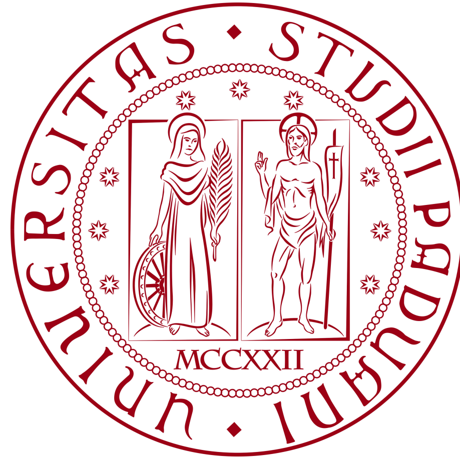
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Integrazione del servizio di Generative AI per la generazione di report di vulnerability assessment partendo dal codice sorgente

Tesi di Laurea

Relatore

Prof.ssa Gaggi Ombretta

Laureando

Samuele Visentin

Matricola 2034343

Anno Accademico 2023-2024

© Samuele Visentin, Settembre 2024. Tutti i diritti riservati. Tesi di Laurea: "*Integrazione del servizio di Generative AI per la generazione di report di vulnerability assessment partendo dal codice sorgente*", Università degli Studi di Padova, Dipartimento di Matematica "Tullio Levi-Civita".

Ringraziamenti

Desidero esprimere la mia gratitudine alla professoressa Gaggi Ombretta, mio relatore, per l'aiuto e il sostegno che mi ha dato durante la stesura dell'elaborato.

Vorrei anche ringraziare, con affetto, i miei genitori per aver creduto nelle mie capacità e per aver sostenuto economicamente il mio percorso di studio.

Desidero poi ringraziare i miei amici, in particolare Andrea con il quale ho collaborato per molti progetti universitari: grazie alla sua dedizione e impegno, abbiamo ottenuto risultati eccezionali.
Inoltre ringrazio Riccardo con il quale ho collaborato per il progetto di SWE, assieme a lui ho affrontato problematiche all'interno del gruppo, ma grazie al suo impegno siamo riusciti a concludere il progetto.

Infine vorrei ringraziare anche l'azienda zero12 che mi ha ospitato per svolgere lo stage, ringrazio il fantastico gruppo di persone che ho conosciuto e, in particolare, il mio tutor aziendale, il quale era sempre pronto a fornirmi un aiuto.

Padova, Settembre 2024 *Samuele Visentin*

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage del laureando Samuele Visentin, della durata di trecentoventi ore presso l'azienda zero12 s.r.l., nella sede di Padova.
Lo scopo del progetto è di realizzare un sistema che utilizza il servizio di *Artificial Intelligence (AI)*$_G$ di *OpenAI*$_G$, in particolare i modelli *Generative Pre-trained Transformer (GPT)*$_G$ per analizzare il codice sorgente di un applicativo software e generare un *report* con le eventuali vulnerabilità riscontrate, associando uno *score*.
In particolare per realizzare il progetto si è utilizzato *Amazon Web Service (AWS)*$_G$ per la gestione del servizio tramite *AWS*$_G$ Step Functions, *AWS*$_G$ *Simple Storage Service (S3)*$_G$ e *AWS*$_G$ Lambda. La prima fase dello stage prevedeva lo studio delle tecnologie da usare, successivamente si doveva studiare un *prompt* efficace per il modello *AI*$_G$ selezionato. La seconda fase invece riguardava l'implementazione del servizio nel seguente ordine:

1. Creazione di un sistema in grado di analizzare il codice sorgente e rilevare le vulnerabilità presenti

2. Generare un *report* con le vulnerabilità rilevate associando a ciascuna uno *score* che ne rilevi l'importanza e, inoltre, fornire dettagli aggiuntivi sull'area d'effetto e su una potenziale soluzione

3. Creare un sistema per automatizzare l'avvio del servizio

Infine, la fase finale del progetto era di valutare i *report* e i costi del particolare modello utilizzato per poter confrontare le differenze dei modelli *AI*$_G$.

Indice

1 Introduzione **1**

1.1 L'azienda 1

1.2 L'idea 1

1.3 Organizzazione del testo 2

2 Descrizione dello stage **4**

2.1 Introduzione al progetto 4

2.2 Analisi preventiva dei rischi 5

2.2.1 Complessità del contesto di utilizzo dei servizi di Generative AI 5

2.2.2 Ridotta conoscenza dei servizi cloud AWS 5

3 Obiettivi e requisiti **6**

3.1 Obiettivi formativi 6

3.2 Obiettivi del progetto 6

3.3 Requisiti del progetto 7

4 Generative AI **8**

4.1 Introduzione 8

4.2 OpenAI 8

4.3 Modelli GPT 9

4.4 Costi dei modelli 10

5 Tecnologie e strumenti **11**

5.1 Tecnologie utilizzate 11

5.1.1 Node.js 11

5.1.2 Framework serverless 11
5.1.3 TypeScript 12
5.1.4 AWS 12
5.1.4.1 AWS Lambda 12
5.1.4.2 AWS Step Functions 12
5.1.4.3 AWS S3 13
5.2 Strumenti utilizzati 13
5.2.1 NPM 13
5.2.2 Git 13
5.2.3 GitHub 14
5.2.3.1 GitHub Actions 14
5.2.3.2 GitHub Secrets 14
5.2.4 Visual Studio Code 14
5.2.5 ESLint 14
5.2.6 AWS Cli 15

6 Progettazione e architettura **16**
6.1 Progettazione 16
6.1.1 Individuazione e categorizzazione delle vulnerabilità . . . 16
6.1.2 Sistema di *scoring* CVSS 19
6.1.2.1 Struttura delle Metriche CVSS 19
6.1.2.2 Punteggio CVSS v3.1 22
6.1.3 Modelli GPT utilizzati 25
6.1.4 Prompt engineering 25
6.1.5 Integrazione del sistema con GitHub 28
6.2 Architettura 29
6.2.1 Architettura serverless 29
6.2.2 Architettura logica del sistema 30
6.2.2.1 GitHub Actions 31
6.2.2.2 Step Function 32

7 Implementazione del sistema **35**
7.1 Configurazione serverless 35

7.2 Configurazione GitHub Actions 37
7.3 Configurazione Step Function 39
7.3.1 Fase iniziale 40
7.3.1.1 Lambda: SplitChunks 40
7.3.2 Analisi del codice 40
7.3.2.1 Comunicazione con il modello 40
7.3.2.2 AnalyzeChunk 42
7.3.2.3 Lambda CheckIfRelevant 42
7.3.2.4 Choice state: IsRelevantChoice 42
7.3.2.5 SplitCWEs 43
7.3.2.6 Lambda ProccessCWEs 43
7.3.2.7 Choice state: CweChoice 45
7.3.2.8 Lambda GenerateVectorScore 45
7.3.3 Generazione del report 46

