzione sarà implementata con Knative Functions (vedi Cap. 3.3.2). I container che sono utilizzatori e/o produttori di eventi utilizzeranno le funzionalità di Knative descritte nel capitolo 3.3.2 su Knative Eventing.

Le funzioni Knative dovranno scalare in automatico in base al numero di richieste ricevute ogni secondo. In tale modo possono garantire una risposta rapida agli utilizzatori del servizio in questione, evitando un degrado di performance.

Ridurre i tempi di avvio delle funzioni e dei container serverless è fondamentale per migliorare la reattività del sistema. Il tempo di avvio a freddo (o cold start) è il tempo necessario per Kubernetes per "accendere" e rendere operativo un container. Andranno adottate delle strategie per evitare l'avvio a freddo o ridurre le tempistiche di cold start [31]. Paul Schweigert e Carlos Santana hanno presentato alcune strategie per ridurre le tempistiche di avvio a freddo alla KnativeCon del 2022 [27], tra cui:

- Minimizzare la dimensione delle immagini.
- Evitare linguaggi di programmazione interpretati, i quali risultano essere più lenti in fase di avvio ed esecuzione di uno script.
- Mantenere attivi quei servizi che devono rimanere sempre online. Ridurre il numero di repliche ad una per evitare l'avvio a freddo; così ci sarà almeno una copia dell'applicazione pronta a servire le prime richieste in arrivo.

- Aumentare la finestra temporale per la scalabilità a zero repliche. In caso di traffico ridotto è conveniente scalare il servizio fino a raggiungere una replica, ma bisognerà attendere un periodo maggiore per terminare l'ultima replica.

Considerazioni sul database Cassandra

Per quanto riguarda il database Cassandra, è possibile soddisfare i requisiti di prestazioni, affidabilità, scalabilità e alta disponibilità con un sistema di storicizzazione *eventualmente consistente*[1]. Possono esistere delle versioni divergenti degli stessi dati temporaneamente, ma alla fine convergeranno nello stesso stato e saranno consistenti. La *consistenza eventuale* è un compromesso per ottenere un'elevata disponibilità e che può portare a sperimentare latenze in fase di lettura e scrittura. I database Cassandra saranno configurati con una replicazione dei dati in due nodi. In tale modo è possibile garantire l'alta disponibilità del servizio, mantenendo una buona consistenza dei dati e una latenza ridotta.

Considerazioni sul database TiKV

Per il database TiKV, è possibile soddisfare i requisiti di prestazioni, scalabilità e alta disponibilità sincronizzando i dati tra tre nodi. Se un nodo master diventerà irraggiungibile, il sistema eleggerà un nuovo master in circa venti secondi, un tempo in cui il sistema non potrà accedere ai database TiKV. Tuttavia, la rapida elezione del nuovo nodo master consente di poter gestire il fallimento di un nodo ritentando l'operazione di lettura o scrittura con un ritardo variabile tra i cinque e i trenta secondi.

Considerazioni sul broker RabbitMQ

RabbitMQ è il message broker utilizzato da Knative Eventing per scambiare gli eventi e come coda per processare in modo asincrono le risorse. Il sistema dovrà essere quindi altamente disponibile. Per ottenere l'alta disponibilità del servizio è possibile connettere più nodi per formare un cluster RabbitMQ. Il clustering connette più macchine insieme, replicando automaticamente gli host virtuali, gli utenti e le autorizzazioni. I nodi di clustering possono aiutare a migliorare la disponibilità, la sicurezza dei dati dei contenuti della coda e sostenere più connessioni client simultanee.

Le code di RabbitMQ, invece, possono trovarsi su un singolo nodo o replicare il loro contenuto per una maggiore disponibilità. Le *Quorum Queues*, sono un tipo moderno di coda replicata tra i nodi del cluster. Un client che si connette a qualsiasi nodo del cluster può utilizzare tutte le code di esso, anche se non si trovano su quel nodo.

Considerazioni sullo storage condiviso Rook

Rook è un orchestratore di storage cloud-native open source, che fornisce la piattaforma, il framework e il supporto per lo storage Ceph. Per ottenere un'alta disponibilità del servizio di storage ad oggetti di Rook c'è bisogno di creare un cluster Ceph. Il cluster dovrà essere composto da tre nodi, i quali saranno monitorati da tre *Ceph Monitor*.

[1] L'eventual consistency è un modello di consistenza dei dati utilizzato in sistemi distribuiti. Il termine "eventualmente consistente" si riferisce alla garanzia che, sebbene i dati possano essere temporaneamente inconsistenti tra diverse parti del sistema, alla fine convergeranno nello stesso stato e saranno consistenti.

4.1 Authentication Microservice

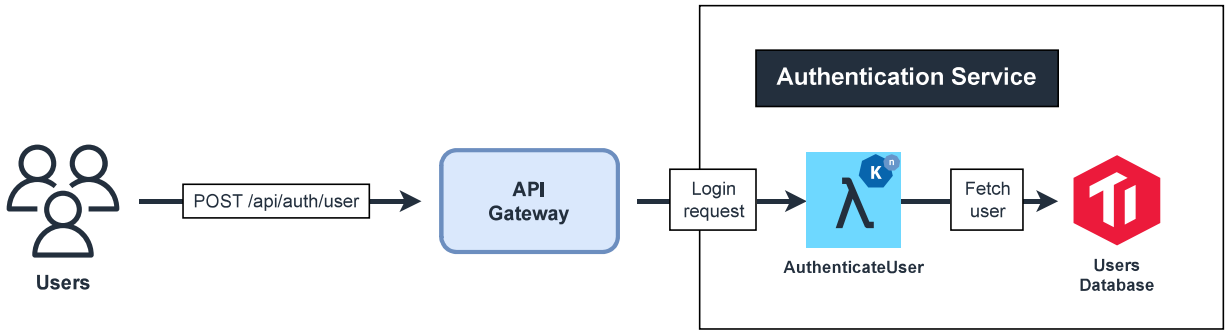

Figura 4.1: Architettura a basso livello del microservizio: Authentication Service

Come descritto nel capitolo 3.2.2, *Authentication Service* è il microservizio che permette agli utenti di autenticarsi al sistema. Si studiano le API del servizio, le risorse e la modellazione del database.

API del microservizio

Nella seguente tabella sono espresse le API esposte dal microservizio.

Tipo risorsa	Metodo HTTP	Endpoint	Descrizione risorsa	Risorsa
API pubblica	**POST**	**/api/auth/user**	Autentica un utente tramite le credenziali fornite e resituisce un token di accesso JWT	**AuthenticateUser**

Tabella 4.1: API del microservizio Authentication Microservice.

4.1.1 AuthenticateUser Function

Il componente *AuthenticateUser* sarà una funzione serverless implementata con Knative Functions (vedi Cap. 3.3.2) e distribuita con Knative Serving (vedi Cap. 3.3.2).

La funzione, sarà responsabile di verificare le credenziali degli utenti e restituirà un token di accesso JWT, il quale verrà utilizzato dall'utente per autenticare le successive richieste. Le credenziali da verificare vengono lette dalla funzione *AuthenticateUser* nel database *Users Database*. In caso di autenticazione fallita verrà inviato un messaggio di errore all'utente.

La funzione in questione è un servizio senza stato; il numero di repliche, quindi, può essere scalato e gestito da Knative Serving (vedi Cap. 3.3.2). Poiché il servizio di autenticazione verrà utilizzato solo all'inizio della gara e occasionalmente durante il corso, è possibile scalare le repliche della funzione serverless a zero dopo un certo periodo di inattività. Ad esempio, la funzione può essere distribuita in due repliche e, dopo dieci minuti di inutilizzo, può essere scalata a zero repliche. In questo modo, si può preservare l'utilizzo delle risorse all'interno del cluster.

Modellazione dei dati del database

Il salvataggio delle credenziali degli utenti avviene mediante l'utilizzo del database *Users Database*. Le azioni che verranno eseguite nel database sono:

- Leggere le credenziali di un utente in base al suo username.
- Inserire le credenziali di un utente.

Vista la tipologia delle operazioni di lettura e scrittura e la necessità di avere il servizio di autenticazione altamente disponibile, è possibile memorizzare le credenziali utente in un database TiKV chiave-valore. La chiave delle righe del database corrisponderà con l'username dell'utente, mentre il valore conterrà la password criptata.

Ecco un esempio di come i dati potranno essere storicizzati nel database:

```
{
  "SamuPert": "$2a$12$IJtCXExGWRCizjXo3Q5pvewo9bbwxDOfQkgSoW.MmaC5pYXhSOBiu",
  "AnotherUser": "$2a$12$as3l3zXGsPceqHkfoG/rZuUz/cOfzE3Nf1TXeIq5Of/BwvSZEHjoS"
}
```

Listing 4.1: Esempio dati: Users Database

4.2 Frontend Microservice

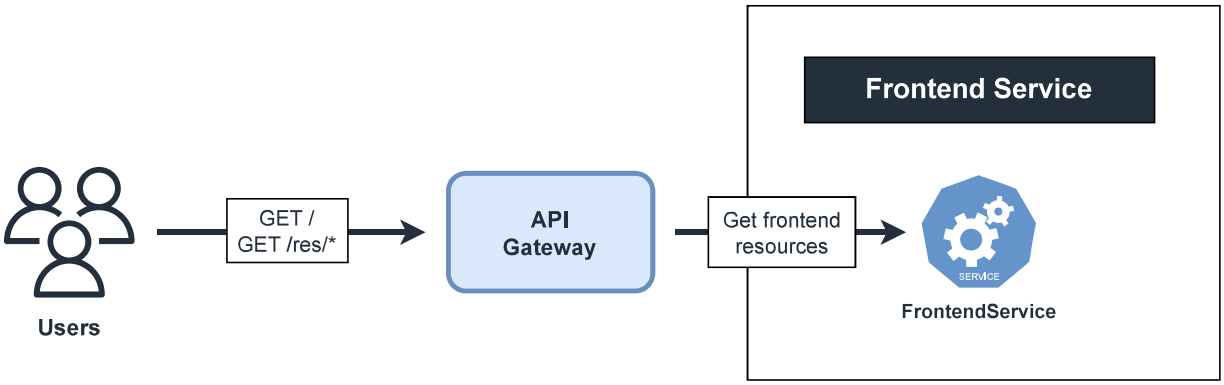

Figura 4.2: Architettura a basso livello del microservizio: Frontend Service

Il Frontend Service è un microservizio che fornisce agli utenti del sistema un sito web a pagina singola (Single Page Application), che permette al team di interagire con le API attraverso un'interfaccia grafica accessibile tramite un browser (vedi Cap. 3.2.3). Nelle prossime sezioni, verranno analizzate le risorse del servizio.

Risorse del microservizio

Nella seguente tabella sono espresse le risorse esposte dal microservizio.

Tipo risorsa	Metodo HTTP	Endpoint	Descrizione risorsa	Risorsa
Pagina web	**GET**	**/**	Restituisce la homepage del sito web	**FrontendService**
Risorse statiche	**GET**	**/res/***	Restituisce le risorse statiche del sito	**FrontendService**

Tabella 4.2: Risorse del microservizio Frontend Microservice.

4.2.1 FrontendService

Il componente FrontendService sarà un container serverless che fornirà tutte le risorse necessarie per il corretto funzionamento del sito web. Una delle immagini container più leggere per il servizio di risorse statiche è l'immagine Nginx basata su Alpine (*mainline-alpine-slim*), ottenibile dal Docker Hub al seguente url `https://hub.docker.com/_/nginx/tags`. Andrà creata un'immagine personalizzata dal template *mainline-alpine-slim* che conterrà le risorse statiche che si intende servire.

Tale componente sarà un servizio senza stato e dovrà essere sempre disponibile. Il traffico stimato verso tale servizio è ridotto, non dovrebbe essere quindi necessario scalare a più di due repliche il servizio.

4.3 Vulnboxes Microservice

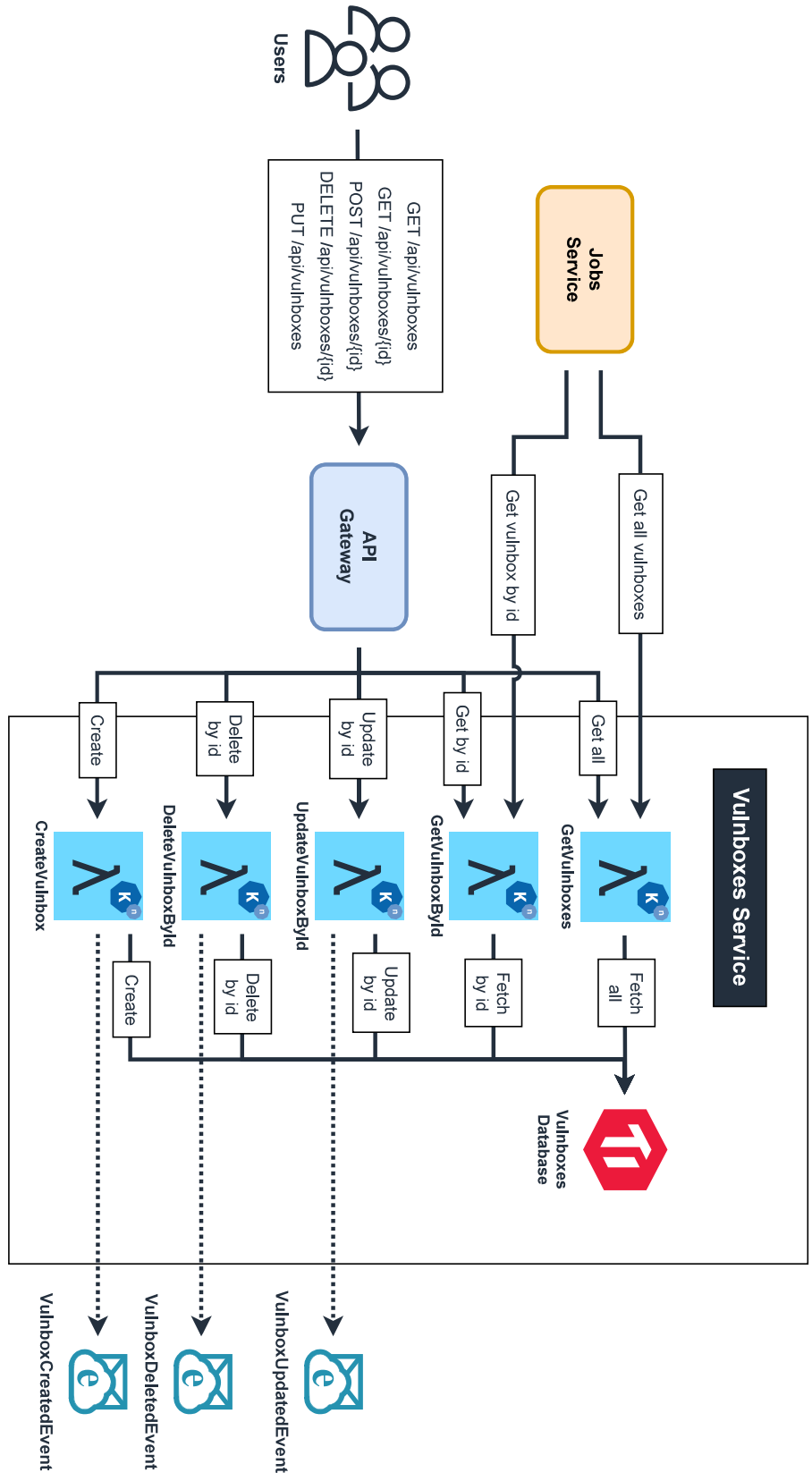

Figura 4.3: Architettura a basso livello del microservizio: Vulnboxes Service

Come descritto nel capitolo 3.2.6, Vulnboxes Service è il microservizio che si occupa di gestire i dati delle vulnbox dei team competitor che si intendono attaccare. Verrà gestita la creazione, la modifica e l'eliminazione di team. Si studiano le API del servizio, le risorse, la modellazione del database e gli eventi prodotti e consumati.

