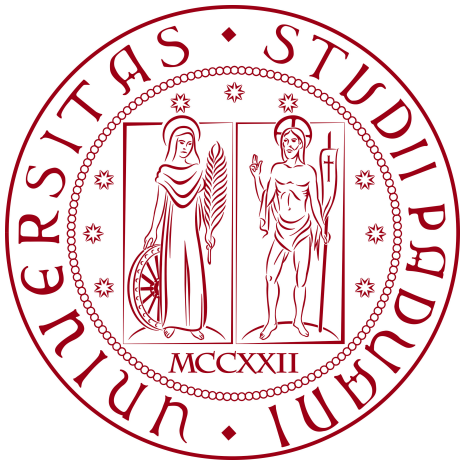
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Sviluppo di un servizio di Generative AI per la generazione di report di vulnerability assessment a partire da codice sorgente.

Tesi di Laurea Triennale

Relatore
Prof. Ballan Lamberto

Laureanda
Nancy Kalaj
Matricola 2042321

Anno Accademico 2023-2024

Ringraziamenti

Desidero esprimere la mia gratitudine al professor Ballan Lamberto, mio relatore, per l'aiuto e il sostegno che mi ha dato durante la stesura dell'elaborato.

Vorrei anche ringraziare, con affetto, la mia famiglia per avermi sempre sostenuto nonostante i miei cambiamenti di percorso repentini e apparentemente ingiustificati; tutto ciò che ho realizzato lo devo a loro. Non sarei mai potuta arrivare fin qui senza farmi ispirare dalla loro dedizione al lavoro e dalla loro disciplina, ma anche e soprattutto dalla loro umiltà, compassione e passione per la vita.

Desidero poi ringraziare i miei amici: in particolare Lorenzo perché ha scelto di lavorare con me a svariati progetti nonostante la mia mancanza di esperienza pregressa e si è dimostrato un collega inestimabile, oltre che un amico affidabile su cui poter sempre contare; Elisa perché ha sempre creduto in me e mi è stata accanto anche nei momenti più bui della mia vita, senza lasciare che rinunciassi a quanto mi rende felice.

Ci tengo a ringraziare anche i miei compagni di SWE, che hanno contribuito direttamente alla riuscita del progetto e, soprattutto, alla sua conclusione in tempi ragionevoli; i miei compagni di stage, per aver reso l'esperienza ancora più memorabile ed avermi supportato fino all'ultimo momento; e, naturalmente, anche il mio tutor aziendale e il resto della bellissima squadra di zero12, per averci ospitati con tanta cordialità ed insegnato tantissimo in così poco tempo.

Padova, Luglio 2024 *Nancy Kalaj*

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di *stage*, della durata di trecentoventi ore, dalla laureanda Nancy Kalaj presso l'azienda zero12 s.r.l., sede di Padova. Lo scopo dello *stage* è la realizzazione di un servizio che, dato in input il codice sorgente di un applicativo *software web* o *mobile*, utilizzi i servizi di Generative AI offerti da Amazon Web Services (AWS), in particolare AWS Bedrock, per produrre un *report* contenente un punteggio di vulnerabilità del *software* analizzato e indicazioni sulle aree che necessitano di intervento per risolvere le vulnerabilità individuate.

La prima fase consiste nell'apprendimento delle tecnologie e dei linguaggi di programmazione necessari per la realizzazione del progetto. Successivamente, il progetto si articola nelle attività seguenti:

- Creazione di un servizio che, dato in input del codice sorgente, fornisca un'analisi delle vulnerabilità e produca un sistema di *scoring* da usare per la generazione del *report*
- Creazione del *report* con punteggio del livello di sicurezza e tutte le informazioni di dettaglio per il team tecnico per svolgere attività di risoluzione
- Creazione della fase del processo di *deploy* per avviare l'analisi del codice di un progetto in automatico

Per concludere, la fase finale del progetto consiste in attività di *testing*, anche del funzionamento di vari modelli offerti da Bedrock nello stesso perimetro di contesto, e nella produzione della documentazione del lavoro svolto.

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici sorgenti

Capitolo 1

Introduzione

In questo capitolo viene fornita una breve panoramica dell'azienda e dell'idea alla base del progetto didattico.

1.1 L'azienda

zero12 s.r.l è un'azienda informatica fondata nel 2012 e specializzata nello sviluppo di soluzioni *software* e consulenza tecnologica. Partner di AWS, l'azienda offre servizi di migrazione al *cloud* e gestione di infrastrutture Cloud Native, sviluppo di applicazioni *web* e *mobile* sfruttando tecnologie di Augmented Reality (AR) e di Internet of Things (IoT), creazione di modelli *custom* di Machine Learning per scopi diversificati e consulenze per l'innovazione digitale. L'obiettivo principale di zero12 è dunque utilizzare appieno il potenziale delle tecnologie *cloud* AWS, seguendo l'evoluzione di questo paradigma tecnologico da vicino, per fornire ai propri clienti soluzioni efficaci ed indispensabili.

1.2 L'idea

La sicurezza delle applicazioni è una priorità crescente in qualsiasi contesto di sviluppo professionale, poiché identificare e mitigare eventuali vulnerabilità prima che il *software* venga distribuito è essenziale per prevenire attacchi e proteggere dati sensibili. Tra le tecniche tradizionali adoperate per perseguire tale scopo vi sono *code reviews* condotte manualmente da professionale esperto di *cybersecurity* e l'adozione di strumenti specializzati di Static Application Security Testing (SAST); in questo contesto, l'emergere di modelli sempre più avanzati di Generative AI costituisce una novità importante, dato che alcuni di essi hanno già dimostrato di avere il potenziale di complementare e potenziare la revisione di codice sorgente. Il progetto nasce dunque dall'esigenza di integrare la sicurezza del *software* direttamente nel flusso di lavoro *CI/CD* caratteristico dell'ambiente di sviluppo agile e DevOps aziendale; infatti, per poter garantire che le modifiche significative apportate al codice vengano valutate sfruttando il potenziale della Generative AI e senza interrompere il ciclo di sviluppo, è stato concepito un servizio in grado di prelevare automaticamente il codice sorgente di una *repository* GitHub per sottoporlo ad una revisione approfondita da parte di

un modello di Generative AI, mirata ad identificarne le vulnerabilità e assegnare uno *score* riassuntivo. I risultati di tale revisione vengono poi rielaborati e resi reperibili all'interno di un *report*, che viene automaticamente inserito all'interno della *repository* di partenza.

1.3 Organizzazione del testo

Il secondo capitolo fornisce una breve introduzione al progetto, seguita dagli obiettivi di formazione delle attività previste dallo *stage*

Il terzo capitolo fornisce alcuni esempi di applicazioni recenti della Generative AI nel contesto della *cybersecurity*

Il quarto capitolo fornisce una spiegazione riassuntiva di ogni tecnologia e strumento di supporto utilizzati durante la fase di implementazione del progetto

Il quinto capitolo descrive la fase di progettazione del servizio

Il sesto capitolo espone gli aspetti più rilevanti della fase di implementazione del servizio

Il settimo capitolo approfondisce alcune considerazioni di carattere pratico sull'uso del servizio realizzato in un contesto reale

Nell'ottavo capitolo si illustrano alcune considerazioni finali sullo svolgimento del progetto

Riguardo la stesura del testo, relativamente al documento sono state adottate le seguenti convenzioni tipografiche:

- Gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del presente documento
- I termini in lingua straniera o facenti parti del gergo tecnico sono evidenziati con il carattere *corsivo*

Capitolo 2

Descrizione

In questo capitolo viene fornita una breve introduzione al progetto, seguita dagli obiettivi di formazione delle attività previste dallo *stage* e dalla loro distribuzione temporale.

2.1 Introduzione al progetto

Il progetto consiste nella creazione di un servizio chiamato Vulnerability Assessment Service (VAS), che ha lo scopo di prelevare il codice sorgente a partire da una *repository* GitHub in seguito a determinati *trigger*, condurre una *code review* utilizzando uno dei modelli di Generative AI messi a disposizione da AWS Bedrock e rendere disponibili i risultati di suddetta revisione all'interno di un *report* inserito nella *repository* di partenza. Lo scopo del progetto nel contesto di sviluppo aziendale è dunque duplice: aumentare l'efficienza operativa del processo di sviluppo creando un'automazione incaricata di rilevare eventuali vulnerabilità nel codice; testare l'utilità e la versatilità di un modello di Generative AI nel contesto descritto, dato che vi è la possibilità che questi si evolvano per arrivare a comprendere il contesto del codice in modo più sofisticato rispetto agli strumenti di analisi statica tradizionali, a simulare il comportamento del codice in scenari diversificati per individuare *exploits* altrimenti difficilmente rilevabili, a fornire suggerimenti intelligenti o modifiche al codice per risolvere le vulnerabilità e a ridurre il numero di falsi positivi (comuni negli strumenti SAST) [17].

2.2 Obiettivi formativi

Gli obiettivi formativi dell'attività di *stage* sono i seguenti:

- Imparare a sviluppare un prodotto *software* in un contesto innovativo come l'utilizzo dell'intelligenza generativa in ambito di *cybersecurity*
- Entrare in contatto con le dinamiche quotidiane di sviluppo di un applicativo *software* utilizzando i servizi *cloud* AWS
- Imparare a gestire le complessità di integrazione di sistemi, in particolar modo la comunicazione tra GitHub e servizi AWS

- Apprendere l'uso di metodologie agili di sviluppo

2.3 Obiettivi del progetto

All'inizio del primo giorno di *stage* si è svolto un incontro con il tutor aziendale per definire gli obiettivi principali del progetto, ovvero:

- Individuazione delle vulnerabilità esatte da includere nella *code review* condotta tramite il modello fornito da Bedrock
- Individuazione del sistema di *scoring* da adottare affinché si possa assegnare uno *score* cumulativo al codice revisionato
- Individuazione del modello più adatto al contesto di applicazione tra quelli forniti da Bedrock
- Individuazione del *prompt* da utilizzare per esaminare il codice sorgente ed ottenere una risposta strutturata nel formato previsto (*prompt engineering*)
- Definizione dei *trigger* che innescano il servizio automaticamente tramite GitHub Actions
- Definizione dei servizi AWS da utilizzare per processare il codice sorgente in input, condurre la revisione, esaminare le risposte del modello e costruire il *report*
- Individuazione del metodo di autenticazione richiesto per fare in modo che il servizio possa usufruire dell'Application Program Interface (API) di GitHub

2.4 Pianificazione

L'attività di *stage* ha avuto una durata complessiva di 8 settimane per un totale di 320 ore; le attività previste sono state suddivise in 2 *sprint* della durata di una settimana e 3 *sprint* della durata di 2 settimane, come concordato inizialmente con l'azienda nel Piano di Lavoro e delineato nella seguente tabella:

Periodo	Durata (ore)	Attività
1	40	Studio della modalità di lavoro, del linguaggio di programmazione *Typescript*, dei servizi AWS, del *framework Serverless* e del contesto applicativo
2	80	Sviluppo del servizio che, dato in input del codice sorgente, fornisce un'analisi delle vulnerabilità e utilizza un sistema di *scoring* adatto alla generazione del *report*
3	80	Integrazione del servizio nella *CI/CD* per avviare l'analisi del codice di una *repository* in seguito a *trigger* pre-determinati

Continua nella pagina successiva

Tabella 2.1 – continua dalla pagina precedente

Periodo	Durata (ore)	Attività
4	80	Sviluppo del *report* contenente lo *score* del livello di sicurezza e tutte le informazioni di dettaglio per il team tecnico per svolgere attività di risoluzione
5	40	Attività di *testing* e produzione della documentazione del lavoro svolto.

Tabella 2.1: Descrizione delle attività pianificate per ciascun periodo

Il lavoro è stato svolto in modalità ibrida e le attività previste non hanno subito variazioni nei contenuti rispetto alla pianificazione; tuttavia, il numero di ore effettivamente dedicato ad alcune attività risulta diverso rispetto a quanto specificato in partenza, così come l'ordine di esecuzione (vedere sezione 8.1). Questo è dovuto principalmente alla natura sperimentale del progetto e alle risorse da dedicare allo studio del contesto applicativo e alla progettazione iniziale, che si sono rivelate più onerose del previsto. In ogni caso, tutti gli obiettivi del progetto sono stati portati a compimento, grazie all'adozione di una modalità di sviluppo agile che ha previsto incontri giornalieri di 15 minuti circa con il tutor aziendale per discutere dei progressi ottenuti e di eventuali problematiche. L'attività di *stage* si è conclusa con una presentazione interna del progetto e una *demo* dei risultati del lavoro svolto.

Capitolo 3

Contesto applicativo

In questo capitolo vengono forniti alcuni esempi di applicazioni recenti della *Generative AI* nel contesto della *cybersecurity*.

3.1 Applicazioni della Generative AI

L'avvento della Generative AI, l'apice dell'evoluzione dell'Artificial Intelligence (AI) nell'ultimo decennio, nel dominio della *cybersecurity* ha aperto nuove frontiere per quanto riguarda sia il rafforzamento delle difese dei sistemi informatici che l'utilizzo di attacchi più complessi e difficili da rilevare da parte di *cyber attackers*. Infatti, si anticipa che la Generative AI possa un giorno contribuire ad un'analisi più approfondita del codice o dell'ambiente di esecuzione rispetto alle tecniche tradizionali di SAST, amplificando le funzionalità degli strumenti odierni di *scanning* di vulnerabilità [17].

Le applicazioni della Generative AI sono numerose. Ad esempio, si è testato il potenziale di Large Language Model (LLM) come GPT3.5 nel contesto del Penetration testing, facendo in modo che il modello potesse analizzare lo stato di una macchina virtuale volutamente resa vulnerabile per individuare vulnerabilità di basso livello [5]; in particolare, si è chiesto al modello di formulare comandi per la *shell* di *Linux* che venivano automaticamente eseguiti sulla Virtual Machine (VM) tramite *SSH*, con lo scopo di acquisire i privilegi del *root user* del sistema. La riuscita dell'esperimento dimostra il potenziale di impiegare i LLM come Sparring partner per potenziare gli sforzi di analisi dei *penetration testers*.

Naturalmente, sono state testate anche le applicazioni di ChatGPT in diversi ambiti della *cybersecurity*. Il modello può contribuire in modo significativo alla creazione e al monitoraggio di *honeypots*, ossia sistemi di sicurezza progettati per attirare, rilevare e analizzare attacchi informatici; creando ambienti apparentemente vulnerabili e attraenti, ma in realtà isolati e monitorati, è possibile utilizzare tali sistemi per rilevare attacchi, apprendere dati e informazioni dettagliate sulle tattiche degli aggressori e proteggere i veri sistemi usati a modello. In particolare, è stato chiesto a ChatGPT di simulare il comportamento di vari sistemi operativi (*Linux*, *Windows*,

Mac) e applicazioni, in modo che questo potesse emulare le risposte realmente date da tali entità in seguito a comandi particolari, consentendo di interagire dinamicamente con gli *honeypots* così realizzati [8].

ChatGPT è anche risultato utile nell'analisi di *code snippets* con lo scopo di rilevare eventuali vulnerabilità (seppur non nel contesto di una *code review* strutturata a dovere), come *buffer overflow* o *SQL injections* [9], e si è dimostrato in grado sia di sfruttarle nella generazione di codice malevolo che di consigliare rimedi efficaci per mitigarle.

Un aspetto interessante dell'applicazione della Generative AI alla *cybersecurity* è l'importanza rivestita dall'abilità dei modelli di sviluppare Commonsense reasoning [16], per poter contestualizzare i dati che vengono presentati loro in modo sensato; anche la *performance* di ChatGPT soffre quando si rende necessaria una conoscenza approfondita del contesto di applicazione dei *prompt* (*contextual knowledge*), portando a risultati incoerenti e la mancata rilevazione di attacchi (*reverse shell*, *ARP spoofing*, *malformed heartbeat request*) o anomalie all'interno di file di *logging* [12]. Di contro, il modello sembra rispondere meglio quando viene fornito un input corredato da una descrizione testuale di un particolare evento, spesso da parte di personale esperto del settore applicativo.

D'altro canto, nel tentativo di testare il livello di conoscenza di *cybersecurity* di 25 LLM tramite i *dataset* CyberMetric, alcuni modelli (GPT-4o, Mixtral-8x7B-Instruct, e GPT4-turbo) hanno esibito una *performance* nettamente migliore di specialisti umani su un *subset* ristretto a 80 domande *multiple-choice*, pur denotando ancora difficoltà su argomenti frutto di ricerche recenti o ragionamenti particolarmente complessi [15].

Altri lavori dimostrano l'abilità dei LLM nel mappare descrizioni di minacce relativamente vaghe a tattiche formalmente spiegate dal MITRE, piuttosto che nella generazione di *network payloads* malevoli facendo riferimento alle tattiche di attacco descritte dal MITRE [2], dimostrandone l'effettivo livello di comprensione di vari domini della *cybersecurity*; in AGIR [10], un *tool* di Natural Language Processing (NLP) progettato per automatizzare la creazione di *reports* di Cyber Threat Intelligence (CTI), si impiegano grafici STIX e ChatGPT per creare *report* con un *recall value* di 0.99 non contenenti allucinazioni e di alta qualità rispetto a valutazioni sintattiche e linguistiche di fluidità e naturalezza del linguaggio utilizzato. ChatGPT dimostra risultati promettenti anche nella comprensione di codice e riparazione di *bugs* [6], seppur l'efficacia dei LLM nel gestire le logiche complesse che sottendono ad alcuni *bug*, la cui comprensione si basa su ragionamenti non banali e analisi ad ampio spettro del sistema in esame, rimane limitata.

In aggiunta, il tentativo di utilizzare ChatGPT per eseguire operazioni di *multi-*

label classification sulle debolezze elencate nel Common Weakness Enumeration (CWE)[1] non ha conseguito risultati significativi, in quanto il modello ha avuto difficoltà nel classificare frammenti di codice *Java* secondo le 5 CWE più frequenti all'interno del *database* da cui proveniva il sorgente (CWE 20, 200, 502, 611 e 79) [3]; da notare però come gli autori suggeriscano l'uso del Chain-of-thought prompting per scopi di identificazione di vulnerabilità, in quanto quest'ultimo può affinare le capacità di ragionamento dei LLM. Un altro studio dimostra come ChatGPT sia stato in grado di mappare la corrispondenza tra 13,513 vulnerabilità listate nel Common Vulnerability and Exposures (CVE)[2] e le CWE corrispondenti, individuando tutte le CWE correttamente in più del 50.88% dei casi [7]; tuttavia, non si può dire lo stesso per il tentativo di mappare le CVE alle tecniche ATT&CK del MITRE, che ha dimostrato come il modello non avesse una comprensione solida di tali tecniche.