8 Uso del sistema 48
8.1 Impostazioni iniziali 48
8.2 GitHub Action 49
8.3 Avvio dell'analisi 49
8.4 Consultazione del report 49

9 Conclusioni 51
9.1 Valutazione dei modelli 51
9.1.1 Costi relativi all'esecuzione 52
9.2 Raggiungimento degli obiettivi 53
9.3 Conoscenze acquisite 54

Acronimi e abbreviazioni 56

Glossario 57

Bibliografia 61

Sitografia 63

Elenco delle figure

6.1 Flusso generale del sistema 30
6.2 Flusso della *Step Function* 34

7.1 Grafico dell'implementazione della *Step Function* 39

8.1 Esempio di come viene visualizzato il *file* report 50

Elenco delle tabelle

3.1 Tabella dei requisiti 7

6.1 Categorie di *Common Weakness Enumaration* nella *Open Web Application Security Project Top Ten* del 2021 18

Elenco dei codici sorgenti

6.1 Esempio di un *Common Vulnerability Scoring System vector* 3.1 23

6.2 *System prompt* per analizzare eventuali debolezze nel codice sorgente . . . 27

6.3 *System prompt* utilizzato per la generazione del vettore *Common Vulnerability Scoring System* 3.1 . . . 28

7.1 Configurazione del file "serveless.ts" . . . 35

7.2 Trigger dell'esecuzione del flusso . . . 37

7.3 Controllo del messaggio del *commit* . . . 37

7.4 *Upload* del codice e avvio della *Step Function* . . . 38

7.5 Codice che permette di fare una richiesta al modello *GPT* . . . 40

7.6 *System promot* per la rilevanza del blocco di codice . . . 42

7.7 Definizione da inserire nel file serverless.ts per definire lo stato IsRelevantChoice . . . 43

7.8 Espressione regolare per estrapolare le informazioni delle debolezze rilevate . . . 43

7.9 Definizione di CweChoice nel file serverless.ts . . . 45

7.10 Espressione regolare per estrarre il vettore *Common Vulnerability Scoring System* . . . 45

7.11 Parte del codice della Lambda "CreateReport" . . . 46

8.1 Impedire l'avvio dell'analisi del codice . . . 49

Capitolo 1

Introduzione

1.1 L'azienda

L'azienda zero12 s.r.l. si occupa di sviluppo software, in particolare è specializzata nei settori di *Cloud Computing*$_{\boldsymbol{G}}$, sviluppo di applicazioni *web* e *mobile*, utilizzo di *Internet of Things (IoT)*$_{\boldsymbol{G}}$ e di creazione di modelli *custom* di *Machine Learning*$_{\boldsymbol{G}}$. Nello specifico l'azienda è partner di *AWS*$_{\boldsymbol{G}}$, infatti il loro obiettivo è di sfruttare al meglio le tecnologie *cloud* scalabili autonomamente messe a disposizione da tale servizio.

1.2 L'idea

Negli ultimi due decenni, gli applicativi software hanno conosciuto una diffusione capillare, coinvolgendo quasi ogni ambito lavorativo e sociale. In particolare, l'interazione sociale ha visto l'emergere di numerosi siti web e applicazioni mobili, che permettono agli utenti di inserire le proprie informazioni personali per interagire con altri, condividendo post e pensieri. Anche nel contesto lavorativo, le applicazioni software sono diventate sempre più rilevanti; un esempio significativo è rappresentato dalle piattaforme di gestione dei progetti, che oggi rivestono un ruolo fondamentale per la collaborazione e l'organizzazione delle attività all'interno dei team. Per questo motivo è indispensabile garantire la sicurezza informatica di tali applicazioni, evitando la presenza di bug o vulnera-

bilità che potrebbero essere sfruttati per attacchi informatici, con conseguente divulgazione di dati sensibili degli utenti.
Pertanto, per prevenire tali rischi, è importante assicurarsi che le applicazioni sviluppate non presentino falle di sicurezza che potrebbero essere utilizzate per facilitare attacchi informatici. Lo scopo dello stage, infatti, era quello di sviluppare un sistema capace di rilevare eventuali vulnerabilità nel codice sorgente utilizzando un servizio di *Generative Artificial Intelligence (Generative AI)*$_{G}$.
Il continuo sviluppo delle tecnologie di *AI*$_{G}$, soprattutto negli ultimi anni, ha consentito loro di analizzare testo, ma anche di generare immagini, video, audio e si prevede che queste tecnologie possano essere utilizzate anche per assistere i programmatori, aiutandoli a evitare errori che potrebbero compromettere la sicurezza delle applicazioni software.

1.3 Organizzazione del testo

Il secondo capitolo introduce il progetto ed espone l'analisi preventiva dei rischi;

Il terzo capitolo descrive i requisiti e gli obiettivi richiesti del progetto;

Il quarto capitolo approfondisce l'argomento della *Generative AI*$_{G}$, nel dettaglio i modelli di *OpenAI*$_{G}$, i quali sono stati utilizzati per il progetto;

Il quinto capitolo elenca le tecnologie e strumenti utilizzati, spiegando la loro funzione nel progetto;

Il sesto capitolo espone la progettazione del sistema realizzato;

Il settimo capitolo approfondisce come è stato implementato il sistema;

L'ottavo capitolo descrive brevemente come usare il sistema, ovvero come iniziare una analisi del codice sorgente;

Il nono capitolo illustra le ultime le considerazioni del progetto, facendo particolare attenzione sui costi dei modelli.