API del microservizio

Nella seguente tabella sono espresse le API esposte dal microservizio.

Tipo risorsa	Metodo HTTP	Endpoint	Descrizione risorsa	Risorsa
API privata e interna	**GET**	**/api/vulnboxes**	Legge la lista delle vulnbox	**GetVulnboxes**
API privata e interna	**GET**	**/api/vulnboxes/{id}**	Legge una vulnbox per l'identificativo	**GetVulnboxById**
API privata	**POST**	**/api/vulnboxes/{id}**	Modifica i metadati di una vulnbox per l'identificativo	**UpdateVulnboxById**
API privata	**DELETE**	**/api/vulnboxes/{id}**	Elimina una vulnbox per l'identificativo	**DeleteVulnboxById**
API privata	**PUT**	**/api/vulnboxes**	Crea i metadati di una vulnbox	**CreateVulnbox**

Tabella 4.3: API del microservizio Vulnboxes Microservice.

4.3.1 GetVulnboxes Function

GetVulnboxes è la funzione serverless che si occupa di fornire una lista delle vulnbox inserite nel sistema. La funzione è un componente senza stato che legge i dati delle vulnbox dal database *Vulnboxes Database*. Il traffico stimato verso tale funzione è ridotto, non dovrebbe essere quindi necessario scalare a più di due repliche la funzione. La funzione, inoltre, sarà utilizzata saltuariamente; sarà quindi necessario scalare le repliche della funzione a zero dopo un certo tempo di inutilizzo. Per esempio, la funzione, può essere distribuita in due repliche e dopo cinque minuti di inattività può scalare a zero repliche.

4.3.2 GetVulnboxById Function

GetVulnboxById è la funzione serverless che legge, tramite l'identificativo specificato, i dati di una particolare vulnbox dal database *Vulnboxes Database*. Tale componente sarà un servizio senza stato che riceverà volumi di traffico ridotti. La funzione andrà scalata a due repliche, non dovrebbe essere quindi necessario scalare a più di due repliche la funzione. La funzione sebbene verrà utilizzata saltuariamente, sarà necessario mantenere attiva almeno una replica per velocizzare i tempi di cold start.

4.3.3 UpdateVulnboxById Function

UpdateVulnboxById è la funzione serverless che modifica, tramite l'identificativo specificato, i dati di una particolare vulnbox nel database *Vulnboxes Database*. Tale componente senza stato sarà un servizio che riceverà volumi di traffico ridotti e sarà utilizzata saltuariamente. La funzione può essere distribuita in una replica e dopo cinque minuti di inattività può scalare a zero repliche. È necessario scalare le repliche della funzione a zero dopo un certo tempo di inutilizzo.

4.3.4 DeleteVulnboxById Function

DeleteVulnboxById è la funzione serverless che elimina, tramite l'identificativo specificato, una particolare vulnbox dal database *Vulnboxes Database*. Tale componente sarà un servizio senza stato che riceverà volumi di traffico ridotti e sarà utilizzata saltuariamente. La funzione può essere distribuita in una replica e dopo cinque minuti di inattività può scalare a zero repliche. È necessario scalare le repliche della funzione a zero dopo un certo tempo di inutilizzo.

4.3.5 CreateVulnbox Function

CreateVulnbox è la funzione serverless che crea le vulnbox da inserire nel database *Vulnboxes Database*. Il servizio di creazione delle vulnbox verrà utilizzato solo all'inizio della competizione e saltuariamente durante essa, è possibile scalare le repliche della funzione serverless a zero dopo un certo tempo di inutilizzo. La funzione andrà scalata a una replica e andrà terminata dopo cinque minuti di inutilizzo.

Eventi

UpdateVulnboxById

La funzione UpdateVulnboxById produrrà un evento di tipo *VulnboxUpdatedEvent* quando avviene la modifica di una vulnbox. Tale evento scatenerà l'esecuzione dell'aggiornamento della configurazione dei job nel microservizio *Jobs Microservice* (vedi Cap. 4.6).

DeleteVulnboxById

La funzione DeleteVulnboxById produrrà un evento di tipo *VulnboxDeletedEvent* quando avviene la cancellazione di una vulnbox. Tale evento scatenerà l'esecuzione dell'aggiornamento della configurazione dei job nel microservizio *Jobs Microservice* (vedi Cap. 4.6).

CreateVulnbox

La funzione CreateVulnbox produrrà un evento di tipo *VulnboxCreatedEvent* quando si crea una vulnbox, da cui si avvierà l'esecuzione dell'aggiornamento della configurazione dei job nel microservizio *Jobs Microservice* (vedi Cap. 4.6).

Modellazione dei dati del database

Per il salvataggio dei metadati delle vulnbox dei team da attaccare verrà utilizzato il database *Vulnboxes Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Leggere la lista delle vulnbox.
- Leggere una vulnbox in base al suo id.
- Modificare una vulnbox in base al suo id.
- Creare una vulnbox.
- Cancellare una vulnbox.

Le operazioni di lettura sono semplici e basate su un interrogazione per chiave o sulla lettura completa del database. È possibile utilizzare il database TiKV chiave-valore per salvare i dati delle vulnbox. La chiave delle righe del database corrisponderà all'identificativo delle vulnbox, mentre il valore conterrà i suoi dati.

Ecco un esempio di come i dati potranno essere storicizzati nel database:

```
{
  "eb0bd9db-a9bf-453d-be88-424ec782fac0": {"name":"polito","ip":"10.10.1.1"},
  "cbc2cdb0-294d-4d83-aabb-8c697587e2e7": {"name":"unicam","ip":"10.20.1.1"},
  "3dac9993-7b9f-40c1-ace7-cccc9842c416": {"name":"unimi","ip":"10.30.1.1"}
}
```

Listing 4.2: Esempio dati: Vulnboxes Database

Nota bene: TiKV supporta il salvataggio di valori di tipo stringa, non oggetto. Nell'esempio appena mostrato i dati sono inseriti come oggetti per facilitarne la lettura. A livello pratico i dati delle vulnbox andranno storicizzati come stringhe.

4.4 Services Microservice

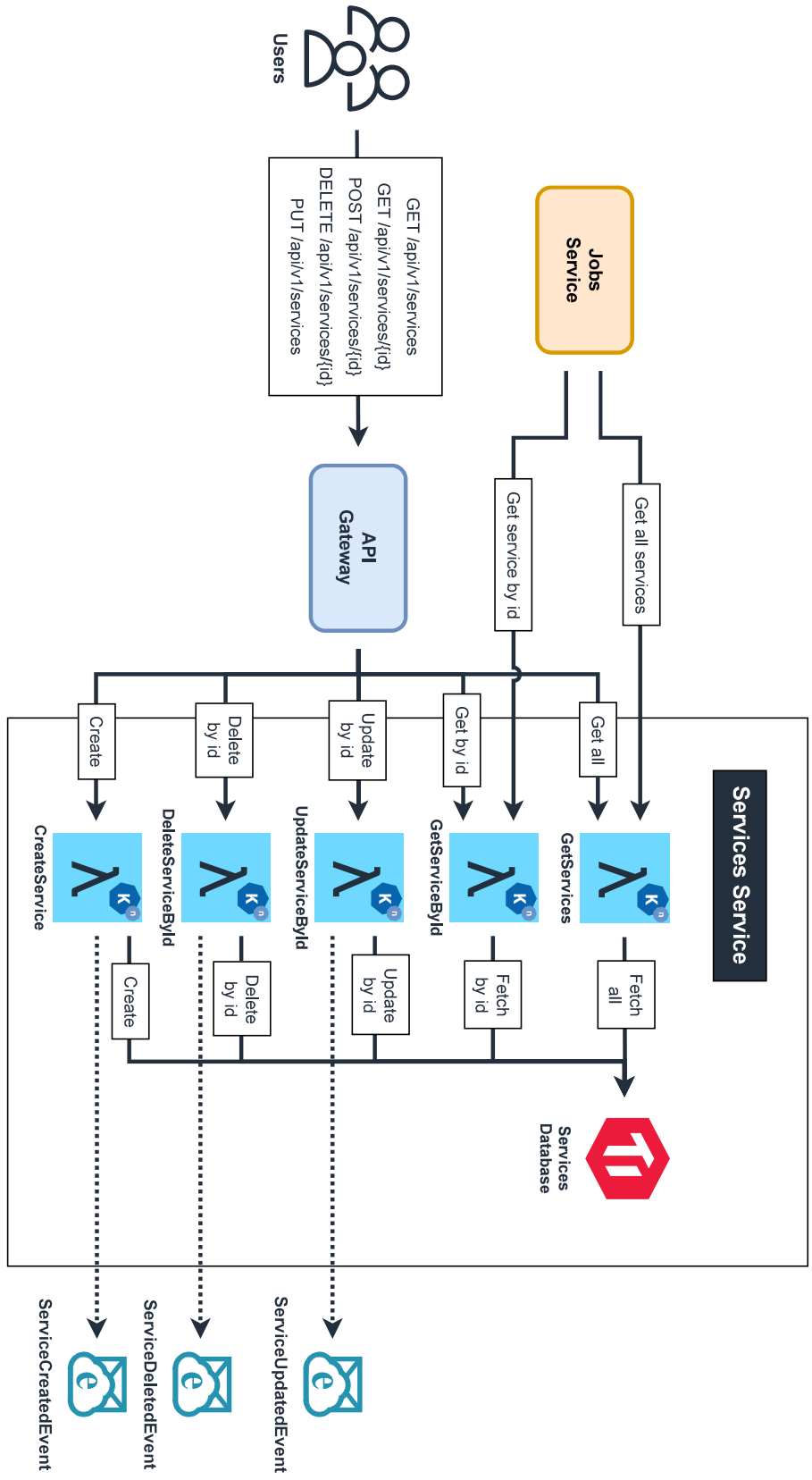

Figura 4.4: Architettura a basso livello del microservizio: Services Microsevice

Come descritto nel capitolo 3.2.5, *Services Microsevice* è il microservizio che si occupa di gestire i dati dei servizi dei team competitor che si intendono attaccare. Verrà gestita la creazione, la modifica e l'eliminazione di team. Si studiano le API del servizio, le risorse, la modellazione del database e gli eventi prodotti e consumati.

API del microservizio

Nella seguente tabella sono espresse le API esposte dal microservizio.

Tipo risorsa	Metodo HTTP	Endpoint	Descrizione risorsa	Risorsa
API privata e interna	**GET**	**/api/services**	Legge la lista dei servizi	**GetServices**
API privata e interna	**GET**	**/api/services/{id}**	Legge un servizio per l'identificativo	**GetServiceById**
API privata	**POST**	**/api/services/{id}**	Modifica i metadati di un servizio per l'identificativo	**UpdateServiceById**
API privata	**DELETE**	**/api/services/{id}**	Elimina un servizio per l'identificativo	**DeleteServiceById**
API privata	**PUT**	**/api/services**	Crea i metadati di un servizio	**CreateService**

Tabella 4.4: API del microservizio Services Microservice.

4.4.1 GetServices Function

La funzione *GetServices* è un servizio serverless senza stato che recupera la lista dei servizi dal database *Services Database*. Si prevede un traffico ridotto verso questa funzione, dal momento che la funzione sarà utilizzata saltuariamente, sarà necessario scalare a zero le repliche dopo un periodo di inutilizzo. La funzione può essere distribuita in due repliche e in caso di inattività per cinque minuti, può essere scalata a zero repliche.

4.4.2 GetServiceById Function

La funzione *GetServiceById* è un servizio serverless che, tramite l'identificativo specificato, recupera i dati di un particolare servizio dal database *Services Database*. Si prevede un traffico ridotto. Il servizio è privo di stato. Anche se la funzione verrà utilizzata solo occasionalmente, sarà importante mantenerne attiva almeno una replica per ridurre i tempi di avvio a freddo. Non sarà necessario scalare la funzione a più di due repliche.

4.4.3 UpdateServiceById Function

La funzione *UpdateServiceById* è un servizio serverless che consente di modificare i dati di un particolare servizio nel database *Services Database* tramite l'identificativo specificato. La funzione verrà utilizzata solo occasionalmente e pertanto si prevede un traffico ridotto verso questo servizio. La funzione dovrà essere scalata a due repliche per garantire una sufficiente disponibilità. Sarà importante mantenere attiva almeno una replica per ridurre i tempi di avvio a freddo, anche se la funzione verrà utilizzata solo saltuariamente.

4.4.4 DeleteServiceById Function

La funzione *DeleteServiceById* è un servizio serverless che permette la cancellazione dei dati relativi a un servizio specifico nel database *Services Database*, identificato mediante un identificativo specifico. La funzionalità di cancellazione di un servizio verrà eseguita con successo solo se non c'è nessun exploit collegato al servizio corrente. Si prevede un traffico ridotto verso di essa. Per garantire una disponibilità sufficiente, la funzione dovrà essere scalata a due repliche, ma non è prevista la necessità di scalare a un numero superiore.