In conclusione, nonostante i LLM, in particolar modo ChatGPT, abbiano già dimostrato di avere il potenziale di rafforzare la *cyber defence* contribuendo attivamente alla creazione di *report*, al *processing* di grandi quantità di dati per identificare *bugs* ed eventuali minacce all'incolubilità dei sistemi, alla generazione di codice sicuro e alla rilevazione di veri e propri *cyber attacks* [4], questi necessitano ancora dell'integrazione con tecniche di SAST o della supervisione attenta di *security experts* per essere utilizzati efficacemente.

[1] Per una descrizione più approfondita del CWE, vedere la sezione 5.1

[2] Per una descrizione più approfondita del CVE, vedere la sezione 5.1

Capitolo 4

Tecnologie e strumenti

In questo capitolo viene fornita una spiegazione riassuntiva di ogni tecnologia e strumento di supporto utilizzati durante la fase di implementazione del progetto.

4.1 Tecnologie

4.1.1 Serverless Framework

Il *Serverless Framework* è un *framework open-source* che facilita la costruzione e la distribuzione di applicazioni *serverless*. Il *framework*, infatti, è stato inizialmente creato per facilitare lo sviluppo e il *deploy* di funzioni AWS Lambda, assieme alle risorse dell'infrastruttura AWS di cui necessitano. Esso consente di implementare architetture *event-driven* basate su funzioni, eventi e risorse definite attraverso un semplice file di configurazione YAML, in modo che sia il *framework* stesso ad occuparsi di distribuire l'ambiente necessario all'esecuzione su *provider cloud* come AWS o altri. Inoltre, *Serverless Framework* può essere esteso con *plugin* personalizzati per adattarlo a esigenze specifiche e aggiungere nuove funzionalità.

4.1.2 Node.js

Node.js è un *runtime JavaScript open-source* e multi-piattaforma che consente di eseguire codice *JavaScript* al di fuori di un *browser* sia sul lato *client* che sul lato *server*, permettendo quindi un ambiente di sviluppo unificato. Basato sul motore V8 di *Google Chrome*, *Node.js* possiede un vasto ecosistema di librerie e pacchetti reperibili tramite il Node Package Manager (npm), il che lo rende utilizzabile per realizzare progetti di ogni tipo.

4.1.3 Typescript

Typescript è un *superset* di *JavaScript*; gli elementi principali che contraddistinguono questo linguaggio di programmazione sono l'introduzione della tipizzazione statica opzionale, il supporto di concetti appartenenti al paradigma dell'Object Oriented Programming (OOP) e l'utilizzo di un compilatore (il TSC). Grazie a queste aggiunte,

Typescript migliora la produttività e la robustezza dello sviluppo *Javascript*, riducendo gli errori comunemente dovuti alla tipizzazione dinamica e facilitando il *refactoring* del codice. Inoltre, il codice *Typescript* viene compilato dal TSC in codice *Javascript*, il che lo rende compatibile con ambienti di sviluppo che fanno uso di quest'ultimo come *Node.js*.

4.1.4 AWS Bedrock

Bedrock è un servizio completamente gestito di AWS che consente di utilizzare *foundation models* ad alte prestazioni forniti da aziende *leader* nel campo dell'intelligenza artificiale, attarverso un'API unificata; Bedrock supporta anche l'integrazione e l'implementazione di modelli *custom* senza la necessità di gestire l'infrastruttura sottostante, in quanto si basa su un'architettura *serverless* ed è integrato con gli altri servizi AWS. Tra le funzionalità di Bedrock vi è anche il *Playground*, una piattaforma interattiva che consente di esplorare i diversi modelli messi a disposizione, personalizzandone la configurazione, per valutarne le prestazioni in modo agevole; essa risulta particolarmente utile per determinare il modello più adatto al proprio caso d'uso e il formato del *prompt* da utilizzare.

4.1.5 AWS Lambda

Lambda è un servizio di calcolo *serverless* che permette di eseguire codice in risposta a particolari eventi senza doversi preoccupare di gestire *server* appositi. Il codice caricato all'interno di Lambda deve essere *stateless* ed è noto come funzione Lambda; ogni funzione richiede una configurazione di base dei propri meta-dati, tra cui il nome, le risorse necessarie e il punto di ingresso. All'interno di tale configurazione si possono specificare anche gli eventi pensati per innescare in automatico l'esecuzione della funzione, come chiamate API provenienti da AWS API Gateway; ciascuna funzione viene eseguita all'interno di un *container* dedicato, il che consente a Lambda di scalare le risorse allocate all'esecuzione in base al numero di richieste (e quindi scalare carichi di lavoro espandibili nel *cloud* AWS in modo affidabile).

4.1.6 AWS Step Function

Step Function è un servizio di orchestrazione *serverless* che consente di coordinare più servizi AWS in flussi di lavoro visivi. Il flusso di lavoro o *workflow* viene definito all'interno di un semplice file di configurazione in formato Amazon States Language (ASL) (una variante del formato `JSON`), che descrive ogni stato, i suoi parametri in ingresso e in uscita e le transizioni ad altri stati e consente di implementare anche eventuali esecuzioni concorrenti di stati, logiche di ramificazione, cicli e la gestione automatica degli errori. In questo contesto, un *workflow* è paragonabile all'esecuzione di una macchina a stati dove ogni stato corrisponde ad un'operazione particolare a seconda della sua tipologia. Volendo fare alcuni esempi a scopo illustrativo: uno stato di tipo `Task` rappresenta una singola unità di lavoro svolta da un altro servizio AWS, come l'invocazione di una funzione Lambda; uno stato di tipo `Parallel`, invece, rappresenta un insieme di unità di lavoro diversificate svolte in modalità concorrente;

uno stato di tipo `Map`, infine, rappresenta un'unità di lavoro eseguita ripetutamente su ciascun elemento dell'array in input in modalità concorrente.

4.1.7 AWS Simple Storage Service (S3)

S3 è un servizio di *storage* per oggetti nel *cloud* AWS. Un oggetto è costituito da un file e dai meta-dati che lo descrivono e viene caricato all'interno di una risorsa nota come *bucket* (un contenitore di oggetti); S3 è progettato per scalare automaticamente, replicare i dati archiviati in più *data center* garantendo una disponibilità elevata, controllare gli accessi tramite Identity and Access Management (IAM), criptare i dati a fini di sicurezza e supportare il trasferimento parallelo di grandi quantità di dati per fornire prestazioni elevate.

4.1.8 AWS Systems Manager Parameter Store

Parameter Store è un servizio gestito di AWS che fornisce un archivio per memorizzare e criptare dati noti come parametri, solitamente dati di configurazione e segreti. Esso consente di memorizzare, recuperare e gestire stringhe di configurazione, *password*, *token* API e altri tipi di dati sensibili in modo sicuro e centralizzato. I parametri sono soggetti a versionamento e a un'organizzazione in gerarchie, il che ne facilita la gestione e la ricerca.

4.1.9 AWS API Gateway

API Gateway è un servizio gestito di AWS che consente di creare, pubblicare e monitorare API *RESTful* e *WebSocket* su grande scala. Risulta particolarmente utile per fare in modo che determinate applicazioni possano accedere a dati o funzionalità di vari servizi *back-end* AWS, come Lambda: come accennato in precedenza, è possibile configurare un evento in arrivo da API Gateway per innescare l'esecuzione di una funzione Lambda in AWS, utilizzando i dati specificati nel corpo della richiesta come input della funzione, se necessario. Le API *RESTful* create da API Gateway seguono il paradigma `HTTP`, consentono una comunicazione *client-server stateless* e implementano metodi `HTTP` standard come `GET`, `POST`, `PUT`, `PATCH` e `DELETE`.

4.2 Strumenti di supporto

4.2.1 Git

Git è un sistema di versionamento distribuito e *open-source* che consente a più sviluppatori di lavorare su un progetto *software* in contemporanea. Infatti, esso permette di creare *branches* a se stanti per sviluppare nuove funzionalità o correggere *bug* in un ambiente isolato, per poi unirli al *branch* principale tramite un *merge*. Git registra anche tutte le modifiche apportate al codice con un identificatore univoco rendendole tracciabili, in modo da poterle ripercorrere e analizzare con facilità.

4.2.2 AWS CodeCommit

CodeCommit è un servizio di versionamento gestito da AWS che ospita *repository* Git in modo da poterle gestire privatamente all'interno del *cloud*. Disponendo di un'interfaccia compatibile con Git, esso consente agli sviluppatori di utilizzare i comandi propri di Git per gestire le *repository* e offre, al contempo, un ambiente altamente scalabile e integrato con altri servizi AWS; infatti, gli eventi provenienti da CodeCommit vengono spesso utilizzati per scatenare flussi di lavoro *CI/CD* configurati tramite CodeBuild e CodePipeline.

4.2.3 AWS CodeBuild

CodeBuild è un servizio di integrazione continua incaricato di compilare il codice sorgente ed eseguire eventuali test, per poi produrre pacchetti *software* destinati al *deployment* (processo di *build*). Non solo CodeBuild scala automaticamente per gestire molteplici *build* in parallelo, eliminando la necessità di gestire *server* di *build*, ma esso si integra anche con la maggior parte degli altri servizi AWS come CodePipeline, S3 e CloudWatch, per una gestione completa del ciclo di vita del *software*.

4.2.4 AWS CloudWatch

CloudWatch è un servizio di monitoraggio e *logging* offerto da AWS. È progettato per raccogliere e visualizzare metriche dettagliate sulle risorse e le applicazioni in esecuzione su AWS, consentendo di monitorarne le prestazioni, configurare allerte basate su determinate metriche e analizzare i *log* di sistema. Risulta particolarmente utile soprattutto per esaminare i *log* dell'esecuzione delle funzioni Lambda per intero, in modo da comprendere le cause che sottendono eventuali errori, se presenti.

4.2.5 ESLint

ESLint è uno strumento di *linting open-source* per *JavaScript* e *TypeScript* che aiuta a individuare e correggere problemi e a mantenere uno stile di codice coerente. Esso analizza il codice sorgente staticamente, identificando potenziali errori, pratiche di codifica non ottimali e violazioni di convenzioni stilistiche configurabili attraverso file di configurazione personalizzati. È anche facilmente integrabile con vari editor di testo e ambienti di sviluppo integrato.

4.2.6 Visual Studio Code

Visual Studio Code (VS Code) è un editor di codice *open-source* progettato per supportare lo sviluppo di applicazioni in diversi linguaggi di programmazione, grazie ad un sistema di estensioni molto flessibile che consente di adattare l'ambiente di sviluppo alle proprie esigenze. Tra le varie funzionalità offerte da VS Code spiccano il *debugging* integrato, il controllo di versione tramite Git e l'*autocomplete* del codice.

Capitolo 5

Progettazione

In questo capitolo viene descritta la fase di progettazione del progetto, che si articola in gestione delle vulnerabilità, individuazione del sistema di *scoring*, scelta del modello di *Generative AI* da utilizzare, *prompt engineering* e infine la definizione dell'architettura *cloud* del servizio.

5.1 Gestione delle vulnerabilità

La fase di progettazione è iniziata con lo studio del contesto applicativo e quindi la strutturazione della *code review* eseguita dal modello di Generative AI. Come accennato nell'introduzione, lo scopo della revisione è identificare eventuali vulnerabilità che compromettono la sicurezza del codice sorgente analizzato o *threat identification*; in tale contesto, spesso si fa riferimento a cataloghi ben noti e comunemente utilizzati nelle revisioni di codice, che aiutano a definire e a standardizzare la terminologia da adottare per descrivere minacce pertinenti a diverse aree della *cybersecurity*.

Due di questi sono il CVE [23], che cataloga vulnerabilità scoperte nel contesto di applicazioni concrete, e il CWE [25], che cataloga le debolezze che appaiono di frequente all'interno di un prodotto *software* o *hardware*; entrambi vengono utilizzati per arricchire la *knowledge base* della CTI. Le vulnerabilità elencate all'interno del CVE coincidono con quelle pubblicate sul National Vulnerability Database (NVD) [28] mantenuto da MITRE: ogni vulnerabilità presenta un identificativo univoco, una descrizione e gli `URL` delle risorse dove è stata pubblicata in origine. D'altra parte, le debolezze elencate all'interno del CWE sono organizzate in una struttura alberata e riportano un identificativo univoco, una descrizione, *code snippets* esemplificativi e raccomandazioni per la risoluzione.

Dunque, CVE e CWE godono di una relazione di interdipendenza: mentre le CVE si riferiscono a vulnerabilità documentate nel *software*, le CWE descrivono le debolezze del codice che possono generare tali vulnerabilità e per questo una singola vulnerabilità nel CVE può essere associata a una o più debolezze nel CWE che hanno portato alla sua manifestazione (*root-cause mapping*); non sorprende, dunque, che ogni vulnerabilità nel NVD sia riportata assieme alle debolezze CWE associate, oltre che al proprio Common Vulnerability Scoring System (CVSS) *Score*. Tuttavia, i due

sistemi di catalogazione sono anche complementari tra loro, in quanto il CVE facilita l'identificazione e la gestione delle vulnerabilità conosciute, mentre il CWE aiuta a prevenire la comparsa di nuove vulnerabilità attraverso una comprensione migliore delle debolezze del codice.

Lo studio del contesto ha portato alla conclusione che, nell'ottica di una *code review* condotta da un modello di Generative AI, utilizzare il CWE è preferibile rispetto al CVE; una prima motivazione del perché risiede nell'intenzione di adottare un approccio preventivo per individuare e correggere eventuali problemi durante lo sviluppo del *software*, prima che questi si trasformino in vere e proprie vulnerabilità. Inoltre, mentre il CVE si riferisce a vulnerabilità specifiche già identificate e documentate, il CWE copre una una vasta gamma di debolezze intrinseche del codice, e si presta meglio ad una revisione dalla copertura più ampia possibile. Il CWE riporta anche debolezze che si applicano a qualunque tipo di codice, indipendentemente dal fatto che sia già stato distribuito o meno; le vulnerabilità nel CVE, invece, sono inevitabilmente inerenti a *software* rilasciato.

Una volta identificato nel CWE il catalogo su cui fare riferimento per condurre la revisione del codice, il passo successivo è consistito nel definire le debolezze CWE da incorporare nella revisione, con lo scopo duplice di fornire una copertura sufficientemente ampia e, al contempo, di non rendere la revisione troppo onerosa in termini di dispendio di risorse e tempo di esecuzione. Un criterio ragionevole per operare la selezione di un *subset* utile al caso d'uso di interesse è l'utilizzo delle CWE pertinenti alle categorie OWASP Top Ten 2021 [30]; si tratta, infatti, di un elenco delle vulnerabilità più prevalenti all'interno delle applicazioni *web*, aggiornato periodicamente dall'Open Web Application Security Project (OWASP) [29]. Le macro-categorie presentate nel 2021 sono mostrate nella seguente tabella:

Categoria	Nome	Descrizione
A01:2021	Broken Access Control	Debolezze che permettono agli utenti di accedere a funzionalità o dati pur non avendo l'autorizzazione necessaria
A02:2021	Cryptographic Failures	Problemi riguardanti la protezione dei dati tramite tecniche crittografiche
A03:2021	Injection	Debolezze che permettono l'invio di comandi non validi a un interprete
A04:2021	Insecure Design	Carenze progettuali che portano a vulnerabilità di sicurezza intrinseche
A05:2021	Security Misconfiguration	Configurazioni errate o insicure di applicazioni e *server*
A06:2021	Vulnerable and Outdated Components	Utilizzo di componenti software con vulnerabilità note o non aggiornate

Continua nella pagina successiva

Tabella 5.1 – continua dalla pagina precedente

Categoria	Nome	Descrizione
A07:2021	Identification and Authentication Failures	Debolezze nei meccanismi di autenticazione e gestione delle sessioni
A08:2021	Software and Data Integrity Failures	Problemi relativi all'integrità del *software* e dei dati
A09:2021	Security Logging and Monitoring Failures	Carenze nel *logging* e monitoraggio della sicurezza, che ostacolano la rilevazione tempestiva degli attacchi
A10:2021	Server-Side Request Forgery (SSRF)	Debolezze che permettono agli aggressori di indurre il *server* a effettuare richieste a un dominio non previsto

Tabella 5.1: Categorie di rischi di sicurezza OWASP Top 10 2021

Di conseguenza, è stato sufficiente individuare le CWE raggruppate all'interno di tali categorie per integrarle efficacemente nel processo di revisione del codice; nella versione 4.14 del catalogo CWE [22], le singole debolezze vengono raggruppate anche secondo particolari categorie di appartenenza, tra cui le OWASP Top Ten 2021 (categorie 1345-1356). Aggregando le CWE elencate sotto tali categorie, si ottiene un *subset* di **175** debolezze, riportate per intero nella sezione A.3.

5.2 Sistema di scoring

Successivamente, è stato necessario stabilire il criterio con cui generare uno *score* cumulativo che quantificasse il livello di sicurezza del codice analizzato; ispirandosi ai risultati di *code review* vere e proprie forniti dall'azienda, si è scelto di fare affidamento su un *framework* standardizzato e ampiamente utilizzato per valutare la gravità delle vulnerabilità rilevate, ovvero il CVSS 3.1 [24]. Il CVSS è composto da tre gruppi di metriche: *Base Metrics*, *Temporal Metrics* ed *Environmental Metrics*.

Le *Base Metrics* valutano le caratteristiche intrinseche di una vulnerabilità e producono un punteggio base o *Base Score* che riflette la gravità di una determinata vulnerabilità in assenza di altri fattori, temporali o ambientali. D'altra parte le *Temporal Metrics* o *Environmental Metrics* non vengono utilizzate altrettanto spesso in revisioni automatizzate poichè rappresentano caratteristiche soggette a variazioni nel tempo o specifiche all'ambiente di esecuzione: nel contesto in cui si utilizza un modello di Generative AI che valuta esclusivamente il codice di un applicativo, tali caratteristiche non sono realisticamente valutabili in quanto spesso non sono direttamente deducibili dal sorgente (richiedono informazioni contestuali esterne). Di conseguenza, si è scelto di utilizzare esclusivamente le *Base Metrics*, sia per semplificare il processo di valutazione, che per ottenere valutazioni più precise e realistiche da parte del modello.