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 dello stage

2.1 Introduzione al progetto

Il progetto ha l'obiettivo di sviluppare un sistema automatizzato per l'analisi del codice sorgente presente in una *repository* $_G$ su *GitHub* $_G$, al fine di individuare eventuali vulnerabilità di sicurezza. Questo sistema è in grado di generare un report dettagliato, che non solo elenca le vulnerabilità riscontrate, ma assegna anche uno *score* di gravità a ciascuna di esse, facilitando l'organizzazione per gli interventi correttivi.
La tecnologia utilizzata di *Generative AI* $_G$ sono i modelli *GPT* $_G$ di *OpenAI* $_G$, nello specifico: *GPT* $_G$-3.5-turbo, *GPT* $_G$-4, *GPT* $_G$-*4-omni (4o)* $_G$, e *GPT* $_G$-*4o* $_G$-mini. Questi modelli sono stati valutati per trovare un equilibrio tra accuratezza nella rilevazione delle vulnerabilità e costi operativi del sistema.
La piattaforma scelta per la gestione del servizio è *AWS* $_G$, che ha fornito l'infrastruttura scalabile e affidabile necessaria per l'implementazione del sistema.
I servizi *AWS* $_G$ utilizzati nel progetto sono: *AWS* $_G$ *S3* $_G$ per l'archiviazione dei dati, *AWS* $_G$ Step Functions per l'orchestrazione dei processi e *AWS* $_G$ Lambda per l'esecuzione delle funzioni *serverless*, garantendo così una gestione efficiente e scalabile delle operazioni.
Inoltre, si è utilizzato il *framework* $_G$ *serverless* $_G$ per facilitare la creazione, il *deployment* $_G$ e la gestione delle funzioni Lambda.
In sintesi, il progetto rappresenta una soluzione che utilizza l'intelligenza artifi-

ciale e il *Cloud Computing*$_G$ per la sicurezza del software, capace di analizzare il codice sorgente e individuare eventuali debolezze così da poter ridurre i rischi associati alle vulnerabilità.

2.2 Analisi preventiva dei rischi

L'analisi preventiva dei rischi è un elemento cruciale per il successo del progetto di stage, in quanto permette di identificare e mitigare i potenziali ostacoli che potrebbero compromettere il raggiungimento degli obiettivi prefissati. Di seguito vengono descritti i principali rischi identificati nel contesto dello stage, insieme alle relative strategie di mitigazione.

2.2.1 Complessità del contesto di utilizzo dei servizi di Generative AI

L'integrazione del servizio di *OpenAI*$_G$ per l'automazione del *vulnerability assessment* richiede una profonda comprensione delle tecnologie di *Generative AI*$_G$; la complessità tecnica deriva dalla necessità di garantire la coerenza e l'affidabilità dei risultati generati.
Per ridurre questo rischio, si è previsto un approfondimento preliminare sulla documentazione tecnica di *OpenAI*$_G$.

2.2.2 Ridotta conoscenza dei servizi cloud AWS

Sebbene il focus principale sia sull'utilizzo di *Generative AI*$_G$, la conoscenza dei servizi *AWS*$_G$ risulta necessaria per integrare diverse infrastrutture *cloud*; quindi la scarsa familiarità con *AWS*$_G$ rappresenta un ostacolo.
Per mitigare questo rischio si è previsto di dedicare del tempo per studiare i servizi di *AWS*$_G$ tramite la documentazione fornita dal servizio stesso. In caso di necessità, era possibile ricorrere al supporto del tutor aziendale che ha esperienza in *AWS*$_G$.

Capitolo 3

Obiettivi e requisiti

3.1 Obiettivi formativi

Gli obiettivi formativi dello *stage* sono i seguenti:

- Imparare a sviluppare in contesto innovativo come l'ambito dell'*AI* $_{\boldsymbol{G}}$
- Apprendere come sviluppare un applicativo software e come utilizzare i servizi di *AWS* $_{\boldsymbol{G}}$
- Imparare a gestire *task* e obiettivi giornalieri per rispettare il piano di lavoro
- Comprendere come integrare *AWS* $_{\boldsymbol{G}}$ con altri servizi

3.2 Obiettivi del progetto

Gli obiettivi richiesti per quanto riguarda al progetto sono:

- Individuare un sistema di classificazione delle potenziali vulnerabilità da individuare nel codice sorgente
- Individuare un metodo di classificazione delle vulnerabilità trovate, associando a ciascuna uno *score* che ne identifica la rilevanza
- Analizzare i diversi modelli messi a disposizione da *OpenAI* $_{\boldsymbol{G}}$ analizzando anche le differenze nei costi e tempi di risposta

- Implementare il sistema tramite i servizi di *AWS*$_G$
- Integrare il sistema con *GitHub*$_G$, in modo tale da poter analizzare le *repository*$_G$ innescando dei *trigger*
- Studio di un *prompt* efficace per ottenere una risposta dal modello in un formato specifico

3.3 Requisiti del progetto

Nella tabella 3.1 vengono elencati i requisiti funzionali del progetto, ovvero le funzionalità che devono essere presenti nel sistema.

Indice	*Descrizione*
1	Il sistema deve caricare il codice presente su *GitHub*$_G$ in un *bucket*$_G$ di *S3*$_G$
2	L'utente deve poter avviare l'analisi del codice eseguendo un *push* tramite *GitHub*$_G$
3	L'utente deve poter impedire l'avviamento del sistema aggiungendo il termine *skip* all'interno del messaggio del comando *commit*
4	Il sistema deve generare il report e caricarlo sulla *repository*$_G$ del progetto
5	L'utente deve poter definire il *branch* della *repository*$_G$ da analizzare, nel quale viene caricato anche il report
6	Il report deve contenere tutte le vulnerabilità trovate, con una spiegazione e possibile soluzione, associando a ciascuna vulnerabilità uno *score*
7	Il sistema deve utilizzare i modelli *GPT*$_G$ per l'analisi del codice sorgente

Tabella 3.1: Tabella dei requisiti

Capitolo 4

Generative AI

4.1 Introduzione

Negli ultimi anni, la *Generative AI*$_{\boldsymbol{G}}$ ha acquisito un ruolo di primaria importanza in diversi ambiti, tra cui la produzione di contenuti, la creazione artistica e, in modo sempre più crescente, nella sicurezza informatica. La capacità di generare testo, immagini e altri tipi di dati in modo autonomo ha aperto nuove frontiere per l'automazione e l'analisi, inclusa l'analisi del codice sorgente per la rilevazione di vulnerabilità.

Questo è particolarmente rilevante nella sicurezza informatica, dove la capacità di analizzare e generare un codice sicuro può prevenire vulnerabilità e mitigare rischi.

4.2 OpenAI

OpenAI è una delle organizzazioni leader nel campo della *Generative AI*$_{\boldsymbol{G}}$, è stata fondata nel 2015 e ha come missione quella di garantire che la *Generative AI*$_{\boldsymbol{G}}$ sia sicura e benefica per l'intera umanità; infatti, OpenAI, ha sviluppato i modelli noti come *GPT*$_{\boldsymbol{G}}$, ovvero modelli molto avanzati di *Generative AI*$_{\boldsymbol{G}}$.