4.4.5 CreateService Function

La funzione *CreateService* è un servizio serverless che consente la creazione di nuovi dati relativi a un servizio nel database *Services Database*. La funzione può essere scalata a due repliche per garantire una disponibilità sufficiente, ma non è necessario mantenere sempre attiva almeno una replica, in quanto il numero di richieste sarà limitato. Inoltre, non è prevista la necessità di scalare a un numero superiore di repliche.

Eventi

UpdateServiceById

La funzione UpdateServiceById produrrà un evento di tipo *ServiceUpdatedEvent* quando avviene la modifica di un servizio, avviando l'esecuzione dell'aggiornamento della configurazione dei job nel microservizio Jobs Microservice (vedi Cap. 4.6).

DeleteServiceById

La funzione DeleteServiceById produrrà un evento di tipo *ServiceDeletedEvent* quando avviene la cancellazione di un servizio, anche qui partirà l'esecuzione dell'aggiornamento della configurazione dei job nel microservizio Jobs Microservice (vedi Cap. 4.6).

CreateService

La funzione CreateService produrrà un evento di tipo *ServiceCreatedEvent* quando si crea un servizio, con conseguente aggiornamento della configurazione dei job nel microservizio Jobs Microservice (vedi Cap. 4.6).

Modellazione dei dati del database

Per il salvataggio dei metadati dei servizi dei team da attaccare verrà utilizzato il database *Services Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Leggere la lista dei servizi.
- Leggere un servizio in base al suo id.
- Modificare un servizio in base al suo id.
- Creare un servizio.
- Cancellare un servizio.

Le operazioni di lettura sono semplici e basate su un interrogazione per chiave o sulla lettura completa del database. È possibile utilizzare il database TiKV chiave-valore per salvare i dati dei servizi. La chiave delle righe del database corrisponderà all'identificativo dei servizi, mentre il valore conterrà i suoi dati.

Ecco un esempio di come i dati potranno essere storicizzati nel database:

```
{
  "7c8c2e7c-f830-4efa-b425-bde63b878d5f":
    {"name":"crypto-maze","ports":{"web":8080,"db":3306}},

  "b8f52109-b829-4c7d-93c0-de87f7fe564d":
    {"name":"pwnerandum","ports":{"netcat":4223}}
}
```

Listing 4.3: Esempio dati: Services Database

Nota bene: Nell'esempio appena mostrato i dati sono inseriti come oggetti per facilitarne la lettura; nella pratica i dati dei servizi andranno storicizzati come stringhe, non come oggetti, in quanto TiKV non supporta il salvataggio di oggetti.

4.5 Exploits Microservice

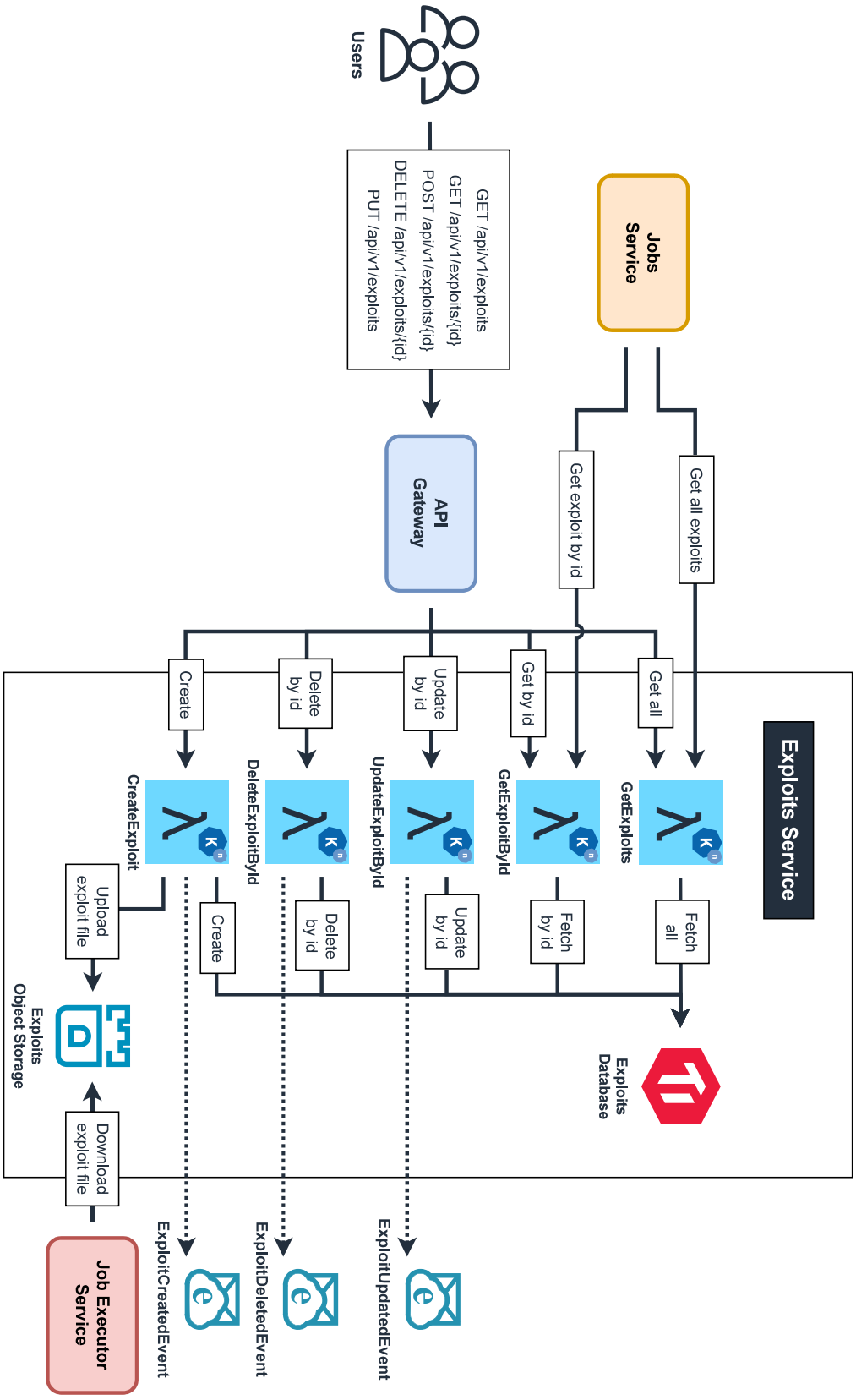

Figura 4.5: Architettura a basso livello del microservizio: Exploits Microsevice

Come descritto nel capitolo 3.2.4, Exploits Microsevice è il microservizio che si occupa di gestire i dati degli exploit dei team competitor che si intendono attaccare. Verrà gestita la creazione, la modifica e l'eliminazione di team. Si studiano le API del servizio, le risorse, la modellazione del database e gli eventi prodotti e consumati.

API del microservizio

Nella seguente tabella sono espresse le API esposte dal microservizio.

Tipo risorsa	Metodo HTTP	Endpoint	Descrizione risorsa	Risorsa
API privata e interna	**GET**	**/api/exploits**	Legge la lista degli exploit	**GetExploits**
API privata e interna	**GET**	**/api/exploits/{id}**	Legge un exploit per l'identificativo	**GetExploitById**
API privata	**POST**	**/api/exploits/{id}**	Modifica i metadati di un exploit per l'identificativo	**UpdateExploitById**
API privata	**DELETE**	**/api/exploits/{id}**	Elimina un exploit per l'identificativo	**DeleteExploitById**
API privata	**PUT**	**/api/exploits**	Crea i metadati di un exploit	**CreateExploit**

Tabella 4.5: API del microservizio Exploits Microservice.

4.5.1 GetExploits Function

La funzione *GetExploits* è un servizio serverless senza stato che recupera la lista degli exploit dal database *Exploits Database*. Si prevede un traffico ridotto verso questa funzione. Poiché la funzione sarà utilizzata saltuariamente, sarà necessario scalare a zero le repliche dopo un periodo di inutilizzo. La funzione può essere distribuita in due repliche e in caso di inattività per cinque minuti, può essere scalata a zero repliche.

4.5.2 GetExploitById Function

La funzione *GetExploitById* è un servizio serverless che, tramite l'identificativo specificato, recupera i dati di un particolare exploit dal database *Exploits Database*. Si prevede un traffico ridotto. L' exploit è privo di stato. Anche se la funzione verrà utilizzata solo occasionalmente, sarà importante mantenere attiva almeno una replica per ridurre i tempi di avvio a freddo. Non sarà necessario scalare la funzione a più di due repliche.

4.5.3 UpdateExploitById Function

La funzione *UpdateExploitById* è un servizio serverless che consente di modificare i dati di un particolare exploit nel database *Exploits Database* tramite l'identificativo specificato. La funzione verrà utilizzata solo occasionalmente e pertanto si prevede un traffico ridotto verso questo exploit. La funzione dovrà essere scalata a due repliche per garantire una sufficiente disponibilità. Sarà importante mantenere attiva almeno una replica per ridurre i tempi di avvio a freddo, anche se la funzione verrà utilizzata solo saltuariamente.

4.5.4 DeleteExploitById Function

La funzione "*DeleteExploitById* è un servizio serverless che permette la cancellazione dei dati relativi a un exploit specifico nel database "*Exploits Database*, identificato mediante un identificativo specifico. La cancellazione di un exploit prevede la cancellazione dell'exploit dal database, ma non dallo storage condiviso. La cancellazione del file di exploit può provocare il crash dei servizi che eseguono il programma.

Si prevede un traffico ridotto verso di essa. Per garantire una disponibilità sufficiente, la funzione dovrà essere scalata a due repliche, ma non è prevista la necessità di scalare a un numero superiore.

4.5.5 CreateExploit Function

La funzione *CreateExploit* è un servizio serverless che consente la creazione di nuovi dati relativi a un exploit nel database *Exploits* Database. Tale funzione si occupa, inoltre, del caricamento del file di exploit in uno storage condiviso ad oggetti, identificato nell'immagine 4.5 come *Exploits Object Storage*. Se il caricamento del file fallisce, la funzione deve restituire un messaggio di errore all'utente. In caso contrario restituirà un messaggio positivo.

La funzione può essere scalata a due repliche per garantire una disponibilità sufficiente, ma non è necessario mantenere sempre attiva almeno una replica, in quanto il numero di richieste sarà limitato. Non è prevista la necessità di scalare a un numero superiore di repliche.

Eventi

UpdateExploitById

La funzione *UpdateExploitById* produrrà un evento di tipo *ExploitUpdatedEvent* quando avviene la modifica di un exploit. Tale evento scatenerà l'esecuzione dell'aggiornamento della configurazione dei job nel microservizio *Jobs Microservice* (vedi Cap. 4.6).

DeleteExploitById

La funzione *DeleteExploitById* produrrà un evento di tipo *ExploitDeletedEvent* quando avviene la cancellazione di un exploit, avviando l'esecuzione dell'aggiornamento della configurazione dei job nel microservizio *Jobs Microservice* (vedi Cap. 4.6).

CreateExploit

La funzione *CreateExploit* produrrà un evento di tipo *ExploitCreatedEvent* quando si crea un exploit. Tale evento scatenerà l'esecuzione dell'aggiornamento della configurazione dei job nel microservizio *Jobs Microservice* (vedi Cap. 4.6).

Modellazione dei dati del database

Per il salvataggio dei metadati degli exploit dei team da attaccare verrà utilizzato il database *Exploits Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Leggere la lista degli exploit.
- Leggere un exploit in base al suo id.
- Modificare un exploit in base al suo id.
- Creare un exploit.
- Cancellare un exploit.

Le operazioni di lettura sono semplici e basate su un interrogazione per chiave o sulla lettura completa del database. È possibile utilizzare il database TiKV chiave-valore per salvare i dati degli exploit. La chiave delle righe del database corrisponderà all'identificativo degli exploit, mentre il valore conterrà i suoi dati.

Verranno salvate le informazioni su come lanciare l'exploit nel campo *command*, le dipendenze che saranno necessarie per lanciare il programma in *dependencies*, lo stato in *isEnabled*, il servizio a cui è riferito in *serviceId*, la versione in *version* e il nome in *name*.

Ecco un esempio di come i dati potranno essere storicizzati nel database:

```
{
  "562489d7-deb8-4939-99b8-1d6a799f92cf":
    {
      "name":"exploit-1",
      "version":"v1.0",
      "command":"python3 exploit.py",
      "dependencies":["requests","pwn"],
      "isEnabled":true,
      "serviceId":"b8f52109-b829-4c7d-93c0-de87f7fe564d"
    },

  "e5c2d766-eca2-4ea5-8ef1-0a07a0f25194":
    {
      "name":"exploit-2",
      "version":"v0.1-beta",
      "command":"node exploit.js",
      "dependencies":[],
      "isEnabled":false,
      "serviceId":"b8f52109-b829-4c7d-93c0-de87f7fe564d"
    },

  "182c7cf0-e931-4063-a59b-9e7c66fdb8b5":
    {
      "name":"exploit-generic",
```

```
25        "version":"v0.8",
26        "command":"node exploit.js",
27        "dependencies":[],
28        "isEnabled":true,
29        "serviceId": null
30      }
31  }
```

Listing 4.4: Esempio dati: Exploits Database

Nell'esempio di storicizzazione sono salvati 3 exploit. Il primo è un exploit attivo riferito ad un exploit, il secondo è un exploit non attivo riferito allo stesso exploit e l'ultimo è un exploit generico.

Nota bene: Nell'esempio appena mostrato i dati sono inseriti come oggetti per facilitarne la lettura; nella pratica i dati degli exploit andranno storicizzati come stringhe, non come oggetti, in quanto TiKV non supporta il salvataggio di oggetti.