Il *Base Score* del CVSS 3.1 varia da 0 a 10 e viene calcolato utilizzando una formula che combina i valori assegnati alle metriche di base; in base allo *score* è

Metrica	Descrizione	Valori Possibili
Attack Vector (AV)	Indica il canale attraverso cui la vulnerabilità può essere sfruttata	- Network (N) - Adjacent (A) - Local (L) - Physical (P)
Attack Complexity (AC)	Valuta quanto è difficile sfruttare la vulnerabilità	- Low (L) - High (H)
Privileges Required (PR)	Indica il livello di privilegi necessari per sfruttare la vulnerabilità	- None (N) - Low (L) - High (H)
User Interaction (UI)	Indica se è necessaria l'interazione dell'utente per sfruttare la vulnerabilità	- None (N) - Required (R)
Scope (S)	Valuta se la vulnerabilità impatta solo il componente vulnerabile o anche altro	- Unchanged (U) - Changed (C)
Confidentiality Impact (C)	Valuta l'impatto della vulnerabilità sulla confidenzialità dei dati	- None (N) - Low (L) - High (H)
Integrity Impact (I)	Valuta l'impatto della vulnerabilità sull'integrità dei dati	- None (N) - Low (L) - High (H)
Availability Impact (A)	Valuta l'impatto della vulnerabilità sulla disponibilità del sistema	- None (N) - Low (L) - High (H)

Tabella 5.2: Base Metrics del CVSS 3.1

possibile determinare anche la *severity* della vulnerabilità, che fornisce semplicemente una valutazione qualitativa del livello di sicurezza.

Severity	CVSS Score
None	0.0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

Tabella 5.3: Severity e CVSS Scores associati

Una rappresentazione alternativa dello *score* è costituita dal CVSS Vector, una stringa di caratteri che indica a sua volta i valori delle metriche utilizzati per calcolare il punteggio di una vulnerabilità. Il vettore è nel formato:

```
CVSS:3.1/AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H
```

Ogni componente è separato da una barra (/) e ogni metrica è rappresentata da una coppia di valori separati da due punti (:), dove il primo valore coincide con l'abbreviazione della metrica e il secondo con il valore che le è stato assegnato. Nel

perseguire l'obiettivo di ottenere uno *score* numerico cumulativo, si è scelto di utilizzare entrambe le rappresentazioni del punteggio CVSS, in particolare: **(1)** il vettore viene utilizzato nell'interazione con il modello in modo che questo sia naturalmente portato a valutare le metriche di base prendendo in considerazione una vulnerabilità specifica e il codice rilevante; **(2)** il vettore generato dal modello viene convertito nel punteggio numerico corrispondente. Tale approccio è giustificato dal fatto che il vettore, esplicitando i valori delle metriche, rende facilmente comprensibile come è stato derivato il punteggio e assicura un maggior livello di trasparenza, in modo che sia più semplice anche verificare l'accuratezza delle valutazioni. Inoltre, l'uso di un *prompt* strutturato per istruire il modello sulla generazione del vettore è più indicato di un *prompt* più libero che richiede direttamente il punteggio, al fine di ridurre la soggettività che potrebbe influenzare la generazione di quest'ultimo e ridurre la probabilità di errori significativi. Infatti, generare direttamente uno *score* numerico aumenta il rischio di ottenere una valutazione errata, soprattutto se il modello non prende in considerazione tutte le variabili coinvolte.

Per quanto riguarda lo *score* cumulativo, questo è semplicemente lo *score* più elevato tra quelli assegnati alle varie vulnerabilità presenti nel codice, nel rispetto del principio del *weakest link* applicato alla *cybersecurity*: "The security of a system is equivalent to the security of its less secure component", ossia l'idea che la sicurezza complessiva di un sistema può essere compromessa se una qualsiasi sua componente è vulnerabile.

5.3 Selezione del modello

La progettazione si è poi orientata sulla scelta del modello di Generative AI offerto da Bedrock da utilizzare per condurre la *code review*; la *console* di Bedrock mette a disposizione una funzionalità di nome *Playground*, che consente di interloquire con i modelli in modo agevole ed è quindi risultata utile per operare uno *screening* iniziale, volto a stabilire quale modello potesse essere più adatto al caso d'uso descritto prima ancora di cominciare la fase di implementazione vera e propria. Più in dettaglio, si è deciso di consultare i vari modelli a disposizione in merito alla presenza o meno di una determinata CWE nel *code snippet* fornito utilizzando il seguente *prompt*:

```
Is CWE{{cweId}} or {{cweName}} present within the code? If yes, answer following
this format: "CWE{{cweId}}: present",
followed by "Affected area: " and the affected code,
followed by "Explanation" and an explanation of why it's affected,
followed by "Resolution:" and a possible resolution tactique and
suggestions. If no, answer following this format: "CWE{{cweId}}: not present";
do NOT add any explanations.

{{sourceCode}}
```

Il frammento di codice fornito di volta in volta varia a seconda della CWE analizzata, in quanto viene estratto appositamente dalla pagina dedicata alla descrizione

di tale CWE dal MITRE; infatti, il sito dedicato al CWE [25] consente di cercare una qualsiasi debolezza all'interno del catalogo utilizzando il suo id univoco e, nella maggioranza dei casi, anche di visualizzare uno o più esempi di codice affetto. La disponibilità di codice "classificato" secondo le debolezze presenti al suo interno ha reso semplice valutare le risposte dei modelli ai *prompt* adoperati per eseguire un controllo di varie CWE tra le 175 selezionate: l'esito della valutazione è positivo se il modello determina con accuratezza qualora la debolezza in esame sia presente (metodo o linea di riferimento, secondo quanto riportato dal CWE stesso) o meno all'interno del codice (*True Positive* o *True Negative*); neutrale se dichiara che una particolare debolezza è presente all'interno del codice, nonostante questa non sia tra quelle indicate dal CWE (*False Positive*)[1]; negativo se il modello non riesce ad individuare la debolezza indicata quando essa è effettivamente presente (*False Negative*). Innanzitutto si è voluto determinare quali modelli esibivano una *performance* migliore tra quelli di un determinato *provider*; da questa analisi preliminare sono emersi quattro modelli in particolare: Titan Text G1 - Premier di *Amazon*, Claude Instant 1.2 di *Anthropic*, Llama 3 70B Instruct di *Meta* e Mistral Large di *Mistral AI*. In un secondo momento sono state testate 58/175 CWE su questi modelli e il modello che è risultato più adatto a condurre revisioni di codice orientate al controllo di CWE, secondo l'esito della metodologia descritta (riportato nella sezione A.1), è **Mistral Large**.

La decisione di utilizzare Mistral Large nell'implementazione del servizio è dovuta non solo al fatto che si tratta del modello che ha rilevato correttamente il maggior numero di CWE, ma anche al fatto che si è distinto dagli altri nel corso dello *screening* per la sua capacità di rispettare più fedelmente il formato della risposta previsto dal *prompt* e per la sua coerenza nel fornire risposte simili tra un'invocazione e l'altra. Inoltre, è anche il modello che ha avuto la tendenza minore ad "allucinare", o generare risposte che comprendessero informazioni errate, inesistenti o non basate sui dati reali forniti. In ambito AI, il termine allucinazione si riferisce a quando un modello produce contenuti che non sono basati su fatti o dati reali, ma che sembrano plausibili. Tentare di ridurre le allucinazioni è cruciale per mantenere l'accuratezza e l'affidabilità delle risposte fornite dal modello.

5.4 Prompt engineering

Successivamente, si è voluto definire con accuratezza il *prompt* da utilizzare nelle invocazioni di Mistral Large, ispirandosi alle pratiche del cosiddetto *prompt engineering*; il *prompt engineering* consiste nel progettare e ottimizzare le istruzioni fornite a un modello AI con lo scopo di ottenere risposte più accurate e pertinenti, ed è particolarmente importante perchè la qualità delle risposte può variare notevolmente a seconda di come viene formulato il *prompt*. L'approccio utilizzato è stato fonda-

[1] Neutrale perché non sempre il CWE riporta tutte le debolezze presenti all'interno del frammento di codice esemplificativo; c'è quindi la possibilità che il modello ne identifichi di nuove o semplicemente di non dichiarate esplicitamente

mentalmente affine ad un *trial and error*, ed è consistito nell'eseguire svariati test con *prompt* leggermente diversi gli uni dagli altri (per uso di linguaggio specifico, struttura della domanda o contesto fornito) per imparare come rispondeva il modello e apportare cambiamenti di conseguenza. Le risposte generate sono state confrontate tra loro per capire quale formulazione del *prompt* consentisse al modello di rispondere in modo più completo e accurato nel contesto della revisione di codice ed è stato selezionato il *prompt* più efficacie nell'ottenere tale scopo (un esempio del suo utilizzo è mostrato nella sezione A.2):

```
Analyze the provided source code and determine the presence of each of the following
CWEs. For each CWE, follow the specified format strictly:

CWE#: present
File: [Provide the file path from the code comments]
Area: [Describe the code area affected by the CWE, highlighting
specific code snippets if possible]
Explanation: [Give a detailed explanation as to why the code is
affected by the CWE]
Resolution: [Suggest a resolution tactic and possible
improvements to the code]

or

CWE#: not present

DO NOT add any extra explanations or deviate from this format. Each CWE should be
answered individually and in the order listed.
Use the following list of CWEs for your analysis:

{{cweList}}

{{sourceCode}}
```

Il *prompt* riportato sopra è quello utilizzato per controllare la presenza o meno di un *set* di CWE all'interno del *code snippet* fornito al modello; lo stesso tipo di analisi e ottimizzazione è stato compiuto per individuare il *prompt* più adatto per guidare il modello nella generazione di un CVSS Vector per ciascuna CWE precedentemente rilevata nel codice in esame e il risultato è il seguente:

```
Generate one CVSS v3.1 Vector for each of the following CWEs, given that they were
detected within the provided source code as shown.
Follow the specified format strictly:

CWE#
Vector: [CVSS3.1 vector]
AV: [Attack Vector explanation]
AC: [Attack Complexity explanation]
PR: [Privileges Required explanation]
UI: [User Interaction explanation]
S: [Scope explanation]
C: [Confidentiality Impact explanation]
```

```
I: [Integrity Impact explanation]
A: [Availability Impact explanation]

Each CWE should be evaluated with a vector. Do NOT add any extra explanations or
deviate from this format.

{{cweList}}

{{sourceCode}}
```

5.5 Autenticazione all'API di GitHub

Nel corso della progettazione ci si è poi soffermati sulla modalità più adatta per fare in modo che il servizio potesse autenticarsi all'API di GitHub tramite un *token* della tipologia indicata per effettuare operazioni di lettura e scrittura all'interno della *repository*, generato in automatico; infatti, volendo evitare uno scenario in cui è richiesto l'intervento manuale dei *developers* per generare un Personal Access Token (PAT) su GitHub, sono state esplorate fondamentalmente due alternative per generare un *token* con i permessi necessari automaticamente.

La prima è l'utilizzo di un *token* generato automaticamente da GitHub stesso all'inizio di ogni *workflow* pertinente ad una GitHub Action nel momento in cui viene innescata, salvato nei *secrets* col nome di `GITHUB_TOKEN` [19]; tuttavia, avendo una durata limitata all'esecuzione dell'Action, questo *token* sarebbe stato utilizzabile nella Lambda incaricata di reperire il sorgente da GitHub inizialmente, prima di lanciare il servizio, ma non al termine della sua esecuzione (per inserire il *report* nella *repository*).

La seconda alternativa, e quella effettivamente adottata, consiste nell'utilizzare un Installation Access Token (IAT), che è un *token* utilizzato per consentire l'interazione automatizzata tramite API tra servizi esterni e le risorse di GitHub [18]. Infatti, esso è specificamente legato ad una determinata installazione di una GitHub App su un *account* o un'organizzazione GitHub; l'utilizzo di un IAT garantisce che le interazioni siano limitate al contesto e alle autorizzazioni definite per l'installazione della GitHub App, fornendo un meccanismo sicuro per l'automazione delle attività, senza necessità di autenticazioni manuali ripetute. Pertanto, configurando il servizio (il VAS) come una GitHub App e installandolo sulla *repository* da monitorare, è possibile utilizzare le credenziali dell'App per creare un IAT (della durata *standard* di 1 ora) che consenta alle Lambda rilevanti di eseguire l'autenticazione e sfruttare l'API di GitHub per compiere le dovute operazioni; infatti, i permessi dell'IAT così generato sono gli stessi di quelli assegnati alla GitHub App nel momento in cui viene creata e/o installata. Perciò, purchè questi siano

- `Read access to metadata and pull requests`
- `Read and write access to code`

il *token* sarà adatto al recupero iniziale del codice sorgente e all'inserimento finale del *report* generato dal servizio.

5.6 Architettura cloud del servizio

Nell'ultima fase della progettazione ci si è concentrati sulla definizione dell'architettura *cloud* del servizio, in modo da definire il flusso di esecuzione in dettaglio a priori e agevolare l'implementazione in un secondo momento.

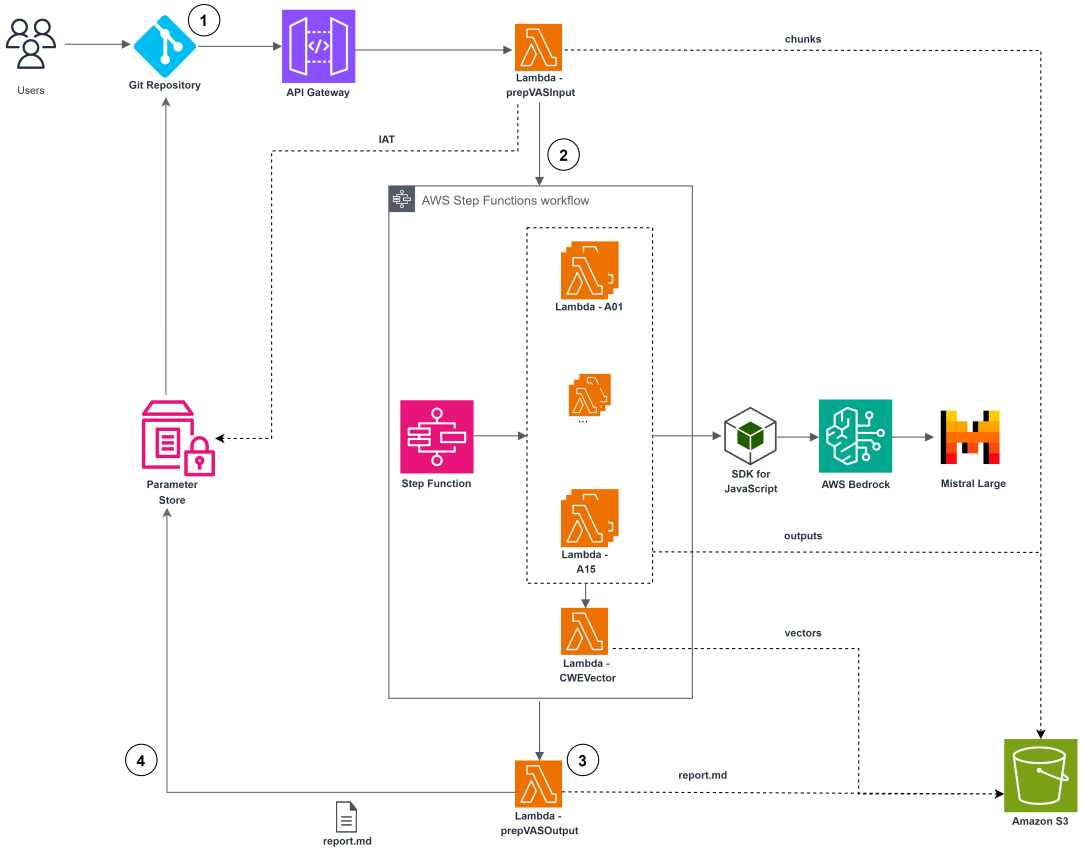


Figura 5.1: Architettura cloud del servizio

Come evidenziato dalla figura 5.1, l'architettura risultante è costituita da varie componenti (prevalentemente servizi AWS) che orchestrano il seguente flusso di operazioni:

1. I *developers* effettuano operazioni su una *repository* GitHub, sulla quale il servizio è installato come GitHub App, che innescano le GitHub Actions configurate per scatenare l'esecuzione del servizio su AWS
2. Le GitHub Actions effettuano una chiamata API agli *endpoints* configurati su API Gateway, in modo da innescare la funzione Lambda che fa partire il servizio vero e proprio (`prepVASInput`)
3. La Lambda elabora i dati in ingresso (incluso il codice sorgente prelevato dalla *repository*) e innesca la Step Function incaricata di condurre la revisione del codice
4. La Step Function organizza un flusso di lavoro che coinvolge più Lambda per controllare la presenza o meno delle 175 CWE nel codice sorgente e generare

un CVSS Vector, e quindi un CVSS 3.1 Score, per ogni CWE rilevata; questo avviene tramite Lambda specializzate che invocano Mistral Large su Bedrock utilizzando il Software Development Kit (SDK) per *JavaScript*

5. Al termine dell'esecuzione del servizio viene invocata la Lambda incaricata di gestire i dati in uscita aggregandoli all'interno di un unico file (`report.md`) per poi caricarlo all'interno della *repository* di partenza (`prepVASOutput`)

Come discusso più in dettaglio nella sezione 5.6.3.2, il servizio si serve anche di un S3 *bucket* per conservare uno storico delle esecuzioni e del Parameter Store per memorizzare il *token* di autenticazione necessario per poter usufruire dell'API di GitHub, sia all'inizio che al termine dell'esecuzione.

5.6.1 VAS: flusso di esecuzione in ingresso

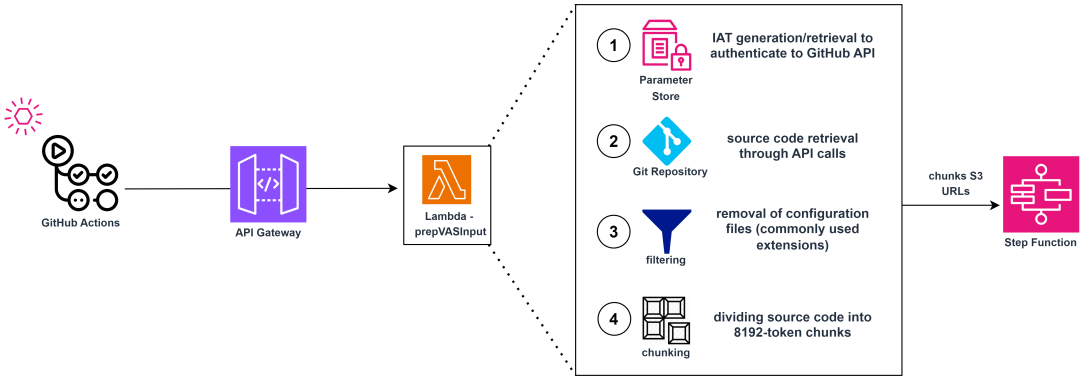


Figura 5.2: Flusso di esecuzione in ingresso

5.6.1.1 GitHub Actions

Il servizio dispone di due GitHub Actions, progettate per inviare una richiesta API a API Gateway in seguito a *trigger* particolari e inserire, quindi, il servizio all'interno della catena *CI/CD* rendendone l'esecuzione completamente automatica.