I modelli *GPT*$_{\boldsymbol{G}}$, che stanno alla base di molte applicazioni moderne, sono progettati per comprendere e generare linguaggio naturale. Questi modelli sono stati addestrati su molti testi provenienti da Internet, consentendo loro di ac-

quisire una conoscenza approfondita delle strutture linguistiche e semantiche. Con il tempo, OpenAI ha continuato a migliorare questi modelli, aumentando la loro capacità di generare testo coerente e contestualmente rilevante.

4.3 Modelli GPT

Nel progetto vengono usati i modelli *GPT* **G** tramite le *Application Programming Interface (API)* **G** messe a disposizione da *OpenAI* **G**. In particolare viene usata la libreria *OpenAI API Node.js* **G**, ovvero una libreria messa a disposizione da *OpenAI* **G** che offre metodi per comunicare con i loro servizi senza dover interagire direttamente con le *API* **G**.
In una richiesta al modello *GPT* **G** si deve fornire un comando attraverso un *prompt*, ovvero un testo fornito al modello dal quale deve generare una risposta; nello specifico si definisce per ogni richiesta:

- *System prompt* che definisce le regole e le istruzioni che deve rispettare il modello, tramite questo *prompt* è possibile stabilire come deve rispondere ad un *user prompt*, modellando la risposta in un formato specifico
- *User prompt* che rappresenta la domanda dell'utente

Quindi è di fondamentale importanza utilizzare un *system prompt* adeguato, cosicché il modello rispetti il formato della risposta. Per questo motivo la parte iniziale dello *stage* è stata dedicata allo studio dei *prompt* per l'analisi del codice sorgente. Nello specifico, per studiare i modelli si è inizialmente usato il *playground* messo a disposizione dal sito web di *OpenAI* **G** nell'apposita sezione. Questa sezione permette di selezionare un modello in particolare e confrontarlo con un altro modello, in questo modo è stato possibile trovare un prompt efficace provando direttamente con degli esempi di prova, che hanno permesso di trovare un *system prompt* adatto; nel capitolo Architettura e progettazione viene analizzato nel dettaglio il flusso di utilizzo del modello *AI* **G**.

4.4 Costi dei modelli

I modelli di *OpenAI* $_{\boldsymbol{G}}$ prevedono costi di utilizzo che vengono calcolati in base ai *token*. Un *token* rappresenta una frazione di testo analizzata dal modello durante l'elaborazione del *prompt* di sistema e del *prompt* dell'utente. L'associazione tra il testo e i *token* non è sempre diretta: un singolo *token* può rappresentare più parole, oppure una singola parola può essere scomposta in più *token*, a seconda della complessità e della lunghezza.
Ogni modello applica un costo per i *token* utilizzati sia in ingresso (cioè nel *system prompt* e *user prompt*) sia in uscita (ovvero nella risposta generata dal modello). Inoltre, i costi variano in base al modello specifico: i modelli più avanzati e "intelligenti" hanno un costo per *token* maggiore rispetto a quelli meno complessi, riflettendo la loro maggiore capacità di elaborazione e comprensione [13].

Capitolo 5

Tecnologie e strumenti

In questo capitolo vengono elencate tutte le tecnologie e strumenti utilizzati per la realizzazione del progetto.

5.1 Tecnologie utilizzate

5.1.1 Node.js

Node.js è un *runtime open source* multipiattaforma che consente l'esecuzione di codice JavaScript al di fuori di un browser, rendendolo ideale per lo sviluppo di applicazioni lato server. Questa tecnologia permette di utilizzare JavaScript per sviluppare applicazioni server-side.
Inoltre, viene fornito anche con un ampio ecosistema di librerie e moduli esterni accessibili tramite *Node Package Manager (NPM)*$_G$.

5.1.2 Framework serverless

Il framework serverless è uno strumento *open source* progettato per semplificare lo sviluppo e la gestione di applicazioni *serverless*. In un'architettura *serverless*, l'infrastruttura di *backend* è completamente gestita dal fornitore di servizi *cloud*, come *AWS*$_G$, Google Cloud o Azure, permettendo agli sviluppatori di concentrarsi sulla scrittura del codice applicativo, senza preoccuparsi della gestione dei server o della scalabilità.

Infatti, il framework automatizza il processo di *deployment*$_G$ delle funzioni *serverless*, gestendo le configurazioni e permettendo di definire l'infrastruttura come codice attraverso *file* di configurazione. Questo approccio consente di specificare tutte le risorse necessarie, come funzioni Lambda, *API*$_G$ *Gateway*, *bucket*$_G$ *S3*$_G$, e altre risorse *cloud*, in un'unica definizione.

5.1.3 TypeScript

TypeScript è un linguaggio di programmazione *open source* sviluppato da Microsoft, che estende JavaScript aggiungendo la tipizzazione statica e il supporto per la programmazione orientata agli oggetti, tramite l'uso di classi e interfacce. Il linguaggio è stato progettato per migliorare la robustezza e la manutenibilità del codice, consentendo di rilevare errori durante la compilazione in JavaScript.

5.1.4 AWS

AWS (Amazon Web Services) è la piattaforma di *cloud computing* offerta da Amazon, la quale fornisce una vasta gamma di servizi e soluzioni *on-demand* per aziende e sviluppatori, permettendo di eseguire applicazioni e gestire dati senza la necessità di hardware fisico. Nel progetto sono stati utilizzati i seguenti servizi AWS: AWS Lambda, AWS Step Functions e AWS *S3*$_G$.

5.1.4.1 AWS Lambda

AWS Lambda è un servizio di *computing serverless* offerto da AWS, che permette agli sviluppatori di caricare il loro codice, specificando le condizioni in cui deve essere eseguito; Lambda si occuperà di gestire l'esecuzione e di scalare le risorse in base al carico di lavoro.

5.1.4.2 AWS Step Functions

AWS Step Functions è un servizio di orchestrazione per applicazioni distribuite offerto AWS, che consente di coordinare più servizi in flussi di lavoro

sequenziali e paralleli, rendendo più semplice la gestione di operazioni complesse che richiedono il coordinamento di vari passaggi e servizi; ovvero è possibile definire una *state machine*, la quale descrivere il comportamento di un sistema definendo una serie di stati e le transizioni tra questi stati in risposta a eventi o condizioni. Ogni stato rappresenta un'attività da svolgere e le transizioni indicano il passaggio da uno stato all'altro basato su delle condizioni.