4.6 Jobs Microservice

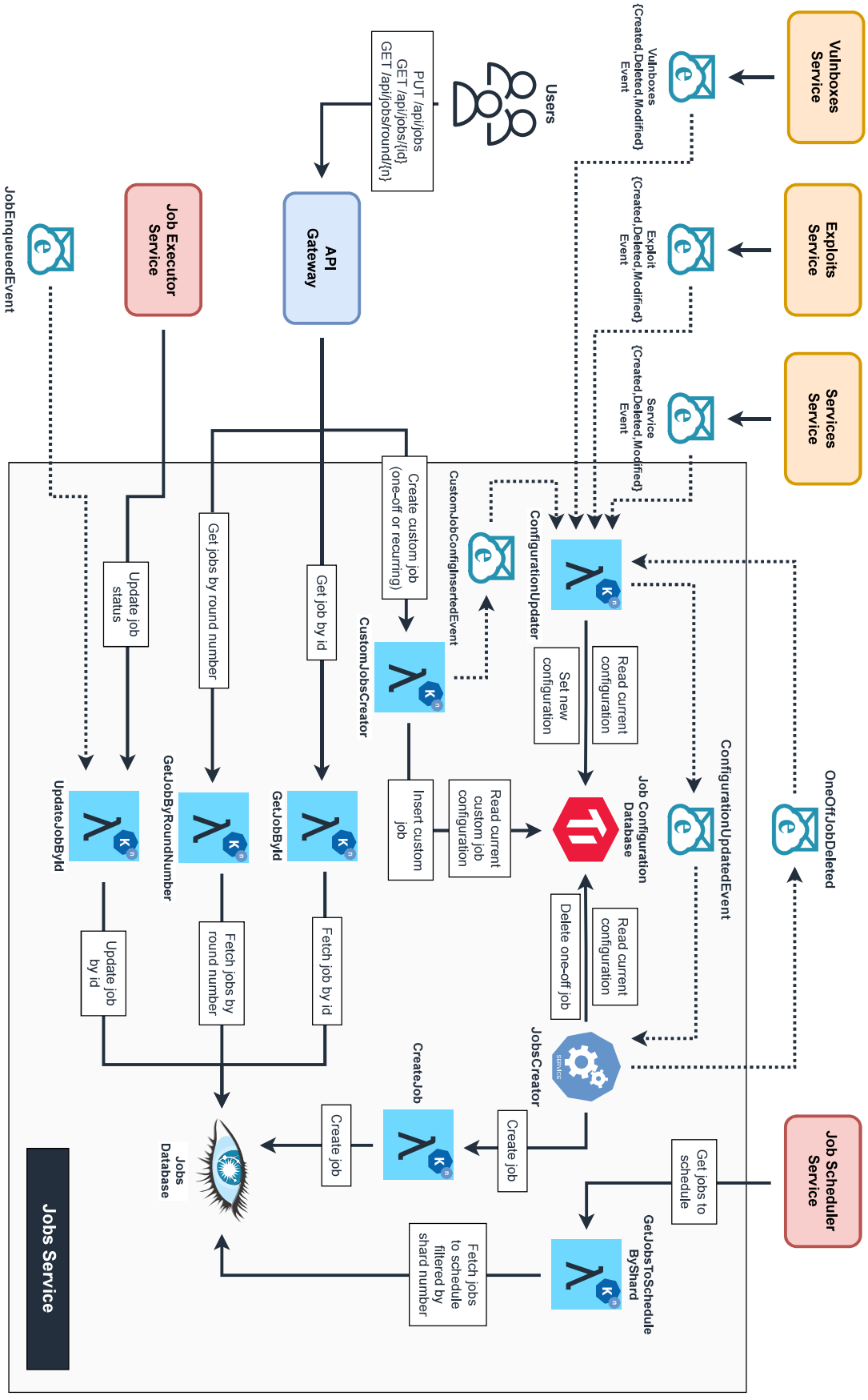

Figura 4.6: Architettura a basso livello del microservizio: Jobs Microsevice

Come descritto nel capitolo 3.2.7, Jobs Microsevice consente agli utenti di creare job personalizzati. Il microservizio fornisce ai suoi utenti l'accesso a delle API per la visualizzazione dello stato dei job, che possono essere filtrati per identificativo, per round e per numero di shard. Queste API permettono anche di creare job personalizzati, che possono essere eseguiti una sola volta o in modo ripetitivo durante la competizione.

Il *Jobs Microservice* comprende molte funzionalità interne per la creazione, la pianificazione e la configurazione dei job, la cui creazione può essere divisa in due categorie: i job programmati automaticamente dal sistema e i job pianificati manualmente dagli utenti. Il sistema automatico adatterà alla configurazione dei job in base allo stato di vulnbox, servizi e exploit creati, modificati o eliminati durante la competizione. Gli utenti possono invece pianificare manualmente i job tramite le API offerte dal microservizio.

L'importanza di studiare le API dei job, le risorse, la modellizzazione del database e gli eventi prodotti e consumati non può essere sottovalutata per comprendere appieno il funzionamento del *Jobs Microservizio*.

API del microservizio

Nella seguente tabella sono espresse le API esposte dal microservizio.

Tipo risorsa	Metodo HTTP	Endpoint	Descrizione risorsa	Risorsa
API privata	**PUT**	**/api/jobs**	Crea un job personalizzato ricorrente o non	**CustomJobsCreator**
API privata	**GET**	**/api/jobs/{id}**	Legge lo stato di un job per identificativo	**GetJobById**
API privata	**GET**	**/api/jobs/round/{n}**	Legge lo stato dei job di un round	**GetJobByRound-Number**
API interna	**POST**	**-**	Aggiorna lo stato di un job per identificativo	**UpdateJobById**

Tabella 4.6: API del microservizio Jobs Microservice.

4.6.1 ConfigurationUpdater Function

La funzione *ConfigurationUpdater* sarà avviata alla ricezione di un evento che modifica lo stato di vulnbox, exploit e servizi, oppure all'inserimento di un job personalizzato. La funzione avrà il compito di aggiornare la configurazione attuale dei job, la quale verrà letta dal servizio *JobsCreator* per creare i job che saranno poi pianificati da un altro microservizio (vedi Cap. 4.7).

La funzione dovrà leggere la configurazione attuale per poterne calcolare una nuova. Verranno salvati i servizi, le vulnbox e i metadati degli exploit per favorire un maggiore disaccoppiamento tra il microservizio de job e quelli delle risorse. Nel capitolo riservato alla gestione degli eventi verrà enunciato come tale funzione dovrà modificare la configurazione in risposta ai vari eventi che consuma.

Dopo il calcolo, la nuova configurazione, verrà archiviata nel database "Job Configuration Database" suddivisa in tre sezioni equilibrate, chiamate shard. Questo garantirà una maggiore affidabilità del servizio che crea i job, poiché sarà possibile continuare a creare la maggior parte dei job anche in caso di guasto di un nodo che ospita un *JobCreator*.

Per fare un esempio, se si dovesse pianificare l'esecuzione di un exploit contro dodici team, la configurazione verrà divisa in tre parti, ognuna contenente quattro job da creare. Il compito di creare i job dalla configurazione specificata è affidato al *JobsCreator*.

4.6.2 JobsCreator Service

Il *JobsCreator* è il servizio incaricato di inserire i job nel database *Jobs Database* a partire dalla configurazione presente nel database *Job Configuration Database*. Per far fronte alla mole di lavoro, esso verrà distribuito in tre repliche, ciascuna delle quali gestirà una sezione specifica dei job della configurazione. Questa divisione permette a ciascuna copia di *JobsCreator* di concentrarsi su una parte specifica del carico di lavoro.

Per creare i job, il servizio, dovrà leggere la configurazione all'avvio e tenerla costantemente aggiornata in caso di modifiche. La creazione dei job viene eseguita tramite l'invocazione della funzione *CreateJob*.

4.6.3 CreateJob Function

CreateJob è la funzione che inserisce i nuovi job che dovranno poi essere schedulati nel database Jobs Database. I job saranno quindi creati con lo stato *WAIT_FOR_QUEUING*, cioè in attesa di essere accodati. Ogni job sarà composto da tutte le informazioni necessarie per essere eseguito correttamente, incluso il servizio di destinazione, la vulnbox bersaglio, i metadati dell'exploit da utilizzare e il numero di round. Dopo aver creato i job non ricorrenti, il servizio dovrà cancellare tali informazioni dalla configurazione per evitare una loro futura pianificazione. La funzione *CreateJob* dovrà leggere e prendere in considerazione eventuali modifiche alla configurazione dei job desiderati.

4.6.4 CustomJobsCreator Function

CustomJobsCreator è la funzione richiamata dall'API Gateway che permette agli utenti di creare nuovi job, ricorrenti o non, che dovranno poi essere creati dal *JobsCreator*. I job non ricorrenti dovranno essere creati una sola volta, mentre quelli ricorrenti andranno creati e pianificati una volta per round. La funzione andrà ad inserire i job nel database di configurazione *Job Configuration Database*. Sarà poi compito della funzione *ConfigurationUpdater* di mantenere aggiornata la configurazione dei job.

4.6.5 GetJobById Function

La funzione *GetJobById* è un servizio serverless che consente di leggere i dati relativi a un particolare lavoro identificato tramite un identificativo specifico, dal database *Jobs Database*. Essendo una funzione senza stato, non mantiene informazioni relative alle sessioni utente. Tuttavia, si prevede una maggiore frequenza di invocazioni verso questa funzione rispetto ad altre e pertanto sarà necessario scalare la funzione per garantirne la massima disponibilità. Per raggiungere questo obiettivo, è probabile che la funzione debba essere scalata a un numero

superiore rispetto alle altre. Questo consente di soddisfare la domanda di richieste e garantire tempi di risposta rapidi.

4.6.6 GetJobByRoundNumber Function

La funzione *GetJobByRoundNumber* è un servizio serverless che consente l'accesso ai dati dei lavori associati ad un determinato round, attraverso la consultazione del database *Jobs Database*. Dato che ci sono più lavori associati ad un singolo round, è prevista una maggiore richiesta di utilizzo di questa funzione. Per soddisfare questa domanda e garantire la massima disponibilità, sarà necessario scalare la funzione. Questo porterà ad una risposta rapida alle richieste degli utenti.

4.6.7 UpdateJobById Function

La funzione *UpdateJobById* è un servizio serverless che consente l'aggiornamento dei dati di un lavoro specifico nel database *Jobs Database*. Viene invocata attraverso un evento e non mantiene informazioni relative alle sessioni utente. A causa della quantità elevata di richieste di aggiornamento prevista (circa 1500 ogni minuto), questa funzione dovrà scalare per garantire performance e disponibilità elevata. Sarà quindi necessario prevedere una capacità di elaborazione adeguata per soddisfare questa domanda, al fine di garantire la massima efficienza e tempestività nell'aggiornamento dei dati dei job.

4.6.8 GetJobsToScheduleByShard Function

GetJobsToScheduleByShard è una funzione serverless che si occupa di elencare la lista dei job da schedulare. Questa funzione si basa sulla selezione dei job in base al loro numero di shard e per lo stato "WAIT_FOR_QUEUING". Essa viene invocata dal microservizio *Job Scheduler Service*. La funzione deve garantire alti livelli di disponibilità, in quanto è cruciale per l'efficienza del sistema di scheduling. La sua architettura senza stato ne garantisce l'elasticità e la scalabilità, permettendo di gestire un elevato volume di richieste senza interruzioni.

Eventi

CustomJobsCreator

La funzione *CustomJobsCreator* produrrà un evento di tipo *CustomJobConfigInsertedEvent* quando avviene l'inserimento di un job personalizzato. Tale evento scatenerà l'esecuzione dell'aggiornamento della configurazione dei job tramite la funzione *ConfigurationUpdater*.

ConfigurationUpdater

La funzione *ConfigurationUpdater* si occuperà di aggiornare la configurazione quando è invocata da un evento che mira a modificare la configurazione. Gli eventi che riceverà conterranno l'intero stato della risorsa modificata, creata o eliminata. La funzione leggerà la configurazione attuale dei job, modificandola in conseguenza all'evento ricevuto.

Ad esempio, se riceverà l'evento *VulnboxesCreatedEvent*, dovrà preoccuparsi di salvare la nuova vulnbox nel database di configurazione, ricalcolare lo stato desiderato dei job per i

round e storicizzarlo. In questo caso sarà necessario schedulare l'esecuzione di ogni exploit anche contro la nuova vulnbox.

La funzione *ConfigurationUpdater* produrrà un evento di tipo *ConfigurationUpdatedEvent* quando avviene la modifica della configurazione desiderata dei job, con conseguente aggiornamento della configurazione dei job nel servizio *JobsCreator*.

JobsCreator

Il servizio *JobsCreator* alla ricezione dell'evento *ConfigurationUpdatedEvent* leggerà la nuova configurazione dei job che andrà a creare per ogni round della competizione, fino a quando non riceverà una nuova richiesta di aggiornamento. Il servizio dopo aver creato un job di tipo non ricorrente dovrà preoccuparsi di eliminarlo da *Job Configuration Database*, per evitare di crearlo di nuovo.

Modellazione dei dati del database

Job Configuration Database

Per il salvataggio della configurazione desiderata dei job verrà utilizzato il database *Job Configuration Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Leggere la configurazione attuale.
- Aggiornare la configurazione attuale.
- Creare un job personalizzato ricorrente o non.
- Cancellare un job non ricorrente.