La prima, `Analyse Default Branch`, è stata pensata per essere scatenata quando si verifica un *push* sul *default branch*, sia esso `main` o `develop`, in modo da analizzare l'intero contenuto della *repository* ed individuare eventuali vulnerabilità; questo può risultare particolarmente utile nel momento in cui si chiude una Pull Request (PR) il cui *target branch* coincide con il *default branch* (un caso d'uso piuttosto comune) e si va ad effettuare il *merge* del *source branch*, introducendo cambiamenti di portata più o meno significativa all'interno del *default branch*.

```
1 name: Analyse Default Branch
2 on:
3   push:
4     branches:
5       - main
```

Listato 5.1: Trigger di Analyse Default Branch

La seconda, `Analyse Pull Request`, è stata pensata per essere scatenata quando una PR viene aperta inizialmente (`opened`) e quando viene aggiornata con *commit* sul proprio *source branch* (`synchronize`), in modo da analizzare solamente i file con stato `added` o `modified` al suo interno ed individuare eventuali vulnerabilità; questo può risultare particolarmente utile nel momento in cui si analizza il contenuto della PR con un *commit* sul *source branch* e si rilevano determinate vulnerabilità prima di chiudere la PR, riversandone il contenuto nel *target branch*.

```
name: Analyse Pull Request
on:
  pull_request:
    types: [opened, synchronize]
```

Listato 5.2: Triggers di Analyse Pull Request

L'uso delle GitHub Actions per inserire il servizio nella *CI/CD* non costituisce l'unica alternativa, in quanto è possibile ottenere lo stesso risultato sfruttando le funzionalità di AWS CodePipeline; infatti, ammesso che la *repository* su GitHub abbia una *repository* corrispondente su CodeCommit, è possibile istanziare una CodePipeline che viene innescata in seguito agli stessi *trigger* descritti sopra ma rilevati da CodeCommit invece che da GitHub. Volendo, la CodePipeline è in grado di utilizzare un comando all'interno di CodeBuild per eseguire una chiamata API ad API Gateway, scatenando il servizio. L'unica differenza sostanziale tra i due approcci, e il motivo principale per cui si è scelto di utilizzare le GitHub Actions, consiste nel fatto che un'istanza di CodePipeline deve essere configurata in partenza specificando il nome del *branch* della *repository* CodeCommit i cui eventi ne innescano l'esecuzione; questo rende possibile configurarla sul *default branch* con le stesse funzionalità descritte sopra, ma rende difficile fare lo stesso sul *source branch* di una PR nel momento in cui viene aperta, dato che questo non è noto a priori.

5.6.1.2 API Gateway

API Gateway fornisce gli *endpoints* necessari affinchè le GitHub Actions possano provocare l'esecuzione della Lambda apposita e dell'intero servizio; in particolare, si ha:

- `https://1y393x0s06.execute-api.us-east-1.amazonaws.com/dev/prepVASInput`: consente a `Analyse Default Branch` di chiamare la Lambda `prepVASInput`
- `https://1y393x0s06.execute-api.us-east-1.amazonaws.com/dev/prepVASInputForPR`: consente a `Analyse Pull Request` di chiamare la Lambda `prepVASInputForPR`

Entrambi sono configurati per ricevere chiamate di tipo `POST` purchè nell'*header* venga inclusa un'API key generata automaticamente dal *Serverless Framework* e salvata su API Gateway.

5.6.1.3 Lambda: prepVASInput

La Lambda `prepVASInput` dispone l'occorrente affinchè la Step Function possa essere avviata correttamente eseguendo una serie di passaggi in modo sequenziale.

Innanzitutto, si interfaccia con il Parameter Store per determinare se esiste un IAT associato alla *repository* in esame creato precedentemente (e non ancora scaduto), in modo da poterlo riutilizzare nel corso dell'esecuzione corrente; se questo non esiste (o è scaduto), ne crea uno nuovo e lo inserisce all'interno del Parameter Store. Gli IAT hanno una durata di un'ora dal momento in cui vengono generati, perciò questa scelta è stata fatta per poterli riutilizzare quanto più possibile tra un'esecuzione e l'altra sulla stessa *repository*, oltre che al termine dell'esecuzione corrente. In secondo luogo, la Lambda utilizza l'API di GitHub [26] per prelevare il sorgente dalla *repository*; in particolare, le tipologie di chiamate effettuate sono tre, a seconda dell'operazione svolta:

- `GET /repos/owner/repo/git/trees/treeSha`: consente di ottenere un elenco dei percorsi di tutti i file presenti all'interno di un particolare *branch* della *repository*, che viene esplorata ricorsivamente grazie al parametro `recursive` nel *body* della richiesta

```
{
  owner: input.owner,
  repo: input.repositoryName,
  tree_sha: input.branchName,
  recursive: 'true'
}
```

- `GET /repos/owner/repo/pulls/pullNumber/files`: consente di ottenere un elenco dei percorsi di tutti i file presenti all'interno di una particolare PR della *repository*, purchè il *body* della richiesta contenga

```
{
  owner: input.owner,
  repo: input.repositoryName,
  pull_number: input.pullRequestId
}
```

- `GET /repos/owner/repo/contents/path`: consente di ottenere il codice sorgente contenuto in un file all'interno di un particolare *branch* della *repository*, purchè il *body* della richesta contenga

```
{
  owner: input.owner,
  repo: input.repositoryName,
  path: filePath,
  ref: input.branchName
}
```

Prima di prelevare il sorgente, la Lambda rimuove le cartelle aventi nomi e i file aventi estensioni tipicamente utilizzate per impostare le configurazioni del progetto, dato che non sono rilevanti ai fini della revisione condotta dal servizio:

```
'node_modules', '.git', 'build', 'dist', 'out', 'coverage', 'venv',
'__pycache__', 'env', 'lib', '.min.js', '.min.css', '.log', '.md',
'.json', '.yaml', '.yml', '.lock', '.txt', '.xml', '.iml', '.class',
'.db', '.*'
```

Infine, la Lambda utilizza il *tokenizer* di Mistral Large per dividere il sorgente in *chunks* di **8,192** *tokens*, in modo da assicurarsi di non superare la massima dimensione dell'*input window* del modello nel costruire e inviare i *prompt*. I *chunks* vengono caricati all'interno di S3 in modo da inviare solamente i loro `URL` alla Step Function, che li utilizza per scaricare il codice vero e proprio in un secondo momento; questo perchè il *payload* che viene passato tra stati distinti all'interno della Step Function può essere al massimo di 256KB, perciò passare semplicemente gli `URL` al posto del sorgente costituisce un approccio più prudente per evitare di superare tale limite.

5.6.2 VAS: flusso di esecuzione in elaborazione

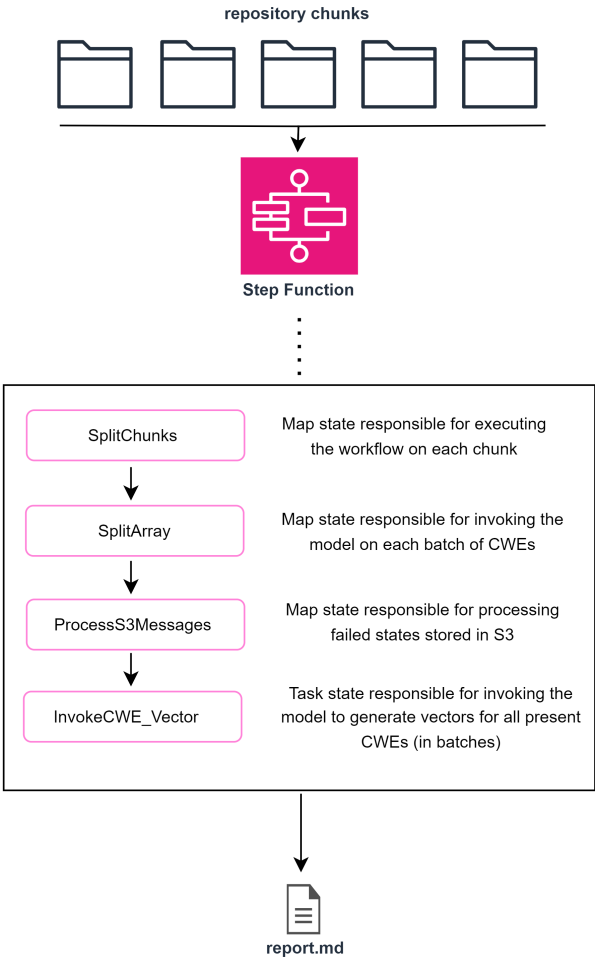


Figura 5.3: Flusso di esecuzione in elaborazione

5.6.2.1 Step Function

La Step Function costituisce il nucleo del servizio in quanto spetta ad essa orchestrare l'esecuzione delle funzioni Lambda a sua disposizione per condurre la *code review* vera e propria, invocando Mistral Large. L'utilizzo di AWS Step Function in questo contesto è giustificato principalmente dalla natura del *workflow* progettato per

strutturare la revisione in modo da controllare le 175 CWE agilmente, senza incorrere in un numero ingestibile di errori di *throttling*.

Infatti, poichè la massima quota di Bedrock per Mistral Large [31] è di **400** richieste/minuto e di 300,000 *tokens*/minuto, è stato necessario gestire le invocazioni in maniera accorta per evitare di sorpassare tale limite, per quanto possibile; allo stesso tempo, tuttavia, è stato tenuto in considerazione anche il tempo di esecuzione e si è tentato di rendere il *workflow* quanto più efficiente per evitare che l'analisi del contenuto di *repository* di grandi dimensioni impiegasse una quantità di tempo irragionevole. Questo soprattutto perchè le singole invocazioni dirette al modello impiegano almeno 20-30 secondi per ricevere risposta (ammesso che il modello risponda nei tempi prestabiliti e nel formato specificato dal *prompt*), perciò eseguirle sequenzialmente era del tutto fuori questione.

Di conseguenza, si è deciso di utilizzare una Step Function per eseguire determinate operazioni in modo concorrente (tramite stati di tipo `Map`). AWS Step Function non solo consente di eseguire determinate funzioni Lambda in un ordine prestabilito, ma anche di coordinare componenti multipli in parallelo, di definire strategie di *retry* e percorsi alternativi per gestire gli errori in modo efficacie e di tracciare lo stato di ogni esecuzione in dettaglio, utilizzando strumenti di monitoraggio integrati nella *console* dedicata; si tratta dunque del servizio più adatto per costruire un *workflow* complesso come quello delineato, che sarebbe altrimenti estremamente difficile da implementare tentando di orchestrare le Lambda manualmente. In altre parole, l'implementazione manuale della logica di orchestrazione è sconsigliata in questo contesto in quanto aumenterebbe significativamente la quantità di codice da mantenere, con conseguente aumento del *technical debt* del progetto e del rischio di introdurre vulnerabilità o *bug* veri e propri.

Il flusso di lavoro risultante fa uso di alcuni stati particolarmente importanti, brevemente delineati in seguito:

- **SplitChunks**: uno stato di tipo `Map` che riceve in input gli `URL` S3 dei *chunks* del codice sorgente ed esegue l'intero *workflow* su ciascuno di essi in modalità concorrente, processando al massimo 3 *chunks* contemporaneamente grazie all'uso del parametro `MaxConcurrency`

```
{
  SplitChunks:
  {
    Type: 'Map',
    ItemsPath: '$.chunks',
    MaxConcurrency: 3
  }
}
```

Listato 5.3: Impostazione del limite di concorrenza in SplitChunks

- **SplitArray**: uno stato di tipo `Map` che riceve in input 15 *batches* numerati di 12 CWE ciascuno ed invoca Mistral Large per il controllo delle CWE

in ciascun *batch* in modalità concorrente, processando al massimo 3 *batches* contemporaneamente

- **ProcessS3Messages**: uno stato di tipo `Map` che riceve in input i dettagli delle invocazioni del modello fallite in precedenza in modo da processarle in un secondo momento, eseguendole sempre in modalità concorrente e processando al massimo 5 *batches* di CWE contemporaneamente
- **InvokeCWEVector**: uno stato di tipo `Task` che riceve in input tutte le CWE rilevate all'interno di un determinato *chunk* e invoca il modello per generare altrettanti CVSS Vector, processando *batches* di 10 CWE ciascuno in modalità concorrente, senza limiti di concorrenza

I limiti di concorrenza imposti possono sembrare restrittivi a prima vista, ma sono necessari per gestire le invocazioni in modo efficiente, soprattutto nell'ottica in cui vengono lanciate più esecuzioni della Step Function in parallelo; inoltre, come spiegato più approfonditamente nella sezione 6.2.3.2, nel momento in cui il modello non risponde in tempo utile o seguendo il formato corretto, questo viene invocato nuovamente N volte (N = numero finito), a seconda della natura dell'invocazione e della qualità delle risposte fornite. Questa scelta progettuale rende difficile quantificare con esattezza il numero di richieste effettuate nel corso di una particolare esecuzione della Step Function, il che ha comportato la necessità di fare un lavoro di *fine-tuning* dei parametri menzionati e di *testing* di *repository* di vario tipo per determinare la configurazione più flessibile.

5.6.3 VAS: flusso di esecuzione in uscita

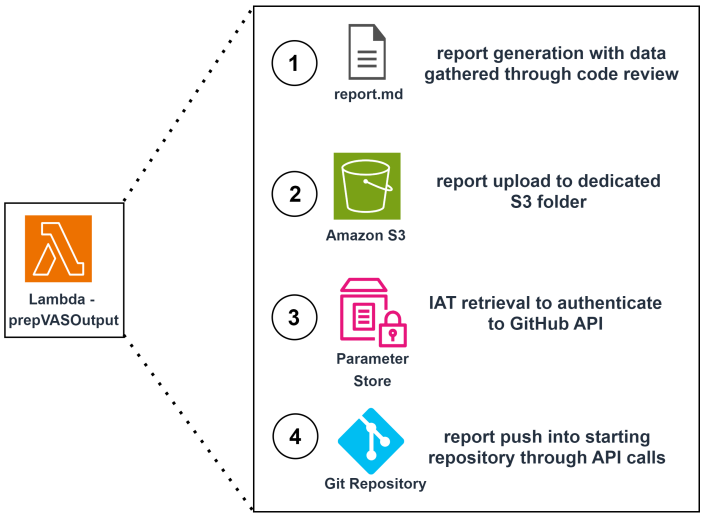


Figura 5.4: Flusso di esecuzione in uscita

5.6.3.1 Lambda: prepVASOutput

La Lambda `prepVASOutput` viene richiamata dall'ultimo stato della Step Function ed è responsabile della generazione del *report* e del suo inserimento all'interno della *repository* GitHub di partenza. Innanzitutto, il *report* viene costruito includendo le

CWE rilevate dal modello su tutti i *chunks* del codice sorgente in ingresso: ogni CWE viene riportata con tutte le informazioni contestuali a disposizione, ossia id e nome ufficiale, il file in cui è stata identificata, CVSS Vector (con spiegazioni del valore di ogni metrica) e CVSS 3.1 Score associato, una spiegazione della rilevazione effettuata dal modello, il frammento di codice affetto e tattiche di risoluzione.

EVIDENCE: 1	
CWE	538, Insertion of Sensitive Information into Externally-Accessible File or Directory
FILE	`src/apps/admin-users/services/admin-users.service.ts`
GRAVITY	medium, CVSS:3.1/AV:N/AC:L/PR:L/UI:N/S:U/C:L/I:N/A:N
Metric	**Explanation**
Attack Vector (AV)	Network - The attack can be performed over the network.
Attack Complexity (AC)	Low - The attack requires a low level of skill and complexity.
Privileges Required (PR)	Low - The attack requires low privileges.
User Interaction (UI)	None - The attack does not require user interaction.
Scope (S)	Unchanged - The scope of the vulnerability is unchanged.
Confidentiality Impact (C)	Low - The confidentiality impact is low, as only sensitive information is exposed.
Integrity Impact (I)	None - The integrity impact is none, as no data is modified.
Availability Impact (A)	None - The availability impact is none, as no data is lost or corrupted.
Score	**4.3**

Tabella 5.4: Esempio di vulnerabilità rilevata nel sorgente e riportata nel report

Le CWE sono ordinate in base al CVSS 3.1 Score in ordine decrescente, in modo da visualizzare subito le debolezze critiche. Il formato del *report* è il *Markdown*, scelto in quanto GitHub può facilmente renderizzarlo per rendere il contenuto del file più leggibile, mentre la versione non renderizzata è soggetta al versionamento integrato in Git. Dopo aver inserito `report.md` all'interno del *bucket* S3, la Lambda si interfaccia a sua volta con il Parameter Store per recuperare l'IAT precedentemente utilizzato da `prepVASInput` per interfacciarsi con l'API di GitHub e lo utilizza per effettuare il *push* del *report* tramite la seguente tipologia di chiamata:

- `PUT /repos/owner/repo/contents/path`: consente di inserire un file nel *path* e nel *branch* della *repository* specificato, purchè il contenuto sia criptato in *base64* e il *body* della richiesta contenga:

```
{
  owner: owner,
  repo: repositoryName,
  path: filePath,
  message: commitMessage,
```

```
    content: Buffer.from(report,'utf-8').toString('base64'),
    branch: pushBranch,
    sha: sha
}
```

È importante sottolineare un dettaglio del messaggio del *commit* utilizzato per effettuare il *push* del *report*, che è nel formato:

```
Add report generated by VAS [skip-VAS]
```

In questo contesto il *tag* **[skip-VAS]** viene utilizzato per evitare di scatenare nuovamente il servizio nel momento in cui il *commit* avviene su *branches* monitorati dalle GitHub Actions come descritto in precedenza; se non vi fosse un *tag* che blocca la GitHub Action in partenza, ci sarebbe il rischio di innescare un *loop* in cui un *commit* su un determinato *branch* richiama il VAS, il VAS effettua il *push* con un *commit* sul *branch* di partenza, il che richiama nuovamente il VAS e così via. Questo meccanismo risulta utile anche per disabilitare il servizio manualmente nel momento in cui non si desidera farne uso in seguito ad un *commit*, semplicemente includendo il *tag* all'interno del messaggio: ad esempio, nell'eventualità in cui si desideri effettuare un *commit* direttamente sul *default branch* per un *hotfix*, oppure sul *source branch* di una PR aperta per aggiornarla senza far partire il servizio, è sufficiente utilizzare il *tag* per ottenere l'effetto desiderato.