5.1.4.3 AWS S3

Amazon S3 (Simple Storage Service) è un servizio di *storage* su *cloud* offerto da AWS, progettato per archiviare qualsiasi quantità di dati da qualsiasi parte del mondo. Il servizio S3 è scalabile e sicuro, rendendolo ideale per una vasta gamma di casi d'uso, come backup, archiviazione di dati e distribuzione di contenuti.
In questo servizio, i dati vengono memorizzati come oggetti all'interno di contenitori chiamati *bucket*$_{G}$, dove ogni oggetto è costituito da un *file* di dati e metadati associati ed è identificato univocamente da una chiave.

5.2 Strumenti utilizzati

5.2.1 NPM

NPM (Node Package Manager) è il gestore di pacchetti predefinito per Node.js. Facilita l'installazione, l'aggiornamento e la gestione delle dipendenze nei progetti JavaScript, permettendo agli sviluppatori di condividere e riutilizzare facilmente codice attraverso un vasto registro di pacchetti *open source*. Con semplici comandi da linea di comando, NPM semplifica l'integrazione di librerie e strumenti esterni, accelerando lo sviluppo e garantendo una gestione efficiente delle versioni all'interno delle applicazioni.

5.2.2 Git

Git è un sistema di controllo versione distribuito, ampiamente utilizzato per la gestione del codice sorgente in progetti software, consentendo agli sviluppatori

di tracciare le modifiche al codice, collaborare con altri e gestire diverse versioni di un progetto.

5.2.3 GitHub

GitHub è una piattaforma di *hosting* per il controllo versione e la collaborazione, basata su Git. La piattaforma mette a disposizione agli sviluppatori un ambiente online per archiviare e gestire *repository*$_{G}$ Git, facilitando la collaborazione su progetti software.

5.2.3.1 GitHub Actions

GitHub Actions è un servizio di automazione del flusso di lavoro integrato in GitHub, che consente agli sviluppatori di automatizzare operazioni come la *build*, il test e il *deployment*$_{G}$ del codice direttamente nei *repository*$_{G}$ GitHub, tramite l'utilizzo di *file* di configurazione YAML.

5.2.3.2 GitHub Secrets

GitHub Secrets è una funzionalità di GitHub che consente di gestire in modo sicuro le informazioni sensibili, come *token* di accesso, chiavi *API*$_{G}$, e altre credenziali, all'interno dei *repository*$_{G}$. Queste informazioni sensibili possono essere utilizzate nei flussi di lavoro di GitHub Actions senza esporle nel codice sorgente.

5.2.4 Visual Studio Code

Visual Studio Code (VS Code) è un editor di codice sorgente gratuito, leggero e multipiattaforma sviluppato da Microsoft, il quale offre supporto per molteplici linguaggi di programmazione attraverso estensioni.

5.2.5 ESLint

ESLint è uno strumento di *linting*$_{G}$ per il linguaggio JavaScript, progettato per identificare e per correggere eventuali problemi nel codice sorgente, aiutando

gli sviluppatori a mantenere standard di qualità elevati.

5.2.6 AWS Cli

AWS CLI (Command Line Interface) è uno strumento che permette di gestire e interagire con i servizi di Amazon Web Services direttamente dal terminale.

Capitolo 6

Progettazione e architettura

Questo capitolo tratta della progettazione del progetto, che si suddivide in individuazione delle vulnerabilità e categorizzare tali vulnerabilità per generare uno *score* per ognuna. Successivamente si descrive l'architettura del servizio *cloud* utilizzato.

6.1 Progettazione

Nel processo di progettazione del sistema, uno degli aspetti cruciali è stato l'individuazione e la categorizzazione delle vulnerabilità nel codice sorgente. Questo passaggio è essenziale per garantire che le debolezze identificate possano essere gestite in modo sistematico e coerente, consentendo una successiva analisi approfondita e la generazione di un report dettagliato.

6.1.1 Individuazione e categorizzazione delle vulnerabilità

Il primo passo nella progettazione del sistema consiste non solo nel rilevare le vulnerabilità nel codice sorgente, ma anche nel classificarle in modo appropriato. Per raggiungere questo obiettivo, è stato adottato il sistema di categorizzazione *Common Weakness Enumeration (CWE)*$_G$, sviluppato e mantenuto da *MITRE*$_G$. Il *CWE*$_G$ è un catalogo standardizzato e ampiamente riconosciuto che descrive le debolezze comuni del software, ciascuna identificata da un *ID*

univoco. Ogni voce nel catalogo *CWE*$_G$ rappresenta una specifica debolezza che potrebbe condurre a una vulnerabilità sfruttabile, come *buffer overflow*, *injection* o gestione impropria della memoria.
Nel contesto di questo progetto, il catalogo *CWE*$_G$ riveste un ruolo centrale, il modello di intelligenza artificiale viene utilizzato per riconoscere specifici *ID* *CWE*$_G$. Questi *ID* vengono specificati nel *system prompt*, un'istruzione iniziale che definisce le regole e le linee guida che il modello *GPT*$_G$ di *OpenAI*$_G$ deve seguire durante l'elaborazione delle richieste. Quando il codice sorgente viene fornito come *input* attraverso l'*user prompt*, il modello *GPT*$_G$ analizza il codice alla ricerca delle debolezze corrispondenti agli *ID* *CWE*$_G$ specificati.
Per affinare ulteriormente il processo di categorizzazione, si è deciso di adottare la *Open Web Application Security Project (OWASP)*$_G$ *Top Ten* come quadro di riferimento principale, ovvero una lista delle dieci categorie di vulnerabilità più critiche nelle applicazioni web. Questo elenco rappresenta una sintesi delle minacce di sicurezza più rilevanti e comuni nel panorama delle applicazioni web moderne.
La decisione di utilizzare la *OWASP*$_G$ *Top Ten* è stata motivata dal fatto che essa è strettamente correlata al catalogo *CWE*$_G$. Ogni categoria di vulnerabilità presente nella *Top Ten* è mappata a una o più voci del *CWE*$_G$, il che fornisce una base solida e standardizzata per la classificazione delle debolezze identificate dal modello *AI*$_G$. Questo approccio consente al sistema di focalizzarsi sulle vulnerabilità più rilevanti, garantendo al contempo che la classificazione delle debolezze sia allineata agli standard internazionali riconosciuti.
Di seguito viene riportata la tabella n. 6.1 con le categoria delle *CWE*$_G$ presenti nella lista del 2021 utilizzate per l'analisi del codice.

Categoria	**Nome**	**Descrizione**
A01	*Broken Access Control*	*Broken Access Control* tratta delle debolezze che permettono di accedere a sezioni o dati senza possedere l'autorizzazione necessaria

Continua nella prossima pagina...