Ecco un esempio di come i dati potranno essere storicizzati nel database:

```
{
  "services": {
    "7c8c2e7c-f830-4efa-b425-bde63b878d5f": {
      "name": "crypto-maze",
      "ports": {
        "web": 8080,
        "db": 3306
      }
    },
    "b8f52109-b829-4c7d-93c0-de87f7fe564d": {
      "name": "pwnerandum",
      "ports": {
        "netcat": 4223
      }
    }
  },
  "vulnboxes": {
    "eb0bd9db-a9bf-453d-be88-424ec782fac0": {
      "name": "polito",
      "ip": "10.10.1.1"
    },
    "cbc2cdb0-294d-4d83-aabb-8c697587e2e7": {
      "name": "unicam",
```

```
      "ip": "10.20.1.1"
    },
    "3dac9993-7b9f-40c1-ace7-cccc9842c416": {
      "name": "unimi",
      "ip": "10.30.1.1"
    }
  },
  "exploits": {
    "562489d7-deb8-4939-99b8-1d6a799f92cf": {
      "name": "exploit-1",
      "version": "v1.0",
      "isEnabled": true,
      "serviceId": "b8f52109-b829-4c7d-93c0-de87f7fe564d"
    },
    "e5c2d766-eca2-4ea5-8ef1-0a07a0f25194": {
      "name": "exploit-2",
      "version": "v0.1-beta",
      "isEnabled": false,
      "serviceId": "b8f52109-b829-4c7d-93c0-de87f7fe564d"
    },
    "182c7cf0-e931-4063-a59b-9e7c66fdb8b5": {
      "name": "exploit-generic",
      "version": "v0.1",
      "isEnabled": true,
      "serviceId": null
    }
  },
  "one-off-jobs": {
    // NON RICORRENTE - GENERICO
    "8253efc8-5e8f-44d2-b096-80112e1409ea": {
      "exploitId": "182c7cf0-e931-4063-a59b-9e7c66fdb8b5",
      "type": "generic"
    },
    // NON RICORRENTE - SINGOLA VULNBOX
    "86745cd3-b16b-461e-8f03-123ef1d829c3": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "eb0bd9db-a9bf-453d-be88-424ec782fac0"
    },
    // NON RICORRENTE - OGNI VULNBOX
    "f611129b-1c6b-44c9-b052-76d5c46a7dba": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "type": "foreach-vulnbox"
    }
  },
  "recurring-jobs": {
    // RICORRENTE - GENERICO
    "7698c81f-b64a-4dfb-858d-4187e056d3a1": {
      "exploitId": "182c7cf0-e931-4063-a59b-9e7c66fdb8b5",
      "type": "generic"
    },
    // RICORRENTE - SINGOLA VULNBOX
    "4da7edf5-32c2-48db-9c14-2f2c6544a9cf": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "3dac9993-7b9f-40c1-ace7-cccc9842c416"
    },
    // RICORRENTE - OGNI VULNBOX
    "0484ac60-3831-42f8-9c7c-072b41abe2a4": {
```

```
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "type": "foreach-vulnbox"
    }
  },
  // Job da pianificare solo per il prossimo round per lo shard 1
  "shard-1-one-off-jobs": {
    "efc67ef8-33e7-4bdc-8157-b431d61d28db": {
      "exploitId": "182c7cf0-e931-4063-a59b-9e7c66fdb8b5"
    },
    "fd39c020-3ac9-4c65-a0b9-446c4231ef99": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "eb0bd9db-a9bf-453d-be88-424ec782fac0"
    }
  },
  // Job da pianificare per i prossimi round per lo shard 1
  "shard-1-jobs": {
    // PIANIFICATI IN AUTOMATICO
    "cfbb4d67-b555-4081-b0c9-380f009bca8a": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "eb0bd9db-a9bf-453d-be88-424ec782fac0"
    },
    "985d90f6-68b9-4c3e-a51a-f8d361198a38": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "cbc2cdb0-294d-4d83-aabb-8c697587e2e7"
    },
    "2fc7cbe7-9c21-4091-91f4-780308507e5b": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "3dac9993-7b9f-40c1-ace7-cccc9842c416"
    }
  },
  // Job da pianificare solo per il prossimo round per lo shard 2
  "shard-2-one-off-jobs": {
  "2ea4af1b-617b-412b-a9fd-8574ea2f0cd6": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "eb0bd9db-a9bf-453d-be88-424ec782fac0"
    },
    "2a31fa80-d306-4cb8-a043-37221bc9862c": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "cbc2cdb0-294d-4d83-aabb-8c697587e2e7"
    },  },
  // Job da pianificare per i prossimi round per lo shard 2
  "shard-2-jobs": {
    // RICORRENTE - GENERICO
    "9afe7aa2-5192-4161-a30e-53345c9b9853": {
      "exploitId": "182c7cf0-e931-4063-a59b-9e7c66fdb8b5"
    },
  // RICORRENTE - SINGOLA VULNBOX
    "4ed8a640-d1ca-4d7c-9263-2730d472c6f4": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "3dac9993-7b9f-40c1-ace7-cccc9842c416"
    }
  },
  // Job da pianificare solo per il prossimo round per lo shard 3
  "shard-3-one-off-jobs": {
    "57409f93-6d55-49c5-b53b-6a7978044c40": {
      "exploitId": "562489d7-deb8-4939-99b8-1d6a799f92cf",
      "vulnboxId": "3dac9993-7b9f-40c1-ace7-cccc9842c416"
```

```
    }
  },
  // Job da pianificare per i prossimi round per lo shard 3
  "shard-3-jobs": {
    // RICORRENTE - OGNI VULNBOX
    "0f4aac47-58f5-4151-8fd0-444361103f50": {
      "exploitId": "e5c2d766-eca2-4ea5-8ef1-0a07a0f25194",
      "vulnboxId": "eb0bd9db-a9bf-453d-be88-424ec782fac0"
    },
    "8402f091-b56e-4353-9cbe-0af42a17be2a": {
      "exploitId": "e5c2d766-eca2-4ea5-8ef1-0a07a0f25194",
      "vulnboxId": "cbc2cdb0-294d-4d83-aabb-8c697587e2e7"
    },
    "628bc038-ba51-45bf-abf4-0fe4ee2cd48a": {
      "exploitId": "e5c2d766-eca2-4ea5-8ef1-0a07a0f25194",
      "vulnboxId": "3dac9993-7b9f-40c1-ace7-cccc9842c416"
    }
  }
}
```

Listing 4.5: Esempio dati: Jobs Database

L'esempio cerca di descrivere come la configurazione dei job è modificata per facilitarne la lettura dal servizio *JobsCreator*.

Di seguito una breve descrizione dei valori inseriti nell'esempio di cui sopra:

- **services**: la lista dei servizi.
- **vulnboxes**: la lista delle vulnbox.
- **exploits**: la lista degli exploit.
- **one-off-jobs**: la lista dei job non ricorrenti.
- **recurring-jobs**: la lista dei job ricorrenti.
- **shard-1-one-off-jobs**: i job non ricorrenti che devono essere pianificati dallo shard 1.
- **shard-1-jobs**: i job ricorrenti che devono essere pianificati dallo shard 1.
- **shard-2-one-off-jobs**: i job non ricorrenti che devono essere pianificati dallo shard 2.
- **shard-2-jobs**: i job ricorrenti che devono essere pianificati dallo shard 2.
- **shard-3-one-off-jobs**: i job non ricorrenti che devono essere pianificati dallo shard 3.
- **shard-3-jobs**: i job ricorrenti che devono essere pianificati dallo shard 3.

Jobs Database

Per il salvataggio dei metadati dei job che saranno creati, pianificati ed eseguiti in ogni turno, verrà utilizzato il database *Jobs Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Leggere lo stato dei job di un round.

- Leggere i job da schedulare filtrati per numero di shard.
- Leggere un job in base al suo id, numero di round e numero shard.
- Modificare un job in base al suo id, numero di round e numero shard.
- Creare un job personalizzato.
- Creare un job.

Nell'analisi dei volumi si è evidenziata la necessità di utilizzare un database capace di sostenere grandi volumi di traffico. Viene utilizzato, quindi, Cassandra per salvare tali dati, dopo averli modellati correttamente. Sarà necessario modellare i dati per favorire il filtro dei dati per round e per shard. Successivamente i dati andranno poi filtrati per stato.

Bisogna quindi inserire come chiave di partizione le colonne *shard_number* e *round_number*, le quali formeranno insieme a *job_id* la chiave primaria del job. Per favorire il filtro per stato è opportuno creare un indice sulla colonna *job_state*.

Ecco la modellazione dei dati nel database *Jobs Database* in Cassandra:

```
-- KEYSPACE:
CREATE KEYSPACE svc WITH replication =
  {'class': 'SimpleStrategy', 'replication_factor' : 3};

-- TABLE
CREATE TABLE svc.jobs (
  shard_number smallint,
  round_number smallint,
  job_id text,
  job_state text,
  job_data text,
  PRIMARY KEY ((shard_number, round_number), job_id)
);

-- INDEX on job_state
CREATE INDEX job_state ON svc.jobs (job_state);
```

Figura 4.7: CQL per creare il Jobs Database

Di seguito è mostrato un inserimento di alcuni job nel database:

```
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (1,1,'e28df730-0cc4-43ce-9501-6234e2bbfabc','QUEUED','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (2,1,'6c530ae5-2816-424f-bc3b-86feec42fed9','QUEUED','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (3,1,'47c630e6-5ee8-4ddc-9d4f-a4a4e5f41124','QUEUED','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (1,1,'dcd637ca-2b32-476a-9a28-0fae49e41d9f','QUEUED','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (2,1,'302fb826-4ebb-4086-a3e2-2229da8b5a24','QUEUED','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (1,1,'85339625-a2b3-4a6f-80ff-94912fd1d3ed','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (2,1,'2748978d-1f35-4d24-b51d-e096da277caf','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (3,1,'ab5a0c92-adad-4b34-b41f-7244b8869a6d','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (1,1,'1ce0dcd4-6738-4e42-8f6a-8f359521c5ad','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (2,1,'987ce62d-eea8-4fcf-b708-fc443161fa45','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (3,1,'e129f398-ae1b-4e45-8158-ddf2267e5a17','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (1,1,'703b8053-92ea-4015-85a0-55a83bff0e41','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (2,1,'edbcf1ab-d7c4-4fbd-acf8-fc6d332aea1f','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (3,1,'ef0ba5b0-8282-4b3f-942e-2acb3240139a','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (1,1,'0a2fa8e4-623c-4d7d-9515-6c1bf6bb8318','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (2,1,'1ccf4364-7dca-4271-95f4-c9133dfecae7','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (3,1,'8748a9ad-e3f6-4352-86f5-85c5ba3d2fad','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (1,1,'95f80ae9-f238-4683-adba-e2fdb81a9259','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (2,1,'71cd6290-40df-4ea4-aed0-cd4f920b4cf2','WAIT_FOR_QUEUING','{...}');
INSERT INTO svc.jobs(shard_number,round_number,job_id,job_state,job_data) VALUES (3,1,'dc5b4181-7d42-4dec-af1b-6a002cf7cb1e','WAIT_FOR_QUEUING','{...}');
```

Figura 4.8: CQL per creare dei job nel Jobs Database

Nella seguente immagine è mostrato un esempio di lettura dei job per numero di round, per shard e filtrati per stato:

```
cqlsh> SELECT * FROM svc.jobs WHERE shard_number = 1 AND round_number = 1;

 shard_number | round_number | job_id                               | job_data | job_state
--------------+--------------+--------------------------------------+----------+------------------
            1 |            1 | 0a2fa8e4-623c-4d7d-9515-6c1bf6bb8318 |    {...} | WAIT_FOR_QUEUING
            1 |            1 | 1ce0dcd4-6738-4e42-8f6a-8f359521c5ad |    {...} | WAIT_FOR_QUEUING
            1 |            1 | 703b8053-92ea-4015-85a0-55a83bff0e41 |    {...} | WAIT_FOR_QUEUING
            1 |            1 | 85339625-a2b3-4a6f-80ff-94912fd1d3ed |    {...} | WAIT_FOR_QUEUING
            1 |            1 | 95f80ae9-f238-4683-adba-e2fdb81a9259 |    {...} | WAIT_FOR_QUEUING
            1 |            1 | dcd637ca-2b32-476a-9a28-0fae49e41d9f |    {...} |           QUEUED
            1 |            1 | e28df730-0cc4-43ce-9501-6234e2bbfabc |    {...} |           QUEUED

(7 rows)
cqlsh> SELECT * FROM svc.jobs WHERE shard_number = 1 AND round_number = 1 AND job_state = 'WAIT_FOR_QUEUING';

 shard_number | round_number | job_id                               | job_data | job_state
--------------+--------------+--------------------------------------+----------+------------------
            1 |            1 | 0a2fa8c4 623c 4d7d 9515 6c1bf6bb8318 |    {...} | WAIT_FOR_QUEUING
            1 |            1 | 1ce0dcd4-6738-4e42-8f6a-8f359521c5ad |    {...} | WAIT_FOR_QUEUING
            1 |            1 | 703b8053-92ea-4015-85a0-55a83bff0e41 |    {...} | WAIT_FOR_QUEUING
            1 |            1 | 85339625-a2b3-4a6f-80ff-94912fd1d3ed |    {...} | WAIT_FOR_QUEUING
            1 |            1 | 95f80ae9-f238-4683-adba-e2fdb81a9259 |    {...} | WAIT_FOR_QUEUING

(5 rows)
cqlsh>
```

Figura 4.9: CQL per filtrare i job per numero di round e per shard su Jobs Database

4.7 Job Scheduler Microservice

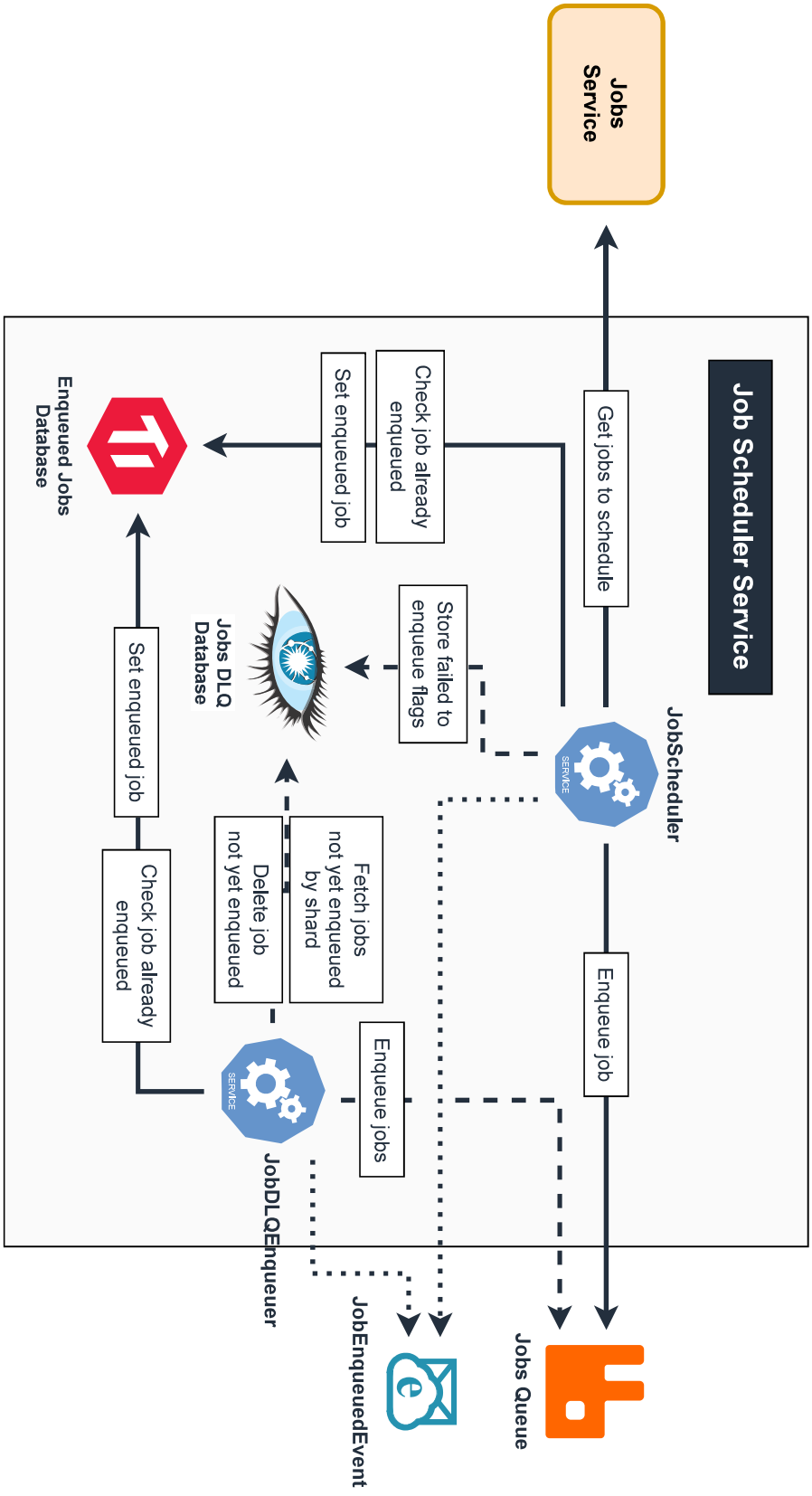

Figura 4.10: Architettura a basso livello del microservizio: Job Scheduler Microsevice

Come descritto nel capitolo 3.2.8, *Job Scheduler Microsevice* è il microservizio che si occupa di mettere i coda per la pianificazione i job che sono in attesa di essere creati. Ora si studiano le API del exploit, le risorse, la modellazione del database e gli eventi prodotti e consumati.