5.6.3.2 S3

Per concludere, un *bucket* S3 viene utilizzato per mantenere lo storico delle esecuzioni del servizio, oltre che per fornire un metodo di archiviazione semplice e rapido di cui la Step Function si serve per salvare e reperire tutti i dati rilevanti per la revisione, come le risposte del modello; esso è strutturato nel modo seguente:

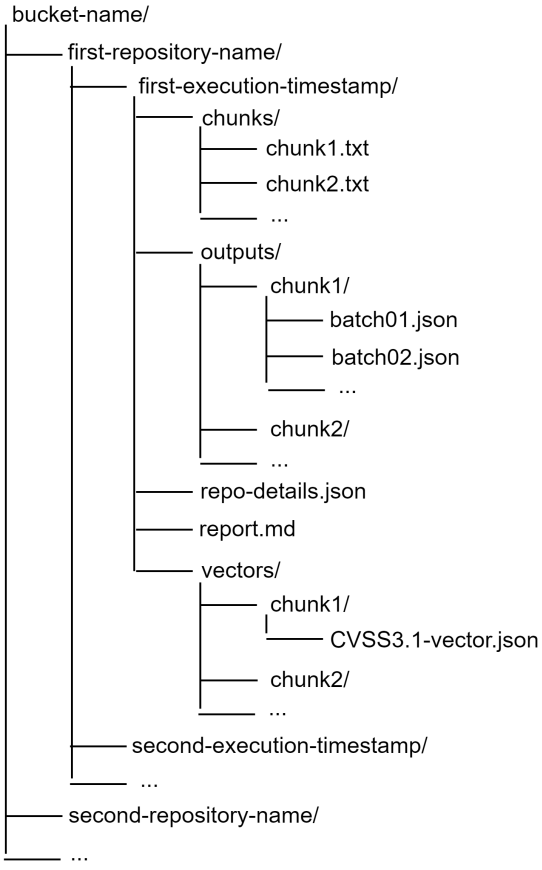


Figura 5.5: Struttura del bucket S3

Capitolo 6

Descrizione del servizio

In questo capitolo vengono esposti gli aspetti più rilevanti della fase di implementazione del servizio.

6.1 Configurazione Serverless

L'architettura AWS del servizio è stata definita utilizzando il *Serverless Framework*, che consente di sviluppare ed effettuare il *deployment* di applicazioni *serverless* su diverse piattaforme *cloud*. Dato che Serverless è impostato per utilizzare sia file di configurazione YAML che *Typescript*, si è scelto di utilizzare *Typescript* perchè fornisce un sistema di tipizzazione statica che aiuta ad individuare eventuali errori di tipo durante la compilazione.

Il file di configurazione `serverless.ts` include la specifica della versione del *framework* e l'elenco dei *plugin* utilizzati per estendere le funzionalità di Serverless. Il provider è naturalmente AWS, configurato con il *runtime Node.js* nodejs20.x e impostato per operare nella regione us-east-1 (visto che mette a disposizione un maggior numero di modelli in Bedrock). Il nome dello *stack* e il *bucket* di *deployment* sono definiti dinamicamente in base allo *stage* (`dev` o `prod`). È configurata anche la gestione dell'API Gateway con una chiave API. Le impostazioni IAM definiscono un ruolo per le funzioni Lambda con autorizzazioni per operare su risorse come S3 e Step Function.

Infine, sono incluse le definizioni delle funzioni Lambda (functions), risorse personalizzate (custom), risorse di infrastruttura (resources) e *workflow* basati su Step Functions (stepFunctions). Essendo di particolare rilevanza, *snippets* di `functions` e `stepFunctions` vengono mostrati nelle sezioni successive per comprendere meglio l'implementazione della Step Function e delle Lambda.

6.2 Implementazione

6.2.1 GitHub Actions

Le due GitHub Actions utilizzate per richiamare il servizio automaticamente in seguito a determinati *trigger* sono strutturate in modo molto simile, ovvero:

```
steps:
  - name: Trigger VAS for Entire Repository

      JSON='{
        "appId": "${{ secrets.APP_ID }}",
        [...]
        "owner": "${{ github.repository_owner }}",
        "repositoryName": "${{ github.repository }}",
        "branchName": "main"
      }'

      curl -X POST https://1y393x0s06.execute-api.us-east-1.amazonaws.com/dev/prepVASInput \
        -H "Content-Type: application/json" \
        -H "x-api-key: ${{ secrets.API_KEY }}" \
        -d "$JSON"
```

Listato 6.1: JSON payload e richiesta API delle GitHub Actions

La Action è condizionata dall'`if` *statement*

```
if: "!contains(github.event.head_commit.message, '[skip-VAS]')"
```

che verifica se il messaggio del *commit* non contiene il *tag* [skip-VAS]. Se questa condizione è soddisfatta, viene eseguito il passaggio `Trigger VAS for Entire Repository`, che costruisce un *payload* `JSON` contenente le credenziali necessarie affinchè le Lambda che devono utilizzare l'API di GitHub possano autenticarsi e informazioni sul *branch* o la PR della *repository* di interesse (a seconda dell'Action); infine, l'Action utilizza `curl` per inviare una richiesta di tipo `POST` all'*endpoint* messo a disposizione da API Gateway. Tutte le informazioni sensibili, inclusa l'API key di *API Gateway*, vengono reperite dai *secrets* di GitHub. Contrariamente a quanto mostrato per `analyse-default.yml`, `analyse-pr.yml` include

```
"pullRequestId": "${{ github.event.pull_request.number }}",
"sourceBranch": "${{ github.event.pull_request.head.ref }}"
```

nel *payload* (al posto di `"branchName":"main"`) e invia la richiesta `POST` all'*endpoint* associato alla Lambda `prepVASInputForPR`.

6.2.2 Lambda: prepVASInput

La Lambda `prepVASInput` è configurata nel modo seguente:

```
prepVASInput: {
  handler: `${handlerPath(__dirname)}/handlers/VASInput/prepVASInput.handler`,
  name: '${self:provider.stage}-${self:service}-prepVASInput',
  events: [{
    http: {
    method: 'post',
    path: 'prepVASInput',
    cors: true,
    private: true,
    },
  }],
```

```
}
```

Listato 6.2: Configurazione di prepVASInput

Il campo `handler` specifica il percorso del file *handler* che contiene la logica eseguita quando la Lambda è invocata. Il nome della funzione Lambda (`name`) è dinamicamente composto utilizzando le variabili di contesto del *provider*, per garantire un nome univoco tra i diversi *stage* e servizi all'interno dell'account AWS. La Lambda è innescata da un evento `HTTP`, configurato per rispondere a richieste `POST` all'*endpoint* `prepVASInput` di API Gateway. Anche opzioni come il supporto `CORS` e la configurazione di accesso privato tramite API key sono gestite direttamente nella configurazione dell'evento `HTTP`.

Da notare che `prepVASInput` e `prepVASInputForPR` sono le uniche Lambda a possedere un evento configurato in questo modo; tutte le altre Lambda non necessitano di eventi scatenanti in quanto vengono orchestrate dalla Step Function.

La Lambda richiama la funzione `filterRelevantFiles` che, secondo necessità, può utilizzare la funzione `createIAT` per creare l'IAT sfruttando `@octokit/auth-app`, una libreria fornita dal *framework client* **Octokit** che facilita l'autenticazione delle applicazioni GitHub. Come accennato poco fa nella sezione 6.2.1, le credenziali necessarie vengono fornite attraverso il *payload* `JSON` della `POST` della GitHub Action, che costituisce l'input stesso della Lambda.

```
const auth: AuthInterface = createAppAuth({
  appId: input.appId,
  privateKey: input.privateKey,
  clientId: input.clientId,
  clientSecret: input.clientSecret,
});

const instAuth: InstallationAccessTokenAuthentication = await auth({
  type: 'installation',
  installationId: input.installationId,
});
```

Listato 6.3: Creazione dell'IAT su GitHub utilizzando @octokit/auth-app

Successivamente, la Lambda richiama le funzioni `listFilesRecursively` e `getFileContent`, incaricate di reperire l'elenco dei percorsi dei vari file e il contenuto di ciascun file, rispettivamente. Entrambe le funzioni utilizzano `@octokit/core`, un pacchetto fondamentale per interagire con l'API di GitHub utilizzando Octokit. Esso fornisce le funzionalità di base necessarie per effettuare richieste `HTTP` all'API di GitHub, gestire l'autenticazione e interpretare correttamente le risposte `JSON` restituite. Una volta autenticatesi utilizzando l'IAT creato o salvato nel Parameter Store in precedenza con

```
const octokit: Octokit = new Octokit({
  auth: `Bearer ${IAT}`,
});
```

Listato 6.4: Autenticazione su GitHub utilizzando @octokit/core

le funzioni utilizzano le chiamate API descritte nella sezione 5.6.1.3 restituendo all'*handler* di `prepVASInput` una stringa contenente tutto il codice sorgente di interesse suddiviso in files, con commenti che ne specificano l'inizio e il termine.

Va specificato che nel caso in cui si stia analizzando il contenuto di una PR, i file recuperati dalla funzione e inclusi nella stringa non sono solamente quelli con stato `added` o `modified` nell'ultimo *commit* sul *source branch*, ma quelli che conservano tale stato considerando tutti i *commit* effettuati all'interno della PR da quando è stata aperta.

Una volta suddivisa la stringa in *chunks* della dimensione adatta utilizzando il pacchetto `mistral-tokenizer-ts` (questi vengono caricati su S3), la Lambda può richiamare la funzione `startVAS`, che utilizza il comando `StartExecutionCommand` di `@aws-sdk/client-sfn`, una parte dell'SDK ufficiale di AWS per *JavaScript* dedicata a AWS Step Functions, per innescare la Step Function.

6.2.3 Step Function

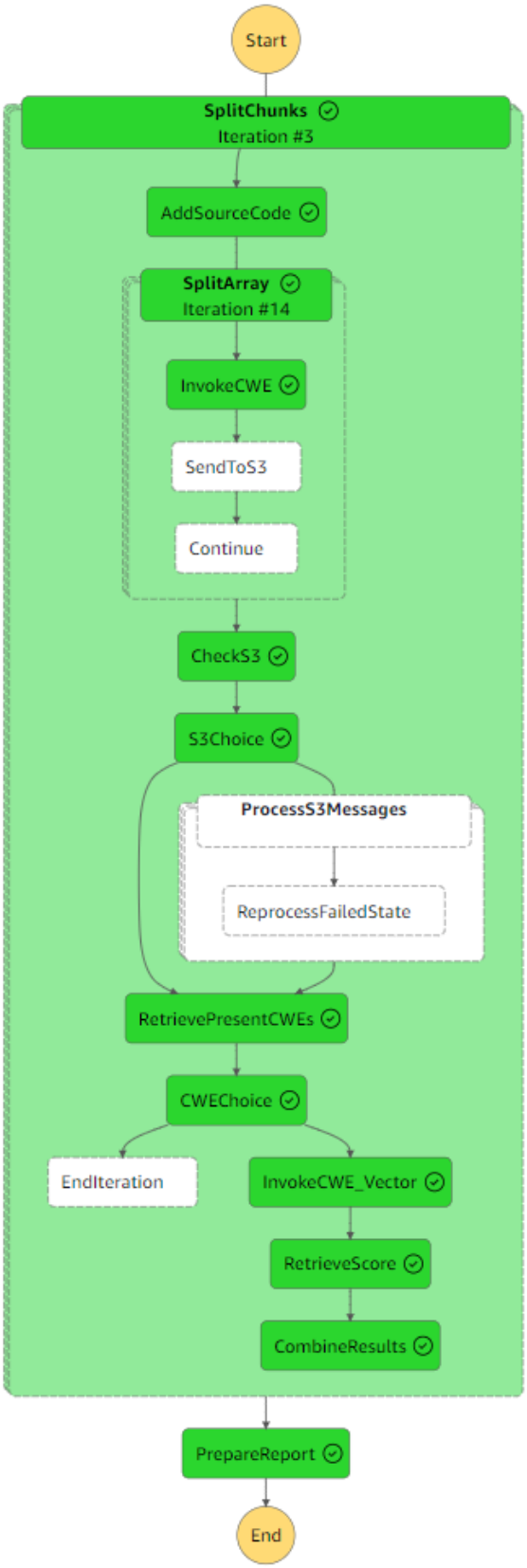


Figura 6.1: Implementazione della Step Function

6.2.3.1 SplitArray

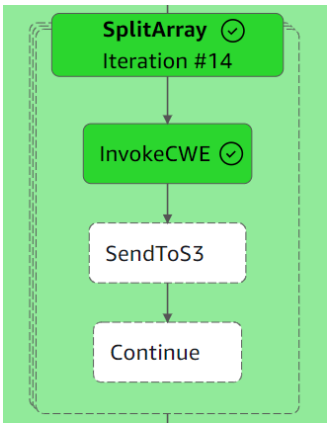


Figura 6.2: Step Function: SplitArray

6.2.3.2 SplitArray

Il *workflow* della Step Function eseguito in ogni iterazione di `SplitChunks`, ossia su ogni *chunk* in cui è stato diviso il codice sorgente, inizia fondamentalmente con lo stato `SplitArray`, che itera a sua volta sui *batches* di CWE; ogni iterazione, infatti, richiama la Lambda `CWEChecker` all'interno dello stato `InvokeCWE` e, in caso di necessità, anche lo stato `SendToS3`:

```
SplitArray: {
  Type: 'Map',
  MaxConcurrency: 3,
  Iterator: {
    StartAt: 'InvokeCWE',
    States: {
      InvokeCWE: {
      Type: 'Task',
      Resource: {
        'Fn::GetAtt': ['CWEChecker', 'Arn'],
      },
      }
    }
  }
}
```

Listato 6.5: Configurazione parziale di SplitArray con mostrato il primo stato richiamato da ogni iterazione: InvokeCWE

La Lambda `CWEChecker` invoca il modello fornendo le CWE del *batch* e il codice del *chunk* processati nell'iterazione corrente utilizzando il *prompt* descritto nella sezione 5.4 e servendosi di un *parser* creato appositamente per accertarsi che il modello risponda come previsto.

Il modello viene invocato per mezzo della funzione `invokeModel`, che utilizza il comando `InvokeModelCommand` di `@aws-sdk/client-bedrock-runtime`, una parte dell'SDK ufficiale di AWS per *JavaScript*, progettato specificamente per interagire con Bedrock:

```
export async function invokeModel(prompt: string, sourceCode: string):
    Promise<string> {
  const command: InvokeModelCommand = new InvokeModelCommand({
    modelId: 'mistral.mistral-large-2402-v1:0',
    contentType: 'application/json',
    accept: 'application/json',
    body: JSON.stringify({
      prompt: `<s>[INST] ${prompt}\n\n${sourceCode} [/INST]`,
      max_tokens: 4096,
      top_k: 50,
      top_p: 0.9,
      stop: [],
      temperature: 0.4,
    }),
  });
  [...]
};
```

Listato 6.6: Estratto della funzione invokeModel con mostrato il body del comando InvokeModelCommand

Da notare l'uso dei *tag* `[INST]` e `[/INST]`, utilizzati in Mistral Large per delimitare l'istruzione particolare che ci si aspetta che il modello processi [33], aiutandolo a separare il testo che costituisce l'input diretto dal resto del contesto o dal testo circostante. Il massimo numero di *tokens* generati in output in una singola risposta (`max-tokens`) è stato ridotto a 4096 (contro il massimo di 8192) per ottenere risposte complete ma non eccessivamente prolisse. `top_k` specifica il numero di *token* più probabili tra cui il modello può scegliere, perciò è stato fissato a 50 per fare in modo che il modello selezioni il prossimo *token* tra i 50 più probabili, il che risulta in risposte più coerenti e meno creative. Allo stesso modo `top_p`, la soglia di probabilità cumulativa che il modello considera nel selezionare i *token*, è stato fissato a 0.9 per fare in modo che questo selezioni i *token* tra quelli la cui somma di probabilità è almeno il 90% del totale, producendo risposte più coerenti. Infine anche la `temperature` (controlla a sua volta la creatività e la casualità delle risposte) è stata calibrata scegliendo un valore relativamente basso (0.4), in modo da ottenere risposte generalmente più precise e meno creative.

Il *parser*, `CWECheckerParser`, inizialmente invoca il modello fornendo tutte le CWE da controllare e analizza la risposta utilizzando le espressioni regolari

1. `/^CWE(\d+)(?: \(.+?\))?: (present|not present)$/`
2. `/^(File|Area|Explanation|Resolution): (.+)$/`

e costruendo degli oggetti `JSON` di tipo `CWECheckResult` man mano che individua risposte complete di tutti i campi previsti, ovvero:

```
export interface CWECheckResult {
  cweId: number;
  status: string;
```

```
  file?: string;
  area?: string;
  explanation?: string;
  resolution?: string;
}
```

La prima *regex* è composta nel modo seguente:

- `^`: indica che il *pattern* deve essere collocato all'inizio della stringa analizzata
- `CWE`: corrisponde esattamente alla sequenza di caratteri "CWE"
- `(\d+)`: *capturing group* che corrisponde a uno o più cifre (`\d`); ad esempio, corrisponde a "22" in "CWE22"
- `(?: \(.+?\))?`: *non-capturing group* che corrisponde a una descrizione opzionale tra parentesi dopo il numero CWE
- `: `: corrisponde esattamente a due punti seguiti da uno spazio
- `(present|not present)`: altro gruppo di cattura che corrisponde esattamente alle espressioni "present" o "not present"
- `$`: indica che il *pattern* non va oltre la fine della stringa analizzata (è su una sola riga)

Gli elementi che contraddistinguono la seconda *regex* rispetto alla prima sono:

- `(File|Area|Explanation|Resolution)`: corrisponde esattamente a una delle parole chiave tra parentesi
- `(.+)$`: *capturing group* che corrisponde a una o più occorrenze di qualsiasi carattere fino alla fine della stringa analizzata (per catturare la spiegazione del modello in merito al campo in considerazione)

Nel caso in cui il modello non risponda correttamente ad alcune CWE (con risposte formattate erroneamente o solo parzialmente complete) o le ignori, l'id di queste viene salvato in modo da invocarlo nuovamente con un input ristretto alle sole CWE che non hanno ancora ricevuto risposta; questo finchè tutte le CWE hanno ricevuto una risposta valida, o finchè `CWEChecker` non non va in *timeout* dopo 150 secondi (il *timeout* di ogni Lambda è configurabile nel file di configurazione `serverless.ts`).