Tabella 6.1 – Continuo della tabella

Categoria	Nome	Descrizione
A02	*Cryptographic Failures*	Rappresenta la gestione errata delle tecniche crittografiche
A03	*Injection*	Con *injection* si intendono le debolezze che permettono l'esecuzione di comandi senza possedere i permessi necessari
A04	*Insecure Design*	Si riferisce a debolezze nel design dell'applicazione che portano a vulnerabilità di sicurezza
A05	*Security Misconfiguration*	Rappresenta l'errore nel configurare correttamente le impostazioni di sicurezza di un'applicazione o di un'infrastruttura
A06	*Vulnerable and Outdated Components*	L'utilizzo di componenti software con vulnerabilità conosciute
A07	*Identification and Authentication Failures*	Debolezze nel processo di autenticazione e gestione delle sessioni che possono portare ad un accesso non autorizzato
A08	*Software and Data Integrity Failures*	Problemi legati all'integrità del software e dei dati
A09	*Security Logging and Monitoring Failures*	Mancanza o insufficienza di monitoraggio software, che può ostacolare la rilevazione di vulnerabilità o errori
A10	*Server-Side Request Forgery*	Debolezze che permettono ad un attaccante di manipolare il *server* per effettuare richieste non autorizzate

Tabella 6.1: Categorie di *Common Weakness Enumaration* nella *Open Web Application Security Project Top Ten* del 2021

6.1.2 Sistema di *scoring* CVSS

Una volta analizzato il codice sorgente tramite le *CWE*$_G$ è necessario utilizzare un sistema che permetta di associare uno *score* alle debolezze trovate; per fare ciò si è scelto di utilizzare il seguente sistema.
Il *Common Vulnerability Scoring System (CVSS)*$_G$ è uno standard aperto utilizzato per valutare la gravità delle vulnerabilità nel software, fornisce un modo coerente e universale per misurare e confrontare la gravità delle vulnerabilità, aiutando le organizzazioni a determinare la priorità delle correzioni e delle misure di mitigazione.

6.1.2.1 Struttura delle Metriche CVSS

Il CVSS si compone di tre gruppi principali di metriche:

1. **Metriche di Base (Base Metrics)**
2. **Metriche Temporali (Temporal Metrics)**
3. **Metriche Ambientali (Environmental Metrics)**

1. Metriche di Base (Base Metrics)

Le metriche di base rappresentano le caratteristiche intrinseche di una vulnerabilità che rimangono costanti nel tempo e tra diversi ambienti. Queste metriche sono fondamentali per il calcolo del punteggio *CVSS*$_G$, poiché descrivono la gravità immediata della vulnerabilità. Le metriche di base si suddividono in:

- **Attack Vector (AV)**: Definisce la distanza o la complessità dell'accesso per sfruttare la vulnerabilità.
  - **Network (N)**: Lo sfruttamento è possibile da remoto tramite la rete.
  - **Adjacent (A)**: Lo sfruttamento richiede l'accesso a una rete limitrofa, come una sottorete.
  - **Local (L)**: Lo sfruttamento richiede accesso locale al sistema.
  - **Physical (P)**: Lo sfruttamento richiede accesso fisico al dispositivo.

- **Attack Complexity (AC)**: Misura la difficoltà di sfruttare la vulnerabilità, indipendentemente dall'attaccante.
  - **Low (L)**: Lo sfruttamento è semplice e non richiede particolari condizioni.
  - **High (H)**: Lo sfruttamento è più complesso e richiede circostanze specifiche.
- **Privileges Required (PR)**: Indica il livello di privilegi richiesti per sfruttare la vulnerabilità.
  - **None (N)**: Nessun privilegio richiesto.
  - **Low (L)**: Sono richiesti privilegi limitati.
  - **High (H)**: Sono richiesti privilegi elevati.
- **User Interaction (UI)**: Indica se l'attaccante ha bisogno di un'interazione con l'utente per sfruttare la vulnerabilità.
  - **None (N)**: Nessuna interazione necessaria.
  - **Required (R)**: L'attacco richiede l'interazione dell'utente.
- **Scope (S)**: Determina se lo sfruttamento della vulnerabilità può influenzare altre componenti al di fuori del controllo del sistema vulnerabile.
  - **Unchanged (U)**: Lo sfruttamento non influisce su altre componenti.
  - **Changed (C)**: Lo sfruttamento può influenzare altre componenti.
- **Impatto sulla Riservatezza (Confidentiality Impact, C)**: Misura l'impatto sulla riservatezza dei dati a causa dello sfruttamento.
  - **None (N)**: Nessun impatto.
  - **Low (L)**: Impatto parziale.
  - **High (H)**: Impatto significativo.
- **Impatto sull'Integrità (Integrity Impact, I)**: Misura l'impatto sull'integrità dei dati.

- **None (N)**: Nessun impatto.
- **Low (L)**: Impatto parziale.
- **High (H)**: Impatto significativo.

- **Impatto sulla Disponibilità (Availability Impact, A)**: Misura l'impatto sulla disponibilità del sistema.
  - **None (N)**: Nessun impatto.
  - **Low (L)**: Impatto parziale.
  - **High (H)**: Impatto significativo.

2. Metriche Temporali (Temporal Metrics)

Le metriche temporali rappresentano le caratteristiche di una vulnerabilità che possono cambiare nel tempo, influenzando così il punteggio *CVSS*$_G$. Queste metriche permettono di riflettere la maturità del codice di *exploit* e la disponibilità delle correzioni. Le metriche temporali sono:

- **Exploit Code Maturity (E)**: Misura la disponibilità e l'efficacia del codice di exploit.
  - **Not Defined (X)**
  - **Unproven (U)**
  - **Proof-of-Concept (P)**
  - **Functional (F)**
  - **High (H)**
- **Remediation Level (RL)**: Misura l'efficacia delle misure di mitigazione disponibili.
  - **Not Defined (X)**
  - **Official Fix (O)**
  - **Temporary Fix (T)**
  - **Workaround (W)**

  - **Unavailable (U)**

- **Report Confidence (RC)**: Indica la qualità e l'affidabilità delle informazioni sulla vulnerabilità.
  - **Not Defined (X)**
  - **Unknown (U)**
  - **Reasonable (R)**
  - **Confirmed (C)**

3. Metriche Ambientali (Environmental Metrics)

Le metriche ambientali rappresentano le caratteristiche di una vulnerabilità che possono variare a seconda dell'ambiente specifico in cui si trova il sistema. Queste metriche permettono di adattare il punteggio *CVSS*$_G$ alla realtà operativa specifica di un'organizzazione. Le metriche ambientali includono:

- **Confidentiality Requirement (CR)**, **Integrity Requirement (IR)**, **Availability Requirement (AR)**: Misurano l'importanza della riservatezza, integrità e disponibilità dei dati nel contesto specifico.
  - **Not Defined (X)**
  - **Low (L)**
  - **Medium (M)**
  - **High (H)**
- **Modified Base Metrics**: Permettono di modificare le metriche di base in base al contesto specifico. Questa caratteristica è utile per calcolare un punteggio che rifletta più accuratamente l'impatto di una vulnerabilità nell'ambiente in cui il sistema è realmente utilizzato.