API del microservizio

Il servizio *Job Scheduler Microsevice* non espone API interne, esterne o pubbliche.

4.7.1 JobScheduler Service

Il *JobScheduler* è un servizio serverless che si occupa di accodare i job che devono essere eseguiti nel round corrente. Il servizio legge la lista dei job che deve accodare dal *Jobs Microservice* filtrandoli per numero di round e shard.

Per aumentare la disponibilità e la resilienza del servizio, i dati dei job sono stati partizionati in tre parti; ogni shard è gestito da una replica del *JobScheduler*. Il servizio, quindi, recupera dal *Jobs Microservice* la lista dei job appartenenti al suo shard del round corrente periodicamente. Una volta recuperata la lista dei job verranno accodati nella coda *Jobs Queue* e verrà salvato l'identificativo del job messo in coda in *Enqueued Jobs Database*, un database chiave-valore.

Nel caso in cui il servizio non riesca a mettere in coda i job, li inserirà nel database *Jobs DLQ Database* che funge da Dead Letter Queue (DLQ[2]). Il servizio *JobDLQEnqueuer* si occuperà dell'invio asincrono dei job alla coda *Jobs Queue*. Una volta che il servizio ha accodato il job con successo, invierà un evento *JobEnqueuedEvent* per notificare che il job è stato accodato con successo.

4.7.2 JobDLQEnqueuer Service

Il servizio *JobDLQEnqueuer* è responsabile di recuperare i job non elaborati dal database Jobs DLQ Database. I job vengono inviati asincronamente alla coda *JobsQueue* per essere elaborati in seguito.

Una volta che il servizio ha accodato con successo un job, invia un evento *JobEnqueuedEvent* per notificare che il job è stato inserito nella coda e può essere elaborato. Questo evento fornisce informazioni importanti ai componenti interessati sul processo di elaborazione del job. Il servizio in questione gestisce una partizione dei job nella stessa maniera del *JobScheduler*, cioè legge i job di cui è responsabile filtrandoli per numero di shard. Il servizio *JobDLQEnqueuer* svolge un ruolo cruciale nel garantire la continuità del processo di elaborazione dei job.

Eventi

JobScheduler

Il servizio *UpdateExploitById* produrrà un evento di tipo *JobEnqueuedEvent* quando avviene con successo l'accodamento di un job nella coda *Jobs Queue*. Tale evento scatenerà l'esecuzione della modifica dello stato del job nel microservizio Jobs Microservice (vedi Cap. 4.6).

[2]La Dead Letter Queue, o coda di messaggi non recapitabili, consente di mettere da parte e isolare i messaggi che non possono essere elaborati correttamente o accodati con successo.

JobDLQEnqueuer

Il servizio *JobDLQEnqueuer* produrrà un evento di tipo *JobEnqueuedEvent* quando avviene con successo l'accodamento di un job nella coda *Jobs Queue*, da cui si avvierà la modifica dello stato del job nel microservizio Jobs Microservice (vedi Cap. 4.6).

Modellazione dei dati del database

Enqueued Jobs Database

Per il salvataggio dei job già messi in coda viene utilizzato il database *Enqueued Jobs Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Inserimento dell'identificativo di un job.
- Lettura dell'identificativo di un job.

Le operazioni di lettura sono semplici e basate su un interrogazione diretta alla chiave. È possibile utilizzare il database TiKV chiave-valore per salvare i dati di interesse.

Ecco un esempio di come i dati potranno essere storicizzati nel database:

```
{
  "cfbb4d67-b555-4081-b0c9-380f009bca8a": true,
  "985d90f6-68b9-4c3e-a51a-f8d361198a38": true,
  "2fc7cbe7-9c21-4091-91f4-780308507e5b": true
}
```

Listing 4.6: Esempio dati: Enqueued Jobs Database

Nell'esempio sono stati salvati tre identificativi di job accodati.

Jobs DLQ Database

Per il salvataggio dei metadati dei job che dovranno essere pianificati in maniera asincrona, se si verifica un errore nell'accodamento dal *JobScheduler*, verrà utilizzato il database *Jobs Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Leggere la lista dei job nella DLQ filtrati per shard.
- Cancellare un job filtrandolo per shard e identificativo.

Bisogna quindi inserire come chiave di partizione la colonna *shard_number*, la quale formerà insieme a *job_id* la chiave primaria di ogni job.

Ecco la modellazione dei dati nel database *Jobs DLQ Database* in Cassandra:

```
-- TABLE
CREATE TABLE svc.jobs_dlq (
    shard_number smallint,
    job_id text,
    job_data text,
    PRIMARY KEY ((shard_number), job_id)
);
```

Figura 4.11: CQL per creare il Jobs DLQ Database

Nella seguente immagine è mostrato un esempio di lettura dei job per shard, viene cancellato un job e infine viene letta nuovamente la lista dei job per shard:

```
cqlsh> SELECT * FROM svc.jobs_dlq;

 shard_number | job_id                               | job_data
--------------+--------------------------------------+----------
            3 | 8748a9ad-e3f6-4352-86f5-85c5ba3d2fad |    {...}
            3 | dc5b4181-7d42-4dec-af1b-6a002cf7cb1e |    {...}
            3 | ef0ba5b0-8282-4b3f-942e-2acb3240139a |    {...}
            2 | 1ccf4364-7dca-4271-95f4-c9133dfecae7 |    {...}
            2 | 71cd6290-40df-4ea4-aed0-cd4f920b4cf2 |    {...}
            1 | 0a2fa8e4-623c-4d7d-9515-6c1bf6bb8318 |    {...}
            1 | 95f80ae9-f238-4683-adba-e2fdb81a9259 |    {...}

(7 rows)
cqlsh>
cqlsh> SELECT * FROM svc.jobs_dlq WHERE shard_number = 3;

 shard_number | job_id                               | job_data
--------------+--------------------------------------+----------
            3 | 8748a9ad-e3f6-4352-86f5-85c5ba3d2fad |    {...}
            3 | dc5b4181-7d42-4dec-af1b-6a002cf7cb1e |    {...}
            3 | ef0ba5b0-8282-4b3f-942e-2acb3240139a |    {...}

(3 rows)
cqlsh> DELETE FROM svc.jobs_dlq WHERE shard_number = 3 AND
   ...     job_id = 'dc5b4181-7d42-4dec-af1b-6a002cf7cb1e';
cqlsh>
cqlsh> SELECT * FROM svc.jobs_dlq WHERE shard_number = 3;

 shard_number | job_id                               | job_data
--------------+--------------------------------------+----------
            3 | 8748a9ad-e3f6-4352-86f5-85c5ba3d2fad |    {...}
            3 | ef0ba5b0-8282-4b3f-942e-2acb3240139a |    {...}

(2 rows)
cqlsh>
```

Figura 4.12: CQL leggere i job per shard e cancellarne uno su Jobs DLQ Database

4.8 Job Executor Microservice

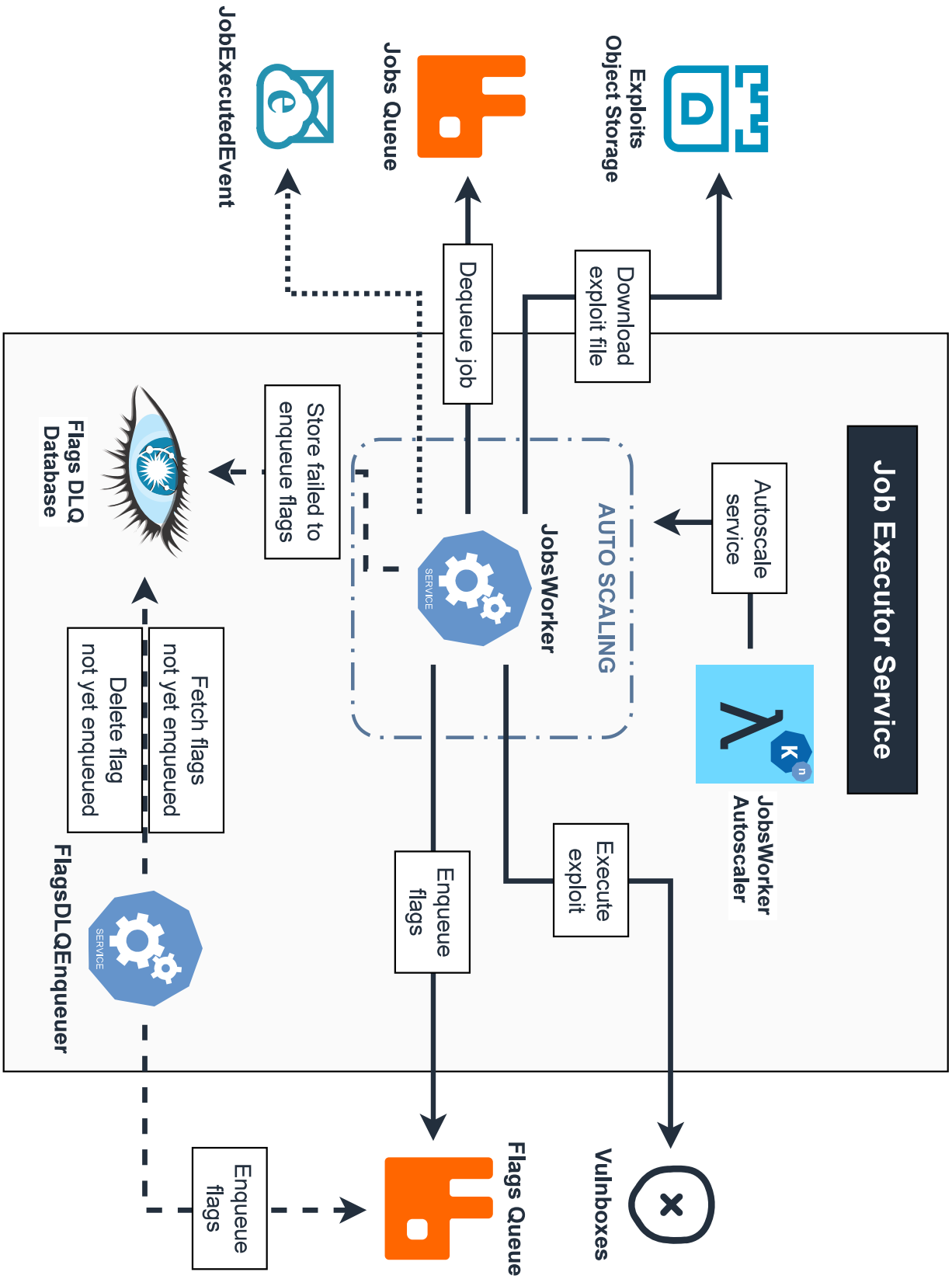

Figura 4.13: Architettura a basso livello del microservizio: Job Executor Microsevice

Come descritto nel capitolo 3.2.9, *Job Executor Service* è il microservizio che si occupa di eseguire i task messi in coda dal *Job Scheduler service* e di recuperare i flag sottratti dall'esecuzione di ogni task. I flag sottratti andranno, poi, inviati al servizio *Flags Service* che si occuperà dell'invio al server di gioco. Il microservizio *Job Executor*, dopo aver eseguito un exploit, notificherà lo stato di esecuzione al *Jobs Service*, il quale aggiornerà lo stato del task.

API del microservizio

Il servizio *Job Executor Microsevice* non espone API interne, esterne o pubbliche.

4.8.1 JobExecutor Service

Il *JobsWorker* del servizio *Job Executor Microservice* è un componente fondamentale per l'esecuzione dei job. Questo servizio serverless si occupa di consumare una lista di job da una coda e scaricare il file di exploit specificato nel job dallo storage condiviso ad oggetti *Exploits Object Storage*.

Il servizio dovrà essere un'immagine container personalizzata, contenente già i motori principali per eseguire i vari exploit; principalmente sarà necessario eseguire codice Python3 e Javascript. Il servizio, nel caso in cui non abbia preinstallata una dipendenza, dovrà scaricarla e installarla autonomamente. Le dipendenze necessarie all'exploit di essere eseguito sono specificate nei metadati dell'exploit stesso.

Il servizio dovrà inviare un messaggio di acknowledgement a RabbitMQ quando avrà eseguito con successo un job. In caso di fallimento, il job che è stato letto dal servizio che si è fermato non è riuscito ad essere eseguito. RabbitMQ, dopo un tempo di timeout configurabile, non ricevendo il messaggio di acknowledgement del job in questione, si occuperà di rimettere in coda tale job. In questo modo è possibile garantire l'esecuzione di ogni job almeno una volta.

Durante l'esecuzione dei job, il *JobsWorker* estrae i flag e li mette in coda per il loro successivo invio da parte del jobs microservice. In caso di fallimento nell'inserimento dei flag in coda, il *JobsWorker* salva i flag in un database, denominato *Flags DLQ Database*, che funge da dead letter queue per un accodamento asincrono, effettuato da un altro servizio. Questo componente garantisce che ogni job venga eseguito con successo e che i flag vengano raccolti e gestiti correttamente. Il servizio *FlagsDLQEnqueuer* si occuperà dell'invio asincrono dei job alla coda *Flags Queue*. *JobsWorker* è progettato per scalare automaticamente in base alla quantità di job in coda. In caso di un aumento del carico di lavoro, sarà scalato automaticamente il numero di istanze per gestire la quantità di job in modo efficiente (vedi Cap. 4.8.3).