`SplitArray` è configurato con una logica di *retry* e un blocco *catch* automatico in `stepFunctions`, in modo che se si verificano errori di *throttling* o la Lambda va in *timeout*, si tenti nuovamente di eseguire lo stato dopo 1 secondo, fino a un massimo di 5 volte:

```
Retry: [
  {
    'ErrorEquals': ['Lambda.TooManyRequestsException', '
    ThrottlingException', 'Lambda.Unknown'],
    'IntervalSeconds': 1,
    'MaxAttempts': 5,
    'BackoffRate': 1.0,
  }
]
```

Listato 6.7: Logica di retry di SplitArray

Se, nonostante ciò, lo stato fallisce dopo i 5 tentativi previsti, il *catch block* fa in modo che informazioni sullo stato vengano registrate all'interno della cartella `failed` in S3 dallo stato `SendToS3`, così da tentarlo nuovamente in un secondo momento.

```
Catch: [
  {
    ErrorEquals: ['States.ALL'],
    ResultPath: '$.error',
    Next: 'SendToS3',
  }
]
```

Listato 6.8: Blocco catch di SplitArray

6.2.3.3 ProcessS3Messages

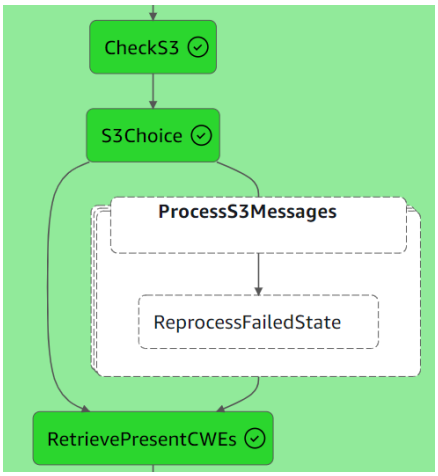


Figura 6.3: Step Function: ProcessS3Messages

Successivamente, lo stato `CheckS3` si occupa di controllare se ci sono stati precedentemente falliti salvati all'interno di S3 e informa la logica di ramificazione implementata nello stato `S3Choice`:

```
S3Choice: {
  Type: 'Choice',
    Choices: [
    {
      Variable: '$.S3CheckResult.hasMessages',
      BooleanEquals: true,
      Next: 'ProcessS3Messages',
    },
    ],
    Default: 'RetrievePresentCWEs',
}
```

Listato 6.9: Configurazione dello stato S3Choice con mostrata la scelta condizionale tra gli stati RetrievePresentCWEs e ProcessS3Messages

Nel caso in cui vi siano stati che necessitano di essere processati nuovamente per mezzo della Lambda `CWEChecker`, viene invocato lo stato `ProcessS3Messages`, che

funziona in modo del tutto analogo allo stato `SplitArray` (pur non implementando l'invio degli stati falliti ad S3); nel caso contrario viene invocato direttamente lo stato `RetrievePresentCWEs`, che si occupa semplicemente di esaminare il contenuto della cartella `outputs` in S3 per determinare quali CWE siano state rilevate dal modello all'interno del *chunk* in esame.

6.2.3.4 InvokeCWEVector

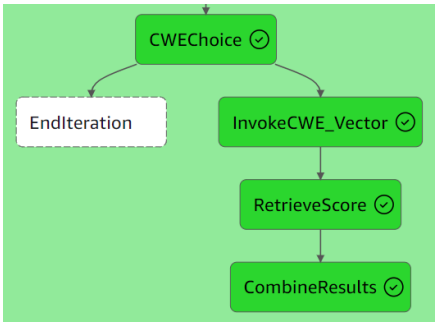


Figura 6.4: Step Function: InvokeCWEVector

Infine, lo stato `CWEChoice` fa in modo che il *workflow* si dirami in base alla presenza o meno di CWE all'interno del *chunk* in esame. Se non sono state rilevate CWE, l'iterazione termina; altrimenti, si prosegue con lo stato `InvokeCWEVector` che si occupa di invocare nuovamente il modello per assegnare un CVSS Vector a ogni CWE presente. Lo stato richiama la Lambda `CWEVector`, che invoca il modello utilizzando il *prompt* descritto nella sezione 5.4 e la funzione `invokeModel` descritta nella sezione 6.2.3.1; inoltre, anche questa Lambda si serve di un *parser* creato appositamente per verificare che il modello generi il numero atteso di vettori nel formato richiesto. `CVSSVectorParser` utilizza la stessa strategia di `CWECheckerParser`, dapprima invocando il modello con un elenco completo di tutte le CWE che necessitano di un vettore associato e restringendo via via l'input alle sole CWE che non sono ancora state processate correttamente nelle invocazioni successive fino ad esaurirle (o fino a quando la Lambda non va in *timeout* dopo 180 secondi). Le espressioni regolari adottate dal *parser* per rilevare i vettori strutturati correttamente in ogni risposta sono:

1. `/CWE\s*(\d+)\b/`
2. `/^Vector: (CVSS:\d\.\d\/(?:AV:[A-Z]\/AC:[A-Z]\/PR:[A-Z]\/UI:[A-Z]\/S:[A-Z]\/C:[A-Z]\/I:[A-Z]\/A:[A-Z]))$/`
3. `/^(AV|AC|PR|UI|S|C|I|A): (.+)$/`

La prima *regex* è piuttosto semplice da comprendere, l'unico elemento di novità rispetto a quelle viste precedentemente è `\b`, che indica che la corrispondenza delle cifre numeriche catturate dopo "CWE" termina quando si incontra un limite di parola,

cioè un carattere che non è un carattere alfanumerico oppure uno spazio; in questo modo ci si assicura che la corrispondenza sia precisa e non includa caratteri non desiderati.

La seconda *regex* è strutturata nel modo seguente:

- `Vector:` : corrisponde esattamente alla sequenza di caratteri "Vector: "
- `(CVSS:\d\.\d\/(AV:[A-Z]\/AC:[A-Z]\/PR:[A-Z]\/UI:[A-Z]\/S:[A-Z]\/C:[A-Z]\/I:[A-Z]\/A:[A-Z]))`: cattura l'intero vettore CVSS, che inizia con "CVSS:", seguito dalla versione in formato X.Y (dove X e Y sono cifre) e tutte le metriche di base separate da "/"; le metriche stesse sono raggruppate in un *non-capturing group* `(?:...)` per evitare di catturarle singolarmente

Infine, la terza *regex* contiene:

- `(AV|AC|PR|UI|S|C|I|A)`: cattura uno qualsiasi dei seguenti prefissi: `AV`, `AC`, `PR`, `UI`, `S`, `C`, `I`, `A`
- `(.+)`: cattura uno o più caratteri qualsiasi, rappresenta il valore assegnato dal modello alla metrica di base

La funzione analizza il contenuto della risposta costruendo oggetti `JSON` di tipo `CVSSVectorResult` man mano che incontra vettori validi, purchè abbiano tutti i campi richiesti, ovvero:

```
export interface CVSSVectorResult {
  cweId: number;
  vector: string;
  av: string;
  ac: string;
  pr: string;
  ui: string;
  s: string;
  c: string;
  i: string;
  a: string;
}
```

A differenza di `CWEChecker` che è inglobata all'interno di uno stato di tipo `Map`, in modo che sia questo a gestire le iterazioni e quindi le invocazioni in modalità concorrente, `CWEVector` non lo è. In principio, infatti, non si credeva fosse necessario processare le CWE rilevate in *batches* e si è tentato di effettuare un'unica invocazione che generasse tutti i vettori corrispondenti; tuttavia, testando il funzionamento della Step Function, ci si è resi conto che non si tratta di un approccio efficiente perchè non c'è modo di sapere a priori quante CWE verranno rilevate dal modello (175, nel caso peggiore), e questo impiega una quantità di tempo non impercettibile a generare ogni vettore (mediamente di più rispetto alla generazione delle CWE in `CWEChecker`). Perciò si è deciso di implementare una logica di *batching* direttamente all'interno della Lambda, in modo da processare *batches* di 10 CWE in modalità concorrente:

```
const batchSize: number = 10;
const batchedCWEIds: CWECheckResult[][] = [];
```

```
3
4  for (let i: number = 0; i < presentCWEs.length; i += batchSize) {
5    batchedCWEIds.push(presentCWEs.slice(i, i + batchSize));
6  }
7
8  const batchPromises: Array<Promise<void>> = batchedCWEIds.map(async (
       batch: CWECheckResult[]) => {
9    [...]
10 }
11 await Promise.all(batchPromises);
```

Listato 6.10: Estratto di CWECheckerParser con mostrata la costruzione e l'esecuzione concorrente di un array di promesse per processare le CWE in batches asincrone

Innanzitutto si suddivide un array di CWE in *batches* di dimensione 10, definita da `batchSize`. Poi si utilizza un ciclo `for` per creare i *batches*, memorizzandoli nell'array `batchedCWEIds`. Successivamente, si crea un array di promesse (`batchPromises`) che processano ciascun *batch* in modo asincrono usando `map`. Infine, `Promise.all(batchPromises)` esegue tutte le promesse in parallelo; questo approccio permette di generare tutti i CVSS Vector in modo efficiente, evitando di sovraccaricare il modello tentando di processare tutti gli elementi simultaneamente.

Una volta generati i vettori, questi vengono utilizzati per calcolare il CVSS 3.1 Score corrispondente dalla Lambda `prepareScore`, invocata dallo stato che succede `InvokeCWEVector`, `RetrieveScore`; la Lambda utilizza la libreria `vuln-vects` per validare il vettore (con `validateCvssVector`) e convertirlo nello *score* numerico e *severity* corrispondenti (con `parseCvssVector`) come segue:

```
1 for (const vectorObject of parsedContent.data) {
2   const vectorString: string = vectorObject.vector;
3   const isValidCvssVector: boolean = validateCvssVector(vectorString);
4   if (isValidCvssVector) {
5     const cvssVectorScore: CvssScore = parseCvssVector(vectorString);
6     const baseScore: number = cvssVectorScore.baseScore;
7     const severity: string = cvssVectorScore.cvss30verallSeverityText;
```

Listato 6.11: Estratto di prepareScore con mostrata la validazione del vettore e la creazione dello score numerico corrispondente

6.2.4 Lambda: prepVASOutput

Infine, l'ultimo stato della Step Function (`PrepareReport`, che viene invocato da `SplitChunks` una volta processati tutti i *chunks* del codice sorgente, richiama la Lambda `prepVASOutput`; la funzione costruisce il *report* e lo invia alla funzione `pushReport`, che si occupa di inserirlo in GitHub. `pushReport` riutilizza `@octokit/core` per autenticarsi con l'IAT recuperato dal Parameter Store ed effettua una chiamata `GET /repos/owner/repo/contents/path` con `path:report.md` per reperire lo *sha* del *report*, se già presente all'interno della *repository*; in questo caso, lo *sha* è necessario per sostituire il file con la versione appena generata. Il *branch* all'interno del quale viene inserito il *report* coincide con il *default branch* specificato nella GitHub Action

pertinente in caso si sia analizzato l'intero contenuto della *repository*, oppure con il *source branch* della PR in caso si siano analizzati solamente i file con stato `added` o `modified` all'interno della PR stessa.

Capitolo 7

Uso del servizio

In questo capitolo vengono fornite alcune considerazioni di carattere pratico sull'uso del servizio realizzato in un contesto reale, assieme ad alcune indicazioni per eventuali miglioramenti.

7.1 Considerazioni sull'uso pratico del servizio

Il servizio è stato concepito e realizzato con lo scopo di individuare vulnerabilità nel codice sorgente all'interno di *repository* GitHub in fase di sviluppo, e integrato nel flusso di lavoro *CI/CD* per rendere il processo completamente automatico; la revisione è orientata all'individuazione di eventuali debolezze intrinseche nel codice (CWE) ed è a carico di Mistral Large, un modello di Generative AI messo a disposizione da AWS Bedrock. Come previsto, i risultati vengono poi aggregati e dispiegati all'interno di un *report*, a disposizione nella *repository*.

Nonostante il servizio soddisfi appieno l'idea di partenza, fornendo un modo rapido e agevole di monitorare il livello di sicurezza del codice in via di sviluppo, è necessario fare alcune considerazioni sulla sua usabilità in un contesto reale.

Innanzitutto, il tempo di esecuzione varia sostanzialmente in base alla lunghezza del codice da analizzare e alla presenza di vulnerabilità al suo interno. Naturalmente, più la *repository* è grande maggiore sarà il numero di *chunks* da processare; dato che questi vengono analizzati al massimo 3 alla volta in modalità concorrente, per *repository* di dimensioni medio-grandi (che tendono a produrre dai 15 ai 20 *chunks*) questo può comportare un tempo di analisi considerevole (20+ minuti).

Purtroppo, come spigeato in precedenza, i limiti di concorrenza sono strettamente necessari per evitare di sovraccaricare Bedrock con le invocazioni del modello ottenendo fin troppi errori di *throttling*, con il conseguente invio della maggior parte degli stati, falliti ripetutamente, ad S3; le quote di Bedrock limitano l'accesso ai modelli in modo non impercettibile, perciò si tratta di un aspetto da tenere indubbiamente in considerazione nel progettare applicazioni che ne fanno uso. Inoltre, più vulnerabilità (CWE, in questo contesto) vengono rilevate all'interno di un determinato *chunk*, maggiore è il numero di CVSS Vector che il modello deve generare; pur dividendo le CWE in *batches* di 10 e processandoli in modalità concorrente, il modello impiega comunque

una quantità di tempo proporzionale al numero di *batches* per la generazione di tutti i vettori.

Si è pensato di invocare il modello per generare i vettori man mano che le CWE vengono rilevate per evitare di introdurre un *bottleneck* verso il termine del flusso di lavoro, ma anche questo tipo di implementazione soffrirebbe di problemi di *throttling*. Ovviamente il problema diventa meno rilevante nel contesto dell'analisi di PR, visto che il codice prelevato da GitHub in quel caso ha dimensioni minori rispetto all'intera *repository*.

In secondo luogo, sempre in merito al tempo di esecuzione, vi è un *overhead* non indifferente introdotto dalle invocazioni ripetute del modello nel momento in cui questo non risponde in modo corretto alle richieste di individuazione di determinate CWE, piuttosto che di generazione dei vettori; pur essendo ragionevole, l'approccio di restringere progressivamente di più l'input utilizzato dai *parser* fa sì che l'esito dell'esecuzione sia fondamentalmente alla merce del comportamento del modello stesso, senza un vero e proprio controllo sul numero di chiamate effettuate piuttosto che sul tempo di esecuzione.

Nel caso pessimo in cui il modello non risponde correttamente a nessuna delle richieste ripetute, essendovi inevitabilmente limiti preimpostati sul numero di tentativi e sui *timeout* delle Lambda, l'esecuzione del servizio fallisce irrimediabilmente. Una delle possibili migliorie potrebbe dunque essere lo sviluppo di *parser* più avanzati che riescano ad interpretare e ricavare il necessario dalle risposte del modello anche qualora queste siano parzialmente inesatte nel formato utilizzato; questo porterebbe sicuramente a fare meno invocazioni ripetute e a ridurre l'*overhead* sul tempo di esecuzione.

A seguito di questo, è importante fare anche alcune considerazioni sui costi del servizio, in particolare quelli attribuibili all'uso di Bedrock; come appena spiegato, non avendo il pieno controllo sul numero di chiamate effettuate o sul tempo di esecuzione complessivo per l'analisi, diventa difficile fare anche una stima accurata dei costi incorsi in seguito ad ogni esecuzione.

A scopo puramente informativo, il costo di Mistral Large in base ai *token* utilizzati su Bedrock è di 0,004 USD/1000 *token* in input e 0,012 USD/1000 *token* in output (il che coincide con i costi ufficiali di Mistral AI [32]); ciò significa che, nel caso **ottimo**, per ogni singolo *chunk* di codice, si effettuano 15 chiamate per controllare ciascun *batch* di CWE, dove ognuna trasporta all'incirca 8,192 *token* in input (visto che quella è la dimensione dei *chunks*). Questo significa che si hanno 8,192 *token* x 15 chiamate = 122,880 *token* in input. D'altro canto, il modello replica descrivendo tutte e 12 le CWE come non presenti all'interno del *chunk* con `CWEXX: not present`, e questo per ogni *batch*; considerando questo scenario, si ha che le 12 risposte moltiplicate per 15 *batches* ammontano all'incirca a 2,880 caratteri di testo che potrebbero corrispondere all'incirca a 576 *token* Mistral (purtroppo l'uguaglianza Mistral *token* = numero di caratteri non è nota).

Non essendo state rilevate CWE, non si rende necessario invocare nuovamente

il modello per la generazione dei vettori. Di conseguenza, si incorre in un costo di 122,880 *token* x (0,004 USD/1000 *token* di input) + 576 *token* x 0,012 USD/1000 *token* di output = 0,49152 USD + 0,006912 USD = **0,498432 USD**/*chunk* di codice; per una *repository* di dimensioni medio-piccole (5 *chunks*) questo significa spendere all'incirca 2,49216 USD/esecuzione del servizio (nel caso ottimo).

Naturalmente, il costo è destinato ad aumentare in proporzione al numero di chiamate e alla dimensione della *repository* in situazioni non ottimali, il che rende la spesa sostenuta per l'uso del servizio un altro fattore da non sottovalutare nel progettare servizi di natura simile al VAS.

Un'ultima considerazione importante va fatta in merito alla capacità del modello di interpretare il codice sorgente che gli viene fornito; infatti, per quanto Mistral Large abbia dimostrato del potenziale nel condurre una *code review* organizzata come descritto, si tratta pur sempre di un modello di testo non specializzato nella visione di codice; un'indicazione per una possibile integrazione futura del servizio è l'utilizzo di un modello il cui processo di *training* include un *corpus* più vasto di frammenti di codice vero e proprio, come **Code Llama** di *Meta* [11].

Infatti, pur essendo esterno a Bedrock, il modello è da poco importabile all'interno di AWS grazie ad una funzionalità recentemente resa disponibile a tutti gli utenti (a partire dal 30 giugno 2024), in modo da poter essere utilizzato esattamente allo stesso modo di Mistral Large in questo contesto; grazie alla sua formazione focalizzata sulla comprensione e generazione di codice, Code Llama potrebbe essere più competente nell'identificare vulnerabilità e nel fornire suggerimenti e correzioni più pertinenti.

7.2 Considerazioni su approcci alternativi

Come evidenziato nella sezione 3.1, la maggior parte delle applicazioni attuali della Generative AI nella *cybersecurity* sfrutta il potenziale dei LLM e, in particolare, di ChatGPT; ChatGPT, infatti, costituisce assieme a Gemini di *Google* il modello più popolare e comunemente utilizzato in tale ambito [17]. Volendo adottare un'approccio alternativo per l'implementazione del VAS, si potrebbe progettare e realizzare tutta l'infrastruttura utilizzando i servizi *Azure* di *Microsoft*, tra cui vi è anche *Azure OpenAI*, che consente di utilizzare i modelli *OpenAI* per effettuare la revisione del codice, inclusi ChatGPT-4o e Codex.

Per quanto riguarda l'orchestrazione delle funzioni *serveless*, *Azure* mette a disposizione un servizio chiamato *Durable Functions* che, similmente ad AWS Step Functions, consente di creare e gestire *workflows* complessi utilizzando *C#* o *JavaScript* (rispetto alla ASL), permettendo l'esecuzione sequenziale o concorrente di funzioni.