6.1.2.2 Punteggio CVSS v3.1

Il punteggio *CVSS*$_G$ finale è determinato combinando le metriche di base, temporali e ambientali, con un valore che varia da 0 a 10. Un punteggio più

alto indica una vulnerabilità più grave. La scala interpretativa del punteggio è la seguente:

- **0.0**: Nessun rischio
- **0.1 - 3.9**: Basso
- **4.0 - 6.9**: Medio
- **7.0 - 8.9**: Alto
- **9.0 - 10.0**: Critico

Nel contesto di questo progetto, si è scelto di adottare solo le metriche di base per la generazione del vettore *CVSS* ***G***. In particolare, il modello *GPT* ***G***, una volta ricevute in *input* le debolezze identificate attraverso le *CWE* ***G***, dovrà produrre il *CVSS* ***G*** *Vector*. Solo le metriche base sono state utilizzate perché il modello *AI* ***G*** non è in grado di considerare il contesto specifico di una vulnerabilità. Ciò significa che il suo approccio si limita alla valutazione delle caratteristiche tecniche e intrinseche della vulnerabilità, senza tenere conto di fattori esterni, come il contesto operativo o ambientale in cui la vulnerabilità potrebbe essere sfruttata.
Un esempio di un vettore *CVSS* ***G*** che il sistema potrebbe generare è riportato di seguito:

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

Listing 6.1: Esempio di un *Common Vulnerability Scoring System vector* 3.1

Come si può osservare, nel vettore è indicata la versione del *CVSS* ***G*** 3.1, e ogni metrica è separata da una barra inclinata ("/"). Le metriche sono identificate da sigle seguite da un valore, diviso da due punti (":"). In questo esempio si può notare che le metriche e i valori specificati sono i seguenti:

1. **AV:L** (*Attack Vector*: *Local*): indica che l'attaccante deve avere accesso fisico o una connessione diretta al sistema vulnerabile. È più difficile da sfruttare rispetto a una vulnerabilità accessibile via rete

2. **AC:H** (*Attack Complexity*: *High*): significa che l'attacco è complesso e richiede condizioni particolari, come conoscenze tecniche avanzate o risorse specifiche per essere eseguito con successo

3. **PR:L** (*Privileges Required*: *Low*): l'attaccante, per eseguire l'attacco informatico, non necessita dei privilegi da amministratore

4. **UI:N** (*User Interaction*: *None*): non è richiesta alcuna interazione da parte dell'utente affinché l'attacco sia efficace. La vulnerabilità può essere sfruttata senza che l'utente esegua alcuna azione

5. **S:U** (*Scope*: *Unchanged*): indica che lo sfruttamento della vulnerabilità non provoca cambiamenti nel contesto di sicurezza; l'attacco rimane confinato all'ambito del sistema vulnerabile

6. **C:H** (*Confidentiality Impact*: *High*): lo sfruttamento della vulnerabilità causa una significativa perdita di riservatezza, consentendo all'attaccante l'accesso a dati sensibili o riservati

7. **I:H** (*Integrity Impact*: *High*): l'integrità del sistema è gravemente compromessa. L'attaccante può modificare o distruggere dati critici, influenzando l'affidabilità delle informazioni

8. **A:H** (*Availability Impact*: *High*): la disponibilità del sistema è seriamente danneggiata, con conseguente interruzione o rallentamento significativo del servizio o delle funzionalità del sistema

Il modello AI_G genera solo il vettore $CVSS_G$, mentre lo *score* viene calcolato successivamente a partire da tale vettore. Questa separazione garantisce una maggiore consistenza tra il vettore e il punteggio finale, poiché l'utilizzo di un *system prompt* ben definito per la generazione del vettore migliora l'accuratezza della risposta.
La scelta di basarsi solo sulle metriche base è importante, poiché fornisce una valutazione standardizzata e universalmente riconosciuta della vulnerabilità, senza la necessità di considerare variabili esterne che il modello non è in grado di processare. Questo approccio consente anche di mantenere una maggiore precisione

nella gestione del rischio, affidandosi a un sistema che genera risultati più consistenti.
La versione 3.1 del *CVSS* $_G$ introduce miglioramenti significativi rispetto alle versioni precedenti, con una maggiore precisione nella valutazione delle metriche base, garantendo un sistema di punteggio affidabile e adattabile alle vulnerabilità attuali.

6.1.3 Modelli GPT utilizzati

Per il progetto, si è deciso di implementare un sistema che potesse sfruttare i modelli più recenti di *OpenAI* $_G$, permettendo il confronto tra i report generati dai diversi modelli. Inizialmente, i modelli considerati erano i seguenti:

1. *GPT* $_G$-4 turbo
2. *GPT* $_G$-4
3. *GPT* $_G$-*4o* $_G$
4. *GPT* $_G$-3.5 turbo

Questi modelli sono elencati in ordine di "intelligenza", ovvero complessità e capacità di comprensione e il modello *GPT* $_G$-4 turbo viene considerato il più avanzato. Tuttavia, durante lo sviluppo del progetto, *OpenAI* $_G$ ha rilasciato aggiornamenti che hanno introdotto un nuovo modello: *GPT* $_G$-*4o* $_G$-mini. Questo modello ha sostituito completamente *GPT* $_G$-3.5 turbo, poiché offre un costo inferiore per lo stesso numero di *token*, mantenendo al contempo una maggiore "intelligenza" rispetto al suo predecessore [12]. Anche *GPT* $_G$-*4o* $_G$ ha subito aggiornamenti, sostituendo *GPT* $_G$-4 turbo come il modello più avanzato di *OpenAI* $_G$, ma a un costo inferiore, pur mantenendo la stessa capacità di elaborazione [11].