4.8.2 FlagsDLQEnqueuer Service

Il servizio *FlagsDLQEnqueuer* è responsabile di recuperare i flag non elaborati dal database *Flags DLQ Database*. I flag vengono inviati asincronamente alla coda *Flags Queue* per essere elaborati in seguito dal microservizio *Flags Service*. Dopo aver accodato con successo un flag, il servizio cancella la riga corrispondente al flag inviato nel database.

Il servizio *JobDLQEnqueuer* svolge un ruolo cruciale nel garantire la continuità del processo di elaborazione dei flag.

4.8.3 JobsWorkerAutoscaler Function

La funzione serverless *JobsWorkerAutoscaler* è responsabile della scalabilità automatica del servizio *JobsWorker*. L'invocazione della funzione è effettuata dal workload CronJob di Kubernetes[3], il quale eseguirà la funzione dei autoscaling cinque secondi prima di ogni round. La funzione controllerà il numero di job che sono stati accodati e il confronterà con il numero di repliche del servizio *JobsWorker*. Nel caso in cui sia necessario un numero maggiore di repliche, tramite le API di Kubernetes aumenterà il numero di repliche del servizio. Al contrario, un numero inferiore di job da eseguire può portare alla riduzione del numero di copie del servizio *JobsWorker*.

Questo garantisce che il servizio JobsWorker sia sempre in grado di gestire la quantità di lavoro richiesta senza problemi di sovraccarico o interruzioni. La scalabilità automatica è un fattore critico per garantire che il servizio sia altamente disponibile e performante per tutti gli utenti.

Eventi

JobExecutor

Il servizio *JobExecutor* produrrà un evento di tipo *JobExecutedEvent* quando avviene con successo l'esecuzione di un job letto dalla coda *Jobs Queue*. Tale evento scatenerà l'esecuzione della modifica dello stato del job nel microservizio Jobs Microservice (vedi Cap. 4.6). Lo stato del servizio diverrà "*EXECUTED*".

Modellazione dei dati del database

Flags DLQ Database

Per il salvataggio dei metadati dei flag che dovranno essere pianificati in maniera asincrona, verrà utilizzato il database *Jobs Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Leggere la lista dei flag nella DLQ filtrati per shard.
- Cancellare un flag filtrandolo per shard.

Bisogna quindi inserire come chiave di partizione la colonna *shard_number*, la quale formerà insieme a *flag* la chiave primaria di ogni flag.

Ecco la modellazione dei dati nel database *Flags DLQ Database* in Cassandra:

[3]CronJob è pensato per eseguire azioni pianificate regolari come backup, generazione di report e così via. Un oggetto CronJob è come una riga di un file crontab (tabella cron) su un sistema Unix.

```
CREATE TABLE svc.flags_dlq (
  shard_number smallint,
  flag text,
  PRIMARY KEY ((shard_number), flag)
);
```

Figura 4.14: CQL per creare il Flags DLQ Database

Nella seguente immagine è mostrato un esempio di lettura dei flag per shard:

```
cqlsh> SELECT * FROM svc.flags_dlq WHERE shard_number = 1;

 shard_number | flag
--------------+----------------------------------
            1 | 0SUUK0H88U2ZBZIMWHD35JV773X7RCK=
            1 | 5OSZ9VSOLYDQMAXD26TCBK8L8K1MSAS=
            1 | C98HXGPXZ5MK2YJMP7V1DJCYJ0J4VW2=
            1 | FQSTZ88TDL8BE2Q6RFSPF20PA0Q1438=
            1 | GA1KXEUD8TREMQ82PBS9ZMVC6FCSO0Z=
            1 | IQBX80JZXBK0BI0XH40FZXSBRU4TH4Q=

(6 rows)
cqlsh>
```

Figura 4.15: CQL leggere i flag per shard dal Flags DLQ Database

4.9 Flags Microservice

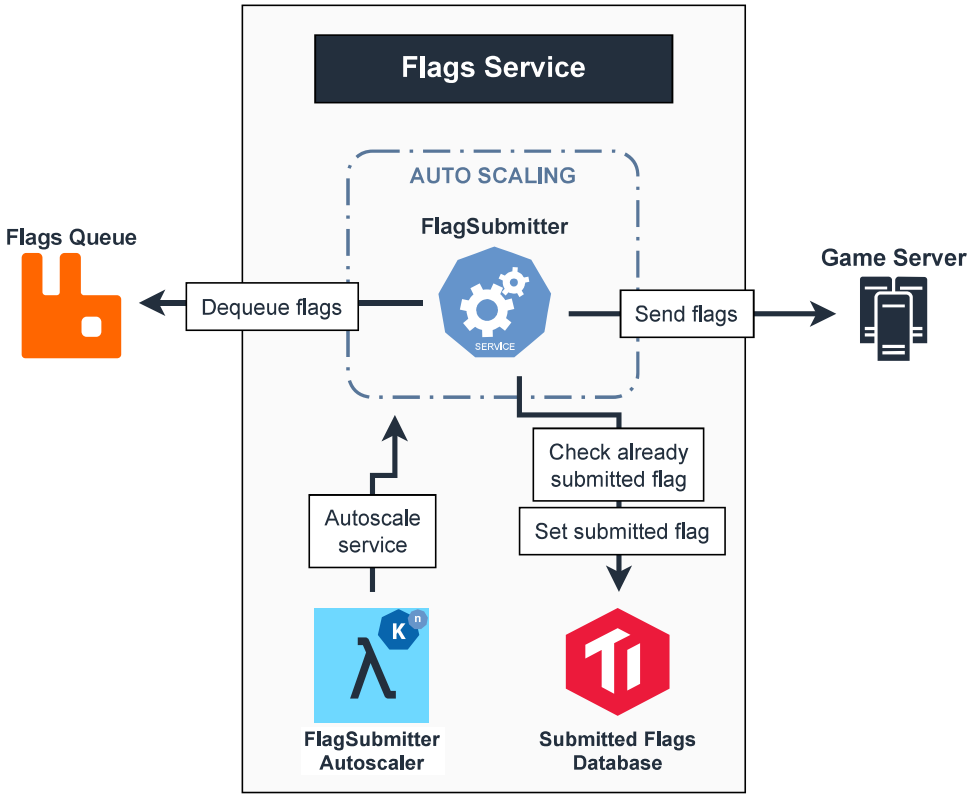

Figura 4.16: Architettura a basso livello del microservizio: Flags Microsevice

Come descritto nel capitolo 3.2.10, *Flags Microservice* è il microservizio che si occupa di inviare i flag in maniera asincrona verso il server di gioco. Il microservizio dovrà inviare i flag sottratti dagli exploit precedentemente eseguiti e inviarli al server di gioco tramite una richiesta HTTP POST.

API del microservizio

Il servizio *Flags Microsevice* non espone API interne, esterne o pubbliche.

4.9.1 FlagSubmitter Service

Il *FlagSubmitter* è un componente fondamentale per l'invio dei flag al server di gioco del servizio *Flags Microservice*. Questo componente è un servizio serverless che legge i flag da una coda e li invia al server di gioco tramite un'API POST.

Per ogni flag inviato al server di gioco viene inserita una riga nel database *Submitted Flags Database*, il quale tiene traccia dei flag inviati correttamente al server. Per evitare di inviare più volte lo stesso flag, viene effettuato un controllo preventivo sul database *Submitted Flags Database*, garantendo che vengano trasmessi al server di gioco solo flag unici.

Il servizio può essere sovraccaricato a causa del numero elevato di flag da inviare. Per evitare tale situazione viene utilizzata una funzione serverless che si occupa di scalare il numero di repliche del *FlagSubmitter* in conseguenza al numero di flag in coda. In caso di un

aumento del carico di lavoro, sarà scalato automaticamente il numero di istanze per gestire la quantità di flag in modo efficiente (vedi Cap. 4.9.2).

4.9.2 FlagSubmitterAutoscaler Function

La funzione *FlagSubmitterAutoscaler*, serverless, ha la responsabilità di gestire la scalabilità automatica del servizio *FlagSubmitter*. Viene invocata dal workload CronJob di Kubernetes ogni 30 secondi, che effettua periodicamente la funzione di autoscaling. La funzione monitora il numero di flag accodati e lo confronta con il numero di repliche del servizio *FlagSubmitter*.

In caso sia necessario aumentare le repliche, viene utilizzata l'API di Kubernetes per aumentare il numero di istanze del servizio. D'altra parte, se il numero di job da eseguire è inferiore, il numero di copie del servizio *FlagSubmitter* viene ridotto.

Modellazione dei dati del database

Submitted Flags Database

Per il salvataggio dei flag già inviati al server della competizione viene utilizzato il database *Submitted Flags Database*. Le operazioni che si intendono effettuare nel database sono quelle di:

- Inserimento di un flag.
- Lettura di un flag.

Le operazioni di lettura sono semplici e basate su un interrogazione diretta alla chiave. È possibile utilizzare il database TiKV chiave-valore per salvare i dati di interesse.

Ecco un esempio di come i dati potranno essere storicizzati nel database:

```
{
  "N3GLMI6DGJ1LOI9MOW0EVQHX48FU8CV=": "OK",
  "YH2BRZHQFJ82Z6HFHQVLVOFT5035CKS=": "OK",
  "B2UQSWWCI8H64342UES18829KWMJIGC=": "EXPIRED",
  "YTU3FZQTQSME49RHL6BUOQEOEHIIOCG=": "OK",
  "LX0KI4UF1YDCKRGZ0J40UFXR4OS4S40=": "OK",
  "T6WP9WQ7FG1GARXSEAW754EIRX4UDGB=": "WRONG",
  "W299N2CL3HDL4OCIW3T3I3RZST5X2U0=": "OK"
}
```

Listing 4.7: Esempio dati: Submitted Flags Database

Nell'esempio sono stati salvati dei flag che sono stati inviati al server di gioco.

4.10 Emissary-Ingress - API Gateway

L'API Gateway Emissary-Ingress è un componente fondamentale per l'architettura del sistema distribuito che si vuole progettare. Il suo compito principale è quello di gestire le richieste in entrata e di indirizzarle verso il microservizio corretto. Questo viene fatto utilizzando regole di routing configurate tramite le quali vengono abbinati gli endpoint specifici alle richieste. Il gateway utilizza una combinazione di informazioni come il path dell'URL, il metodo HTTP e i parametri della richiesta per determinare il microservizio corretto a cui indirizzare la richiesta [19]. Questo offre una grande flessibilità e semplicità nella gestione delle richieste, poiché tutte le richieste passano attraverso un unico punto di ingresso centralizzato prima di essere gestite dai microservizi corretti [10].

Le policy di routing dell'API Gateway del sistema andranno configurate come nella tabella 4.7. La configurazione del routing di ogni API deve fare riferimento alle API enunciate nella progettazione a basso livello dei microservizi delle tabelle 4.1 per l'*Authentication Microservice*, 4.2 per il *Frontend Microservice*, 4.3 per il *Vulnboxes Microservice*, 4.4 per il *Services Microservice*, 4.5 per l'*Exploits Microservice*, 4.6 per il *Jobs Microservice*, 4.7 per il *Job Scheduler Microservice*, 4.8 per il *Job Executor Microservice* e 4.9 per il *Flags Microservice*.

La seguente tabella indica quali sono le policy di protezione che vanno applicate per ogni microservizio che espone un endpoint pubblico o privato.

Metodi HTTP	Endpoint	Microservizio	Autenticazione
GET	/	Frontend Microservice	-
GET	/res/*	Frontend Microservice	-
POST	/api/auth/*	Auhtentication Microservice	-
GET POST UPDATE DELETE PUT	/api/vulnboxes/*	Vulnboxes Microservice	JWT token
GET POST UPDATE DELETE PUT	/api/services/*	Services Microservice	JWT token
GET POST UPDATE DELETE PUT	/api/exploits/*	Exploits Microservice	JWT token
GET PUT	/api/jobs/*	Jobs Microservice	JWT token

Tabella 4.7: Regole di routing delle API pubbliche e private del sistema distribuito esposte dall'API Gateway.

CAPITOLO 5

Conclusioni

In questo ultimo capitolo vengono introdotte le conclusioni che si possono trarre dallo sviluppo della tesi. Verrà ricordato il processo seguito per analizzare e progettare il sistema distribuito, con particolare riferimento alle soluzioni tecniche adottate in fase di progettazione per soddisfare i requisiti non funzionali descritti, nella fase di analisi, al capitolo 2.3. Vengono infine forniti degli spunti per possibili sviluppi futuri.

In conclusione, la progettazione di un sistema è un processo complesso che richiede una metodologia rigorosa e una comprensione approfondita dei requisiti. Nella progettazione del sistema in questione sono stati seguiti una serie di passi, descritti di seguito.

Prima di iniziare la progettazione, è necessario definire lo scopo e il contesto in cui verrà utilizzato. L'analisi dei requisiti è un passo cruciale che consente di comprendere le funzionalità e le caratteristiche richieste dal sistema. I requisiti funzionali descrivono le funzionalità che il sistema deve fornire, mentre i requisiti non funzionali riguardano aspetti come le prestazioni, la sicurezza, la scalabilità e la disponibilità. L'analisi dei volumi di traffico consente di identificare le esigenze di capacità per effettuare una progettazione in modo adeguato. La progettazione del sistema distribuito richiede una particolare attenzione alla scalabilità e alla disponibilità, poiché questi sistemi sono costituiti da molti componenti che devono funzionare in modo affidabile e sincronizzato. La progettazione ad alto livello e a basso livello devono tenere conto di questi fattori, definendo architetture e componenti che possano scalare e rispondere alle esigenze del sistema in modo efficiente. Per garantire la scalabilità, disponibilità e performance del sistema distribuito, sono state adottate diverse tecniche, le quali verranno trattate nei paragrafi successivi.

Architettura a microservizi

La scelta di un'architettura a microservizi è stata cruciale per poter sfruttare i vantaggi dei sistemi distribuiti e dei microservizi, enunciati nei capitoli 1.1.1 e 1.5.1. I microservizi sono un modello architetturale software che prevede la suddivisione di un sistema complesso in piccoli servizi indipendenti e autonomi, ciascuno dedicato a una specifica funzionalità. Questo approccio promuove la modularità, la flessibilità e l'efficienza del sistema, rendendo più semplice la manutenzione e l'evoluzione del sistema nel tempo.