Per quanto riguarda invece l'uso di ChatGPT-4o, ad esempio, le quote definite da *Azure* per il piano "Standard GPT-4o (Default)" sono di 900 richieste/minuto e di 150,000 *tokens*/minuto [21], e sono le più limitanti rispetto a quelle definite per piani più costosi; confrontando tali limitazioni con le 400 richieste/minuto e ai 300,000 *tokens*/minuto di Bedrock per Mistral Large, ci si rende facilmente conto che i due

scenari sono piuttosto simili in quanto al carico che ci si può aspettare di far gestire ai modelli per ogni minuto di esecuzione. Con più del doppio delle richieste ma con la metà dei *tokens* in input, anche l'utilizzo di ChatGPT-4o tramite *Azure OpenAI* sarebbe inevitabilmente vincolato dalle stesse problematiche discusse sopra, a meno che non si decida di investire in piani che offrono limitazioni meno stringenti. Da notare anche che, a livello di costi, ChatGPT-4o è fondamentalmente più costoso di Mistral Large, in quanto il suo utilizzo richiede 0,005 USD/1000 *token* in input e 0,015 USD/1000 *token* in output [20].

In conclusione, a livello progettuale l'infrastruttura del VAS, seppur realizzato con i servizi *Azure*, dovrebbe sempre tener conto delle limitazioni imposte sull'utilizzo del modello; a livello di qualità del servizio offerto, tuttavia, ci si aspetta che ChatGPT-4o risulti più efficace di Mistral Large nell'individuare e spiegare eventuali CWE all'interno del codice sorgente.

Capitolo 8

Conclusioni

In questo capitolo vengono fatte alcune considerazioni finali sullo svolgimento del progetto.

8.1 Consuntivo finale

In tabella vengono riportate le attività effettivamente svolte nel corso delle 320 ore che hanno composto lo *stage* curricolare:

Periodo	Durata (ore)	Attività
1	32	Studio del linguaggio di programmazione *Typescript*, dei servizi AWS, del *framework Serverless*
2	64	Studio del contesto applicativo e progettazione della *code review*, selezione del modello e dei *prompt* e definizione dell'architettura
3	48	Sviluppo della Step Function che, dato in input del codice sorgente, fornisce un'analisi rudimentale delle CWE
4	32	Rimodellazione della Step Function per ovviare ai problemi di *throttling* in Bedrock e integrazione del sistema di *scoring*
5	24	Sviluppo del *report* contenente tutte le informazioni per svolgere attività di risoluzione
6	40	Sviluppo della logica di *parsing* per analizzare le risposte del modello
7	48	Integrazione del servizio nella *CI/CD* per avviare l'analisi del codice utilizzando le GitHub Actions e l'inserimento del *report* nella *repository* di partenza
8	32	Attività di *debugging* e produzione del Manuale Utente, oltre che della presentazione conclusiva.

Tabella 8.1: Descrizione delle attività effettuate in ciascun periodo

Il consuntivo finale diverge dalla pianificazione iniziale in quanto alcune attività hanno richiesto più tempo del previsto e si è scelto di riordinarle in modo leggermente diverso.

La fase iniziale di progettazione, e in particolare la definizione degli elementi da utilizzare nella *code review* e del flusso di lavoro della Step Function, è stata relativamente più lunga di quanto anticipato; questo è dovuto principalmente ad una lacuna di conoscenze nell'ambito della *cybersecurity* e a qualche difficoltà iniziale nel comprendere il funzionamento della Step Function (specie il modo in cui i parametri vegono passati tra stati).

Successivamente, la Step Function e le Lambda implicate sono state soggette ad un *refactoring* parziale e a *testing* estensivo per determinare la giusta combinazione di variabili (come i limiti fissati sulla concorrenza) per rendere il flusso di esecuzione più efficiente e minimizzare gli errori di *throttling* e *timeout*.

L'ostacolo successivo è consistito nell'individuazione di espressioni regolari per i *parser* che fossero adatte a catturare le informazioni di interesse nelle risposte del modello (pur mantenendo un margine minimo di flessibilità); le ore spese su questa attività sono giustificate dalla scarsa esperienza nell'utilizzo di *regex*.

Solo dopo aver realizzato il nucleo del servizio, si è deciso di passare al suo inserimento nella *CI/CD*, il che in realtà ha comportato meno tempo del previsto.

8.2 Raggiungimento degli obiettivi

Gli obiettivi del progetto delineati in partenza sono stati pienamente soddisfatti e le 8 settimane a disposizione, di cui 5 sono state utilizzate per l'implementazione vera e proria, sono risultate adeguate. Come già accennato la progettazione prolungata è stata frutto di una mancanza di conoscenze e/o esperienze pregresse nell'ambito applicativo, ma è comunque stata fondamentale per far sì che la fase di implementazione fosse conclusa entro i tempi previsti. Se ci fosse stato leggermente più tempo a disposizione, sarebbe stato interessante testare il servizio concluso con i vari modelli messi a disposizione da Bedrock per corroborare l'ipotesi che Mistral Large sia il più adatto al caso d'uso descritto o direttamente con modelli esterni come Code Llama.

8.3 Conoscenze acquisite e valutazione personale

L'esperienza di stage presso zero12 s.r.l si è trasformata entro i primi pochi giorni da un obbligo curricolare ad un'attività che ha riempito ogni giorno di sfide professionali e spunti per imparare dagli altri. Proveniendo da una realtà accademica in cui l'individualità e la competitività erano favorite e spesso addirittura premiate, sono stata sorpresa di trovare l'esatto opposto nell'ambiente creato dal personale aziendale; ho imparato molto nell'assistere a conversazioni tra colleghi e *meeting* veri e propri in cui diverse individualità si confrontavano per perseguire un'obiettivo comune, sempre in modo rispettoso e amichevole.

Posso dire, dunque, di essere pienamente soddisfatta di come si è svolta l'attività di *stage*, che mi ha insegnato tantissimo a livello di conoscenze tecniche, visto che

ho dovuto maneggiare in modo consapevole tecnologie mai utilizzate prima d'ora, ma ancor di più a livello personale. Ho imparato, infatti, che il *modus operandi* e le capacità di analisi critica che ho acquisito in passato si sposano molto bene con l'ambito dell'informatica e che, nelle circostanze giuste, sono in grado di rendere molto più di quanto non creda possibile.

Acronimi e abbreviazioni

AI Artificial Intelligence. 6, 18, 54

API Application Program Interface. 4, 10, 11, 20–24, 28, 31–34, 53, 55

AR Augmented Reality. 1, 53

ASL Amazon States Language. 10, 46

AWS Amazon Web Services. iv, 1, 3, 4, 9–12, 21, 23, 25, 26, 31, 33, 34, 36, 44, 46, 48, 54, 55

CTI Cyber Threat Intelligence. 7, 13

CVE Common Vulnerability and Exposures. 8, 13, 14, 54

CVSS Common Vulnerability Scoring System. 13, 15–17, 19, 22, 27, 28, 40–42, 44, 60

CWE Common Weakness Enumeration. 8, 13–15, 17–19, 21, 22, 26–28, 36–38, 40–42, 44, 45, 47, 48, 54, 58–61

IAM Identity and Access Management. 11, 31, 54

IAT Installation Access Token. 20, 24, 28, 33, 42

IoT Internet of Things. 1

LLM Large Language Model. 6–8, 46, 53

NLP Natural Language Processing. 7

npm Node Package Manager. 9, 55

NVD National Vulnerability Database. 13

OOP Object Oriented Programming. 9

OWASP Open Web Application Security Project. 14, 15, 58, 61

PAT Personal Access Token. 20

PR Pull Request. 22–24, 29, 32, 34, 43, 45

SAST Static Application Security Testing. 1, 3, 6, 8, 55

SDK Software Development Kit. 22, 34, 36

VAS Vulnerability Assessment Service. 3, 20, 29, 46, 47, 59

VM Virtual Machine. 6

VS Code Visual Studio Code. 12

Glossario

Amazon Web Services Piattaforma di *cloud computing* offerta da Amazon che fornisce una vasta gamma di servizi *cloud*, tra cui calcolo, *storage*, database, analisi, Machine Learning, sicurezza, sviluppo di applicazioni e molto altro.. 51

Application Program Interface Insieme di regole e protocolli che permette a diverse applicazioni *software* di comunicare tra loro; le API definiscono i metodi e i dati che gli sviluppatori possono utilizzare per interagire con servizi o applicazioni, facilitando l'integrazione e l'interoperabilità.. 51

Augmented Reality Tecnologia che sovrappone informazioni digitali, come immagini, video o dati, al mondo reale in tempo reale; utilizzando dispositivi come *smartphone*, *tablet*, occhiali AR o altri visori, l'AR integra elementi virtuali con l'ambiente fisico, migliorando l'interazione dell'utente con il mondo circostante.. 51

Chain-of-thought prompting Tecnica di dialogo in cui l'utente interagisce con un LLM guidando il flusso della conversazione attraverso una serie di *prompt* in sequenza; questa tecnica è utilizzata per ottenere informazioni dettagliate, esplorare diverse prospettive su un argomento o generare testi che seguano un filo logico preciso, consentendo una conversazione più strutturata con il modello.. 8

Cloud Native Approccio alla costruzione e gestione di applicazioni che sfrutta appieno i vantaggi del *cloud computing* basandosi su pratiche come i microservizi, i *container*, l'orchestrazione (ad esempio, Kubernetes), e il continuous integration/continuous delivery (CI/CD).. 1

Commonsense reasoning La capacità umana di comprendere e interpretare il mondo circostante attraverso logiche e presupposti che sono generalmente accettati come ovvi o intuitivi; questo tipo di ragionamento si basa su conoscenze e esperienze comuni, anziché su conoscenze specializzate o tecniche.. 7

Cyber Threat Intelligence Disciplina che si occupa della raccolta, analisi e interpretazione di informazioni riguardanti minacce informatiche potenziali o attuali; il suo scopo è identificare e comprendere le tattiche, tecniche e procedure (TTP) utilizzate in attacchi cibernetici.. 51

DevOps Metodologia che promuove la collaborazione e l'integrazione tra sviluppo *software* (Dev) e *operations* (Ops) all'interno di un'organizzazione; lo scopo

principale di DevOps è migliorare la velocità di sviluppo e rilascio del *software*, mantenendo nel contempo l'affidabilità, la stabilità e la sicurezza del sistema. Tale obiettivo viene raggiunto attraverso l'automazione dei processi di sviluppo, test e distribuzione, l'adozione di pratiche di monitoraggio continuo e il miglioramento continuo delle operazioni IT.. 1, 54

Generative AI Categoria di intelligenza artificiale che si concentra sulla creazione di contenuti originali, come immagini, testi, suoni o persino video, utilizzando algoritmi e modelli addestrati su grandi *set* di dati; a differenza dei modelli tradizionali che possono solo analizzare e rispondere a dati esistenti, l'intelligenza artificiale generativa può generare nuovi dati che sono coerenti con i modelli e le strutture presenti nei dati di addestramento.. iv, 1–3, 6, 7, 13–15, 17, 44, 46

Identity and Access Management Servizio AWS che consente di gestire in modo sicuro l'accesso ai servizi e alle risorse di AWS. IAM permette di implementare il controllo degli accessi basato sui ruoli (RBAC) e di assegnare politiche dettagliate per specificare esattamente quali azioni sono consentite o vietate per un determinato utente o gruppo.. 51

Internet of Things Interconnessione di dispositivi fisici, veicoli, elettrodomestici e altri oggetti dotati di sensori o *software* aventi la capacità di scambiare dati con altri dispositivi e sistemi attraverso Internet.. 51

Large Language Model Tipologia di modello di AI progettato per comprendere e generare testo con una capacità simile a quella umana; utilizzando reti neurali profonde, questi modelli apprendono regole linguistiche, coerenza del contesto e stili di scrittura da grandi quantità di dati testuali.. 51

Machine Learning Sottoinsieme dell'intelligenza artificiale che permette ai sistemi di apprendere e migliorare automaticamente dalle esperienze senza essere esplicitamente programmati; utilizzando algoritmi statistici e modelli matematici, il Machine Learning analizza grandi quantità di dati per identificare schemi e fare previsioni o decisioni basate su nuovi input.. 1, 53, 54

MITRE Organizzazione che gestisce progetti di ricerca e sviluppo in settori come la sicurezza nazionale, la difesa, l'aviazione civile, la sanità e altri ambiti tecnologici avanzati; è nota soprattutto per aver creato il CVE e il CWE, standard utilizzati globalmente per identificare e categorizzare vulnerabilità e debolezze *software*.. 7, 8, 13, 18

National Vulnerability Database Database gestito dal National Institute of Standards and Technology (NIST); fornisce informazioni sulle vulnerabilità di sicurezza note per una vasta gamma di *software* e *hardware*. Le vulnerabilità presenti nel NVD sono catalogate utilizzando un identificatore univoco, il CVE, che facilita la condivisione e l'aggregazione di informazioni tra diverse piattaforme e organizzazioni.. 51

Node Package Manager Gestore di pacchetti predefinito per *Node.js*; permette agli sviluppatori di condividere e riutilizzare codice, facilitando la gestione delle dipendenze nei progetti *JavaScript*. Fornisce un vasto *repository* online di pacchetti *open source*, accessibile tramite il comando `npm install`, che consente di integrare facilmente nuove funzionalità nei propri progetti. npm supporta anche script di automazione e configurazione di progetto attraverso il file `package.json`, dove sono definite le dipendenze e i meta-dati del progetto.. 51

Penetration testing Pratica mirata a valutare la sicurezza di un sistema informatico, di una rete o di un'applicazione simulando un attacco informatico reale; questo processo coinvolge l'uso di strumenti, tecniche e metodologie per identificare e sfruttare vulnerabilità potenziali che potrebbero essere utilizzate da *cyber attackers*.. 6, 55

Software Development Kit Libreria che consente agli sviluppatori di interagire e gestire i servizi di AWS utilizzando *JavaScript* o *TypeScript*; fornisce un'interfaccia semplice e strutturata per accedere ai servizi AWS direttamente dalle applicazioni *JavaScript*, consentendo di integrare facilmente funzionalità di *cloud computing* senza dover gestire direttamente la complessità dell'API sottostante.. 52

Sparring partner Individuo o gruppo di individui che collaborano con un'organizzazione per simulare attacchi informatici realistici e testare la resilienza di un sistema informatico di difesa; questo termine è spesso usato per descrivere esperti esterni o interni che agiscono come avversari simulati durante esercitazioni di sicurezza come il Penetration testing.. 6

Static Application Security Testing Metodologia di analisi utilizzata per identificare vulnerabilità di sicurezza all'interno del codice sorgente di un'applicazione; a differenza dei test dinamici, che analizzano il comportamento dell'applicazione in esecuzione, SAST esamina il codice staticamente, senza eseguirlo. Gli strumenti SAST scansionano il codice sorgente, il *bytecode* o il codice binario e forniscono un *report* dettagliato delle potenziali vulnerabilità, permettendo agli sviluppatori di correggere i problemi prima che l'applicazione venga distribuita.. 52

YAML Formato di serializzazione di dati utilizzato principalmente per la configurazione di file; utilizza l'indentazione per definire la struttura dei dati e supporta array, liste, e associazioni.. 9, 31

Bibliografia

Articoli

[1] Markus Bayer, Tobias Frey e Christian Reuter. «Multi-level fine-tuning, data augmentation, and few-shot learning for specialized cyber threat intelligence». In: *Computers & Security* 134 (2023), p. 103430.

[2] PV Charan et al. «From text to mitre techniques: Exploring the malicious use of large language models for generating cyber attack payloads». In: *arXiv preprint arXiv:2305.15336* (2023) (cit. a p. 7).

[3] Anton Cheshkov, Pavel Zadorozhny e Rodion Levichev. «Technical Report: Evaluation of ChatGPT Model for Vulnerability Detection». In: *arXiv preprint arXiv:2304.07232* (2023) (cit. a p. 8).

[4] Maanak Gupta et al. «From chatgpt to threatgpt: Impact of generative ai in cybersecurity and privacy». In: *IEEE Access* (2023) (cit. a p. 8).

[5] Andreas Happe e Jürgen Cito. «Getting pwn'd by ai: Penetration testing with large language models». In: (2023), pp. 2082–2086 (cit. a p. 6).

[6] Haonan Li et al. «The Hitchhiker's Guide to Program Analysis: A Journey with Large Language Models». In: *arXiv preprint arXiv:2308.00245* (2023) (cit. a p. 7).

[7] Xin Liu et al. «Not the end of story: An evaluation of ChatGPT-driven vulnerability description mappings». In: (2023), pp. 3724–3731 (cit. a p. 8).

[8] Forrest McKee e David Noever. «Chatbots in a honeypot world». In: *arXiv preprint arXiv:2301.03771* (2023) (cit. a p. 7).

[9] Shafi Parvez Mohammed e Gahangir Hossain. «Chatgpt in education, healthcare, and cybersecurity: Opportunities and challenges». In: (2024), pp. 0316–0321 (cit. a p. 7).

[10] Filippo Perrina et al. «Agir: Automating cyber threat intelligence reporting with natural language generation». In: (2023), pp. 3053–3062 (cit. a p. 7).

[11] Baptiste Roziere et al. «Code llama: Open foundation models for code». In: *arXiv preprint arXiv:2308.12950* (2023) (cit. a p. 46).

[12] Mark Scanlon et al. «ChatGPT for digital forensic investigation: The good, the bad, and the unknown». In: *Forensic Science International: Digital Investigation* 46 (2023), p. 301609 (cit. a p. 7).

[13] Tushar Sharma et al. «A survey on machine learning techniques for source code analysis». In: *arXiv preprint arXiv:2110.09610* (2021).

[14] Norbert Tihanyi et al. «The formai dataset: Generative ai in software security through the lens of formal verification». In: (2023), pp. 33–43.

[15] Norbert Tihanyi et al. «Cybermetric: A benchmark dataset for evaluating large language models knowledge in cybersecurity». In: *arXiv preprint arXiv:2402.07688* (2024) (cit. a p. 7).

[16] Yuqing Wang e Yun Zhao. «Gemini in reasoning: Unveiling commonsense in multimodal large language models». In: *arXiv preprint arXiv:2312.17661* (2023) (cit. a p. 7).

[17] Yagmur Yigit et al. «Review of generative AI methods in cybersecurity». In: *arXiv preprint arXiv:2403.08701* (2024) (cit. alle pp. 3, 6, 46).

Sitografia

[18] *Authenticating as a GitHub App installation*. URL: https://docs.github.com/en/apps/creating-github-apps/authenticating-with-a-github-app/authenticating-as-a-github-app-installation (cit. a p. 20).

[19] *Automatic token authentication*. URL: https://docs.github.com/en/actions/security-guides/automatic-token-authentication (cit. a p. 20).

[20] *Azure OpenAI Service - Pricing*. URL: https://azure.microsoft.com/en-us/pricing/details/cognitive-services/openai-service/ (cit. a p. 47).

[21] *Azure OpenAI Service - Quotas*. URL: https://learn.microsoft.com/it-it/azure/ai-services/openai/quotas-limits#gpt-4o-standard (cit. a p. 46).

[22] The MITRE Corporation. *CWE Version 4.14*. 2024. URL: https://cwe.mitre.org/data/published/cwe_latest.pdf (cit. a p. 15).

[23] *CVE Website*. URL: https://www.cve.org (cit. a p. 13).

[24] *CVSS v3.1 Specification Document*. URL: https://www.first.org/cvss/v3.1/specification-document (cit. a p. 15).

[25] *CWE - Common Weakness Enumeration*. URL: https://cwe.mitre.org/index.html (cit. alle pp. 13, 18, 58).

[26] *GitHub REST API documentation*. URL: https://docs.github.com/en/rest?apiVersion=2022-11-28 (cit. a p. 24).

[27] *Import a model with Custom Model Import*. URL: https://docs.aws.amazon.com/bedrock/latest/userguide/model-customization-import-model.html#:~:text=You%20can%20create%20a%20custom,that%20has%20proprietary%20model%20weights..

[28] *NVD - Home*. URL: https://nvd.nist.gov (cit. a p. 13).

[29] *OWASP Foundation*. URL: https://owasp.org (cit. a p. 14).

[30] *OWASP Top 10*. URL: https://owasp.org/www-project-top-ten/ (cit. a p. 14).

[31] *Quotas for Amazon Bedrock*. URL: https://docs.aws.amazon.com/bedrock/latest/userguide/quotas.html (cit. a p. 26).

[32] *Technology | Mistral AI*. URL: https://mistral.ai/technology/ (cit. a p. 45).

[33] *Tokenization | Mistral AI Large Language Models*. URL: https://docs.mistral.ai/guides/tokenization/ (cit. a p. 37).

Appendice A

Dettagli aggiuntivi

A.1 Selezione del modello

	CWE	G1 Premier	Instant 1.2	70B Instruct	Mistral Large
Prs.	**CWE22**	TP	FN	TP	TP
	CWE23	TP	TP	TP	TP
	CWE79	FN	FN	TP	TP
	CWE434	FN	TP	FN	TP
Not Prs.	CWE27	FP	TN	TN	TN
	CWE35	FP	TN	TN	FP
	CWE73	TN	FP	TN	TN
	CWE611	TN	TN	FP	TN
	CWE732	FP	TN	TN	TN

Tabella A.1: Performance di G1 Premier, Instant 1.2, 70B Instruct e Mistral Large su code snippet relativo a CWE22

La tabella raffigura l'esito (parziale) della *code review* eseguita controllando la presenza di 58 CWE (le CWE appartenenti alle categorie OWASP A01 e A02, mostrate nella tabella sopra) su un *code snippet* pertinente alla CWE22, recuperato dal sito del CWE [25]. La revisione è stata condotta utilizzando i modelli Titan Text G1 Premier di *Amazon*, Claude Instant 1.2 di *Anthropic*, Llama 3 70B Instruct di *Meta* e Mistral Large di *Mistral AI*. Delle 58 CWE utilizzate, la tabella riporta solamente quelle effettivamente presenti nel frammento di codice secondo il CWE (Prs.) e quelle individuate da uno o più modelli con un *False Positive* (FP) ma non presenti secondo il CWE (Not Prs.); le restanti CWE non sono state inserite in quanto correttamente individuate da tutti i modelli come non presenti con un *True Negative* (TN). Sono stati utilizzati 15 *code snippets* diversi per ultimare la valutazione delle *performance* dei modelli citati, relativi alle CWE: 22, 59, 201, 261, 319, 337, 352, 497, 540, 601, 706, 863, 913, 922 e 1390. I risultati complessivi di tali *performance* sono riassunti nella seguente tabella:

	G1 Premier	Instant 1.2	70B Instruct	Mistral Large	Total
TP	35	36	43	47	**161**
TN	5	6	12	15	**38**
FP	17	16	10	7	**50**
FN	15	14	7	3	**39**

Tabella A.2: Esito della code review

Avendo totalizzato il maggior numero di TP e TN e, allo stesso tempo, il minor numero di FP e FN, Mistral Large è stato selezionato per condurre la *code review* in versione integrale all'interno del VAS (come spiegato nella sezione 5.3).

A.2 Prompt engineering

I primi *screenshot* riportati mostrano l'utilizzo del *Playground* di Bedrock per determinare l'efficacia dei *prompt* selezionati durante la fase dedicata al *prompt engineering* della progettazione; in particolare, si può vedere il modello di richiesta inviata a Mistral Large per determinare la presenza o meno di un *batch* di 12 CWE (proprio come avviene in automatico all'interno del VAS) nel *code snippet* d'esempio relativo a CWE22. Come delinato nella tabella A.1, il modello individua correttamente la presenza delle CWE 22 e 23, e segnala anche la presenza della CWE35; le altre CWE del *batch* vengono correttamente etichettate come non presenti all'interno del codice. Da notare anche come il modello rispetti il formato specificato nel *prompt* in input nel strutturare la risposta data.

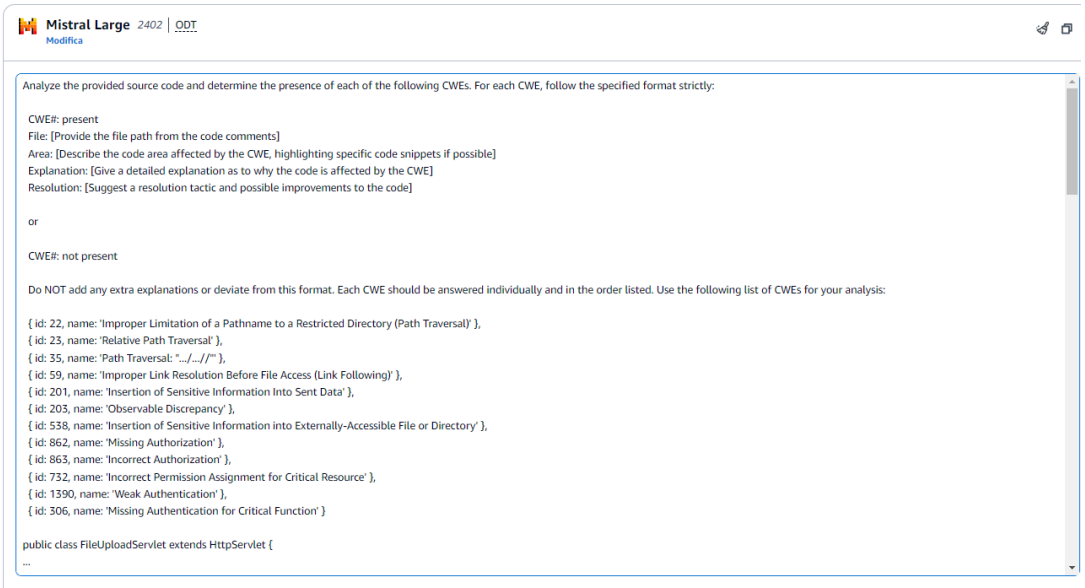


Figura A.1: CWECheck input: prompt per controllare un batch di CWE su code snippet relativo a CWE22 tramite Mistral Large

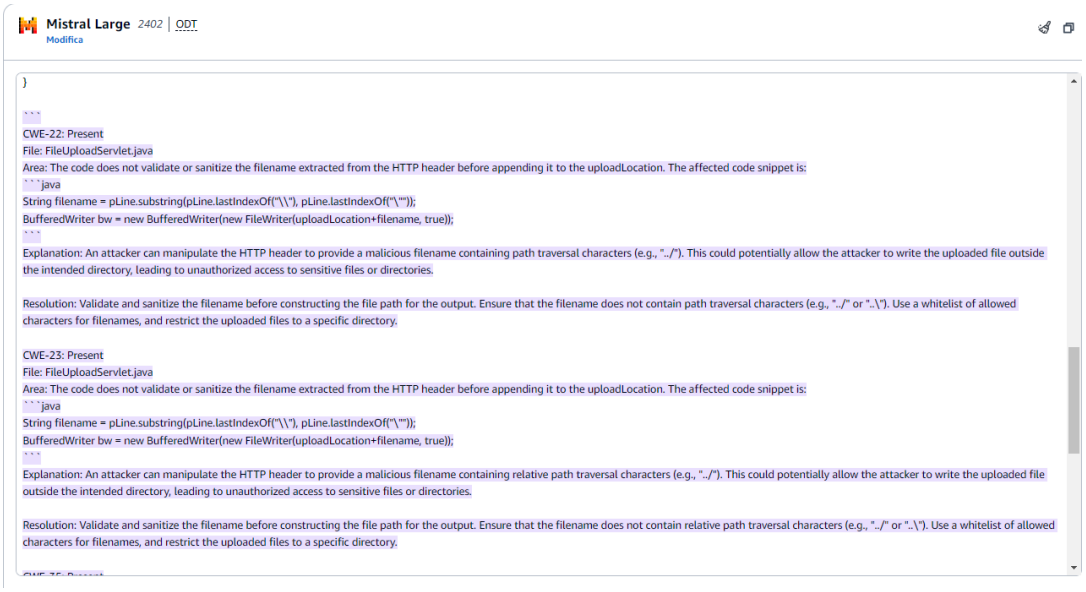


Figura A.2: CWECheck output: prima parte della risposta di Mistral Large

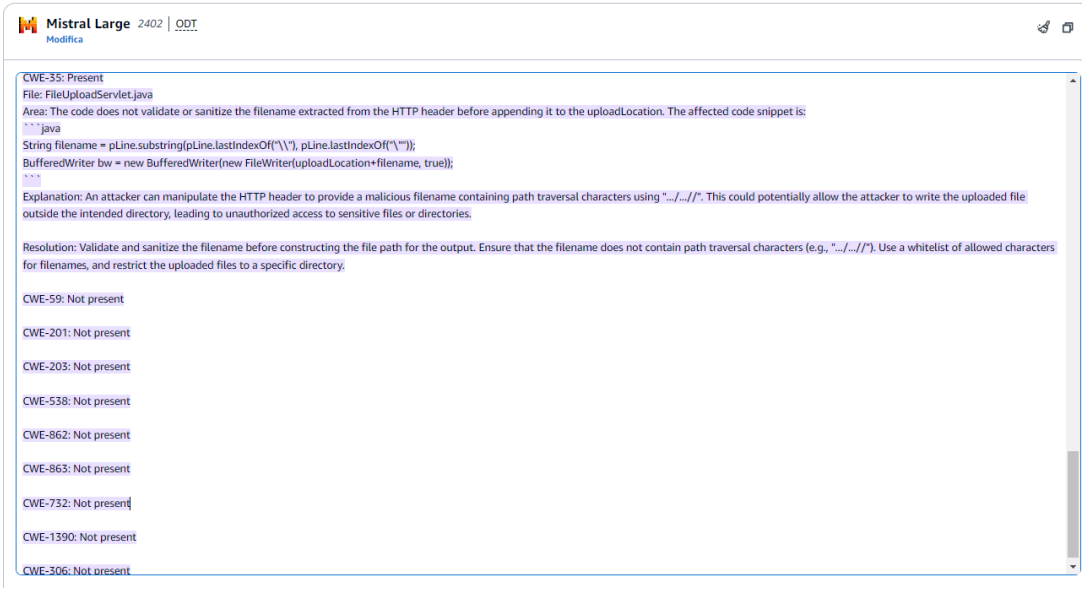


Figura A.3: CWECheck output: seconda parte della risposta di Mistral Large

Gli *screenshot* riportati successivamente, invece, mostrano fondamentalmente il seguito dell'interazione precedente, ossia il modello di richiesta inviata a Mistral Large per generare un CVSS Vector per ogni CWE rilevata poco prima. Anche in questo caso il modello si dimostra in grado di generare un vettore formattato come ci si aspetta per tutte le CWE, in modo da non impegnare il *parser* troppo a lungo.

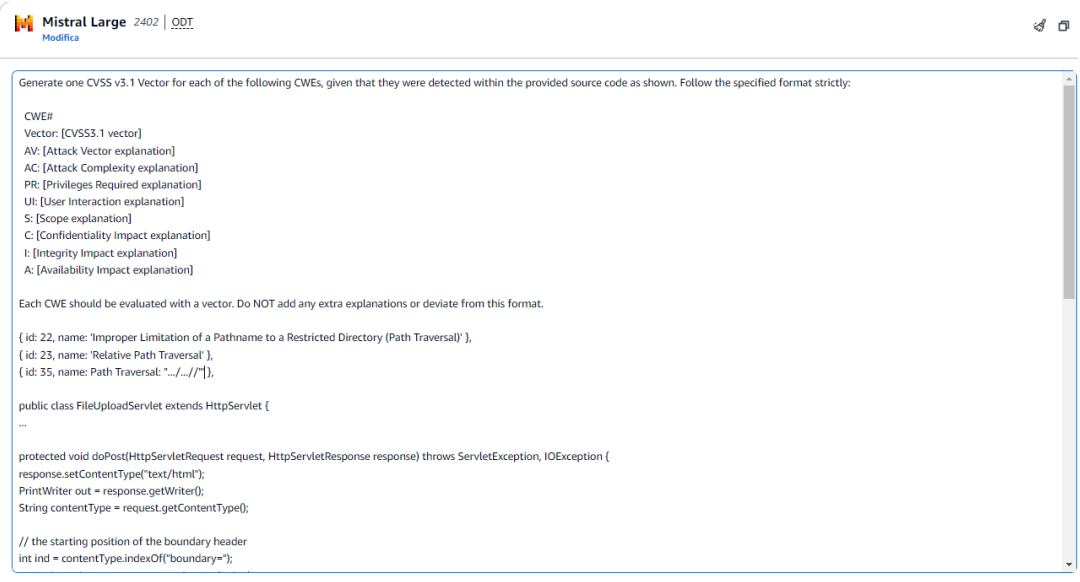


Figura A.4: CVSSVector input: prompt per generare un CVSS Vector per ogni CWE rilevata nel code snippet relativo a CWE22 tramite Mistral Large

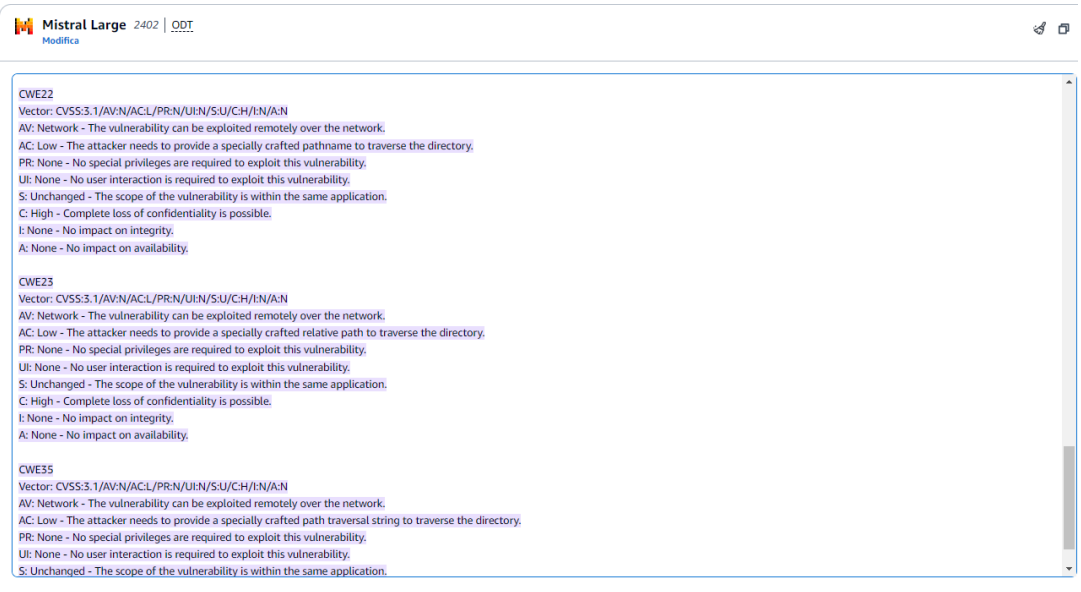


Figura A.5: CVSSVector ouput: risposta di Mistral Large

Va notato che i *prompt* mostrati nelle figure A.1 e A.4, per quanto ottimizzati, non chiedono al modello di utilizzare un formato esplicito nel strutturare le risposte (come potrebbe essere il formato `JSON`, ad esempio), il che potrebbe renderle più facili da analizzare con il *parser* apposito.

A.3 Gestione delle vulnerabilità

La seguente tabella riporta tutte le CWE appartenenti alle categorie OWASP Top10 2021, per un totale di 175 CWE:

OWASP A01	OWASP A02	OWASP A03	OWASP A04	OWASP A05	OWASP A06	OWASP A07	OWASP A08	OWASP A09	OWASP A10
CWE22	CWE261	CWE22	CWE183	CWE11	CWE1104	CWE259	CWE353	CWE117	CWE918
CWE23	CWE296	CWE1285	CWE287	CWE73		CWE288	CWE426	CWE223	
CWE35	CWE319	CWE1286	CWE312			CWE290	CWE494	CWE532	
CWE59	CWE321	CWE1287	CWE213			CWE294	CWE502	CWE778	
CWE201	CWE322	CWE1288	CWE256			CWE295	CWE565		
CWE203	CWE324	CWE1289	CWE275			CWE297	CWE784		
CWE538	CWE325	CWE77	CWE229			CWE301	CWE829		
CWE862	CWE326	CWE79	CWE314			CWE302	CWE830		
CWE863	CWE327	CWE89	CWE316			CWE304	CWE915		
CWE732	CWE328	CWE91	CWE311			CWE307			
CWE1390	CWE329	CWE94	CWE319			CWE346			
CWE306	CWE335	CWE95	CWE419			CWE384			
CWE219	CWE336	CWE96	CWE430			CWE521			
CWE276	CWE337	CWE97	CWE434			CWE613			
CWE352	CWE340	CWE90	CWE242			CWE620			
CWE359	CWE347	CWE91	CWE501			CWE621			
CWE377	CWE536	CWE98	CWE502			CWE640			
CWE402	CWE757	CWE99	CWE525			CWE641			
CWE425	CWE760	CWE184	CWE579			CWE643			
CWE441	CWE780	CWE184	CWE598			CWE644			
CWE497	CWE916	CWE471	CWE646			CWE652			
CWE540		CWE184	CWE653			CWE798			
CWE548		CWE471	CWE680			CWE799			
CWE552		CWE98	CWE798			CWE799			
CWE566		CWE599	CWE797			CWE841			
CWE601		CWE184	CWE841			CWE927			
CWE639		CWE599	CWE1173						
CWE651									
CWE693									
CWE732									
CWE1275									

Tabella A.3: Debolezze CWE appartenenti alle categorie OWASP Top 10 2021