Le soluzioni che sono state adottata per implementare l'architettura a microservizi sono:

- **Tecnologia a container**: la semplicità, leggerezza e autonomia dei container consentono di adattarsi alla perfezione alla progettazione e implementazione di un architettura a microservizi.

- **Kubernetes**: la scelta dell'orchestratore Kubernetes ha consentito di poter gestire la distribuzione dei container in modo rapido ed efficace.

Architettura guidata ad eventi

La capacità di far comunicare i servizi in maniera asincrona tramite l'architettura guidata ad eventi (event-driven architecture) consente di accoppiare i microservizi in modo lasco, migliorando la resilienza e l'affidabilità del sistema.

La scelta di Knative Eventing è stata cruciale per poter implementare tale architettura. L'utilizzo dell'architettura guidata ad eventi ha facilitato, tramite Knative Eventing, la progettazione del sistema, consentendo una maggiore scalabilità e un accoppiamento lasco tra i microservizi.

Bilanciamento del carico

Il load balancing, o bilanciamento del carico, consiste nel bilanciare la quantità di lavoro tra i vari server, in modo da evitare che alcuni nodi diventino sovraccarichi e altri sottoutilizzati. Questo aiuta a garantire una maggiore disponibilità e scalabilità del sistema.

Kubernetes è stata la scelta tecnica capace di consentire il bilanciamento del carico tra le varie repliche di ogni servizio del sistema. In tale maniera è stato possibile aumentare la disponibilità del sistema e le performance dei servizi.

Autoscaling

L'autoscaling è la tecnica che consente di aumentare o ridurre dinamicamente il numero di istanze di un servizio in base alle esigenze di carico. Questo aiuta a gestire in modo ottimale la capacità di un sistema di gestire un aumento del carico di lavoro in modo efficiente, senza compromettere le prestazioni.

Per garantire la autoscaling sono state adottate principalmente due strategie:

- **Knative Serving**: è stato utilizzato per consentire ai servizi senza stato, di essere scalati in base la numero di richieste ricevute per secondo oppure all'utilizzo di CPU.
- **Kubernetes CronJob che invoca una funzione serverless**: per la scalabilità dei servizi senza stato che non possono essere scalati in modo efficace controllando l'utilizzo delle risorse, viene richiamata tramite un'esecuzione periodica da parte di Kubernetes CronJob di una funzione serverless. Tale funzione si occuperà di scalare tali servizi in base ad altre metriche. Tale soluzione viene utilizzata, per esempio, per scalare i nodi worker che eseguono gli exploit in modo proporzionale al numero di job in coda.

Replicazione dei dati

La replicazione consiste nella creazione di copie multiple dei dati su più nodi di un sistema distribuito, in modo che se uno dei nodi viene a mancare o diventa non disponibile, le copie dei dati possono essere utilizzate per garantire la continuità del servizio.

Nel sistema, tutti i servizi con stato, sfruttano la replicazione dei dati per ottenere alta disponibilità e resilienza ai guasti. In particolare Cassandra, TiKV, Rook, Emissary-Ingress e RabbitMQ devono essere configurati in modo tale da poter garantire la continuità del servizio in risposta alla terminazione di un nodo.

Partizionamento dei dati

Il partizionamento dei dati, noto anche come sharding, consiste nella divisione dei dati in parti più piccole che possono essere gestite da nodi diversi all'interno di un sistema distribuito. Questo aiuta a ridurre il carico su singoli nodi, aumentando così la scalabilità e la disponibilità del sistema.

Viene sfruttato il partizionamento dei dati per aumentare la scalabilità e la disponibilità dei servizi che sono responsabili della creazione (vedi Cap. 4.6.2), pianificazione (vedi Cap. 4.7.1) e accodamento di job (vedi Cap. 4.7.2). In tale modo è possibile aumentare la disponibilità e le performance dei servizi con stato.

Consistenza dei dati replicati e partizionati

Per garantire che i dati replicati e partizionati siano sincronizzati e coerenti, possono essere utilizzati algoritmi di sincronizzazione dei dati. Questi algoritmi gestiscono la replicazione e la sincronizzazione dei dati, garantendo che tutte le copie dei dati siano coerenti e disponibili in ogni momento.

Nel sistema i servizi con stato replicati e partizionati mirano ad essere eventualmente consistenti. In particolare Cassandra, TiKV, Rook, Emissary-Ingress e RabbitMQ utilizzano internamente degli algoritmi di sincronizzazione, garantendo consistenza eventuale dei dati tra nodi differenti.

Sicurezza delle richieste

La sicurezza è un fattore cruciale da considerare nella progettazione di un qualsiasi sistema informatico. Una violazione della sicurezza nel sistema distribuito progettato in questa tesi, può comportare la perdita di dati e l'interruzione dei servizi. La sicurezza, nell'architettura a microservizi deve essere garantita a livello di ogni singolo servizio e nella comunicazione tra questi.

La sicurezza a livello di comunicazione tra servizi non è stata trattata in quanto, come descritto nel capitolo 2.1, il sistema verrà eseguito in un ambiente isolato ed esporrà dei servizi agli utenti. La comunicazione tra servizi non può quindi né essere intercettata, né dirottata.

Nella progettazione del sistema è stato inserito un API Gateway, chiamato Emissary-Ingress (vedi Cap. 3.2.1 e 4.10), il quale si occupa della gestione delle richieste, controllando e limitando l'accesso ai servizi. Tramite l'utilizzo di un API Gateway, ogni servizio può essere monitorato e gestito separatamente, aumentando la resilienza del sistema e minimizzando i rischi di attacchi o falle di sicurezza.

5.1 Sviluppi futuri

L'architettura a microservizi consente al sistema distribuito di essere facilmente estensibile con nuove funzionalità. Nella progettazione ad alto livello (vedi Cap. 3.2) e in quella a basso livello (vedi Cap. 4) sono presenti tutte le specifiche per poter comprendere le caratteristiche di ogni componente e come essi interagiscono tra loro. La documentazione fornita nella tesi in tali capitoli è utile per poter sviluppare funzionalità aggiuntive nel sistema.

I possibili spunti di sviluppo del sistema potrebbero includere:

1. **Implementazione del sistema**: una volta che tutti i requisiti sono stati analizzati e definiti, il prossimo passo sarebbe implementare effettivamente il sistema. Ciò richiede l'integrazione di diversi componenti esterni, come Cassandra, TiKV, Rook, Emissary-Ingress e RabbitMQ, e lo sviluppo e realizzazione dei servizi descritti, analizzati e progettati nei capitoli 2.3, 3 e 4.

2. **Aumento dell'osservabilità dei servizi**: potrebbe essere utile aumentare l'osservabilità dei servizi in esecuzione, al fine di poter individuare eventuali problemi o inefficienze più facilmente. Ciò potrebbe essere realizzato tramite l'aggiunta di componenti che monitorano lo stato dei microservizi, favorendo una maggiore facilità dell'interpretazione delle metriche di monitoraggio.

3. **Aggiunta di un sistema di logging dell'esecuzione dei job**: potrebbe essere utile mantenere traccia delle esecuzioni dei job, al fine di poter analizzare e comprendere meglio gli exploit caricati nel sistema per risolvere i bug negli script. Ciò potrebbe essere realizzato tramite l'aggiunta di un microservizio che si occupi di logging. Tale servizio dovrebbe registrare le informazioni e i log delle esecuzioni dei job, facilitando la risoluzione dei bug.

4. **Aggiunta dell'analisi statistica dei flag inviati**: potrebbe essere interessante analizzare i dati raccolti tramite l'invio dei flag al server di gioco, al fine di identificare eventuali problemi o tendenze. Ciò potrebbe essere realizzato tramite l'aggiunta di un microservizio che si occupi analizzare i dati statistici dei flag. Tale servizio dovrebbe registrare le informazioni e i log dell'invio dei flag, elaborando delle statistiche su di essi da fornire all'utente.

5. **Modifica del partizionamento dei job rendendolo dinamico**: potrebbe essere utile adattare il partizionamento dei job in base alle esigenze del sistema, al fine di migliorare le prestazioni e la scalabilità. Ciò potrebbe essere realizzato tramite la modifica dei microservizi di partizionamento in modo che l'assegnazione delle partizioni sia dinamica, in grado di adattarsi alle esigenze di carico in tempo reale.

Bibliografia

[1] ALEX, X. (2020), *System Design Interview – An insider's guide*, Independently published (June 12, 2020). (Cited at page 20)

[2] AMAZON-WEB-SERVICES (2023), «Event-Driven Architecture», URL `https://aws.amazon.com/event-driven-architecture/`. (Cited at page 15)

[3] AMAZON-WEB-SERVICES (2023), «Horizontal scaling - AWS Well-Architected Framework», URL `https://wa.aws.amazon.com/wellarchitected/2020-07-02T19-33-23/wat.concept.horizontal-scaling.en.html`. (Cited at pages 8 e 9)

[4] AMAZON-WEB-SERVICES (2023), «Serverless Computing – Amazon Web Services», URL `https://aws.amazon.com/serverless/`. (Cited at page 43)

[5] AMAZON-WEB-SERVICES (2023), «What are Microservices?», URL `https://aws.amazon.com/microservices/`. (Cited at pages v, 11 e 12)

[6] AMAZON-WEB-SERVICES (2023), «What Is Distributed Computing?», URL `https://aws.amazon.com/what-is/distributed-computing/`. (Cited at pages 3 e 4)

[7] APACHE-CASSANDRA (2023), «Apache Cassandra | Apache Cassandra Documentation», URL `https://cassandra.apache.org/_/index.html`. (Cited at pages 44 e 45)

[8] CHRIS, R. (2019), *Microservices Patterns: With Examples in Java*, Manning Publications. (Cited at pages v, 5, 6, 7, 8 e 10)

[9] CHRIS, R. (2023), «Pattern: Access token», URL `https://microservices.io/patterns/security/access-token.html`. (Cited at page 36)

[10] CHRIS, R. (2023), «Pattern: API Gateway», URL `https://microservices.io/patterns/apigateway.html`. (Cited at pages 36 e 85)

[11] CHRIS, R. (2023), «Pattern: Microservice Architecture», URL `https://microservices.io/patterns/microservices.html`. (Cited at pages v, 12 e 13)

[12] CHRIS, R. (2023), «Pattern: Monolithic Architecture», URL `https://microservices.io/patterns/monolithic.html`. (Cited at pages 10 e 11)

[13] CLOUD-NATIVE-COMPUTING-FOUNDATION (2023), «Cloud Native Computing Foundation - Mission», URL `https://github.com/cncf/foundation/blob/main/charter.md`. (Cited at page 41)

[14] CLOUDEVENTS.IO (2023), «CloudEvents», URL `https://cloudevents.io/`. (Cited at page 44)

[15] CRI O.IO (2023), «cri-o», URL `https://cri-o.io/`. (Cited at page 43)

[16] CYBERCHALLENGE.IT (2023), «CyberChallenge.IT», URL `https://cyberchallenge.it/`. (Cited at page 18)

[17] DEIMOS.IO (2023), «Authentication and Authorization in a Distributed System | Deimos», URL `https://deimos.io/post/authentication-and-authorization-in-a-distributed-system`. (Cited at pages v, 37 e 38)

[18] DOCKER.COM (2023), «Docker: Accelerated, Containerized Application Development», URL `https://www.docker.com/`. (Cited at pages 42 e 44)

[19] EMISSARY-INGRESS (2023), «emissary-ingress/emissary: open source Kubernetes-native API gateway for microservices built on the Envoy Proxy», URL `https://github.com/emissary-ingress/emissary`. (Cited at pages 46 e 85)

[20] GEORGE, C., JEAN, D., TIM, K. e GORDON, B. (2012), *Distributed Systems: Concepts and Design*, Addison-Wesley. (Cited at pages 3 e 4)

[21] GILBERT, S. e LYNCH, N. (2002), «Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services», *SIGACT News*, vol. 33 (2), p. 51–59, URL `https://doi.org/10.1145/564585.564601`. (Cited at page 5)

[22] HIJAZI, N. (2016), «Dependency Hell in Microservices and How to Avoid It», URL `https://www.linkedin.com/pulse/dependency-hell-microservices-how-avoid-nabil-hijazi/`. (Cited at pages v, 14 e 15)

[23] JWT.IO (2023), «JSON Web Tokens - jwt.io», URL `https://jwt.io/`. (Cited at page 36)

[24] KNATIVE.DEV (2023), «Home - Knative», URL `https://knative.dev/docs/`. (Cited at pages 43 e 45)

[25] KUBERNETES.IO (2023), «Kubernetes», URL `https://kubernetes.io/`. (Cited at page 42)

[26] MANSUROV, A. (2016), «A CTF-Based Approach in Information Security Education: An Extracurricular Activity in Teaching Students at Altai State University, Russia», *Online published*, p. 159–162, URL `https://doi.org/10.5539/mas.v10n11p159`. (Cited at page 17)

[27] MAXIMILIEN, M. (2022), «KubeCon Europe 2022 - IBM Developer», URL `https://developer.ibm.com/blogs/kubecon-europe-2022/`. (Cited at page 47)

[28] MOZILLA.ORG (2023), «SPA (Single-page application) - MDN Web Docs Glossary: Definitions of Web-related terms | MDN», URL `https://developer.mozilla.org/en-US/docs/Glossary/SPA`. (Cited at page 38)

[29] NETFLIX-TECHNOLOGY-BLOG (2012), «Embracing the Differences: Inside the Netflix API Redesign», URL `https://netflixtechblog.com/embracing-the-differences\-inside-the-netflix-api-redesign-15fd8b3dc49d`. (Cited at page 36)

[30] OPENCONTAINERS.ORG (2023), «Open Container Initiative - Open Container Initiative», URL `https://opencontainers.org/`. (Cited at page 42)

[31] PAUL, S. e DAVID, H. (2022), «Reducing cold start times in Knative, Red Hat OpenShift Serverless, and IBM Cloud Code Engine - IBM Developer», URL `https://developer.ibm.com/articles/reducing-cold-start-times-in-knative/`. (Cited at page 47)

[32] RABBITMQ (2023), «Messaging that just works — RabbitMQ», URL `https://www.rabbitmq.com/`. (Cited at page 45)

[33] RED-HAT (2023), «What does an API gateway do?», URL `https://www.redhat.com/en/topics/api/what-does-an-api-gateway-do`. (Cited at page 36)

[34] ROOK.IO (2023), «Rook», URL `https://rook.io/`. (Cited at page 45)

[35] TIKV.ORG (2023), «TiKV | TiKV is a highly scalable, low latency, and easy to use key-value database.», URL `https://tikv.org/`. (Cited at page 45)