6.1.4 Prompt engineering

Uno degli aspetti centrali del progetto è stato lo sviluppo di un *system prompt* efficace. Come descritto nel capitolo 4.3, il *system prompt* è una parte cruciale

della comunicazione con i modelli di *Generative AI*$_G$, poiché definisce il modo in cui il modello risponderà all'*user prompt*, cioè alla richiesta dell'utente. È essenziale formulare un *system prompt* chiaro e strutturato per ottenere risposte in un formato che possa essere elaborato automaticamente e utilizzato per generare report dettagliati sulle vulnerabilità individuate nel codice.

Inizialmente, si era tentato di richiedere al modello di generare direttamente il punteggio (*score*) delle vulnerabilità fornendo il codice sorgente come *input* nel *user prompt*. Tuttavia, questo approccio si è rivelato inefficace perché la richiesta era troppo generica. Un compito mal definito porta spesso a risposte incoerenti e poco accurate da parte dei modelli di intelligenza artificiale. La qualità delle risposte migliora sensibilmente quando il modello ha un compito specifico e ben delimitato [8].

Per risolvere questo problema, si è optato per una suddivisione del processo di generazione del report in due fasi distinte:

1. **Analisi delle debolezze**: In questa prima fase, il modello analizza il codice sorgente per identificare le debolezze utilizzando le *CWE*$_G$. Il *system prompt* chiede esplicitamente al modello di rilevare le debolezze, senza fornire una lista di *CWE*$_G$ specifiche. L'output generato include le vulnerabilità individuate, accompagnate da una descrizione delle stesse.

2. **Generazione del vettore CVSS**: Nella seconda fase, utilizzando le descrizioni generate dal primo passo, si chiede al modello di creare un vettore *CVSS*$_G$ 3.1. Il vettore viene successivamente utilizzato per calcolare lo *score* associato a ciascuna vulnerabilità.

Tuttavia, questo approccio ha mostrato alcune limitazioni, specialmente con i modelli meno avanzati. In alcuni casi, l'*AI*$_G$ tendeva ad associare vulnerabilità a descrizioni di *CWE*$_G$ errate o generava risultati incoerenti, ad esempio indicando vulnerabilità non presenti nel codice. Questo problema era causato dal fatto che, lasciando troppa libertà al modello, si correva il rischio di ottenere risposte non accuratamente focalizzate sulle debolezze reali.

Per migliorare la coerenza del sistema, si è scelto di fornire una lista precisa di *CWE*$_G$ nel *system prompt*, insieme ai rispettivi *ID* e descrizioni. In questo

modo, il modello doveva semplicemente verificare la presenza o meno di queste debolezze specifiche nel codice sorgente. L'output dell'analisi delle debolezze quindi include l'area di codice vulnerabile, il CWE_G associato e una breve spiegazione della vulnerabilità.

Il *system prompt* dell'analisi delle debolezze, illustrato nel codice n. 6.2, è stato progettato per guidare il modello nell'analisi del codice sorgente, associando eventuali debolezze rilevate alle corrispondenti CWE_G, chiedendo al modello di fornire una descrizione tecnica e suggerendo possibili risoluzioni.

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

CWE[cwe id]: present
File: Provide the file path from the code comments
Area: Describe the code area affected by the CWE and
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[cwe id]: not present

Do NOT add any extra explanations or deviate from
this format. You MUST answer every CWE and in order
listed. Use the following list of CWEs for your
analysis:

*lista delle CWE con id e nome relativi*

Do NOT add any extra CWE and do NOT use any CWE more
then once if they aren't present. Follow strictly the
previous format only
```

Listing 6.2: *System prompt* per analizzare eventuali debolezze nel codice sorgente

In questo modo, il codice sorgente viene fornito nel *user prompt*, e il modello di intelligenza artificiale è incaricato di analizzarlo utilizzando solo il *set* di CWE_G specificato. Il risultato finale contiene le vulnerabilità rilevate, se presenti, con le informazioni richieste.

Una volta identificate le debolezze, si passa alla generazione del vettore *CVSS*$_G$, il *system prompt* per la generazione del vettore *CVSS*$_G$ è presente nel codice 6.3.

```
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

Every CWE must be evaluated with a vector, even if
the CWE id is already present. Do NOT add any extra
explanations or deviate from this format.
```

Listing 6.3: *System prompt* utilizzato per la generazione del vettore *Common Vulnerability Scoring System* 3.1

In questa fase, le *CWE*$_G$ rilevate vengono inserite nell'*user prompt* e il modello di *AI*$_G$ genera il vettore *CVSS*$_G$, che poi verrà utilizzato per calcolare il punteggio della vulnerabilità.

6.1.5 Integrazione del sistema con GitHub

Un passo fondamentale del progetto è stato collegare il sistema di analisi del codice sorgente alla *repository*$_G$ del progetto ospitata su *GitHub*$_G$. Per implementare questa integrazione, si è deciso di utilizzare *GitHub*$_G$ *Actions*, uno strumento che consente di automatizzare flussi di lavoro direttamente all'interno della *repository*$_G$. *GitHub*$_G$ *Actions* permette di definire una serie di azioni da eseguire al verificarsi di determinati eventi, come un *push* (invio del codice dalla *repository*$_G$ locale a quella remota), la creazione di una *pull request* o l'esecuzione di un *merge*.

Per integrare il sistema di analisi con *GitHub*$_G$, è necessario configurare correttamente le credenziali di accesso a due servizi chiave:

1. *AWS*$_G$: poiché il sistema di analisi del codice sorgente risiede su un'infrastruttura ospitata in *AWS*$_G$, è fondamentale fornire a *GitHub*$_G$ le credenziali per accedere a questo servizio in modo sicuro. Queste credenziali consistono in un *access key* e un *access id*, che devono essere ottenuti dall'account *AWS*$_G$.

2. *GitHub*$_G$ *Access Token*: per poter interagire con la *repository*$_G$, come leggere il codice sorgente e eseguire modifiche automatizzate, è necessario un *access token* con i permessi appropriati. Questo token deve avere i privilegi per leggere, scrivere e gestire le operazioni nella *repository*$_G$.

Entrambi questi valori (l'*access key* e l'*access id* di *AWS*$_G$, insieme al *access token* di *GitHub*$_G$) devono essere memorizzati nella sezione *Secrets* all'interno della *repository*$_G$. *GitHub*$_G$ *Secrets* è una funzionalità che consente di salvare e gestire in modo sicuro le credenziali sensibili, evitando che vengano esposte pubblicamente. Una volta salvate, queste credenziali possono essere utilizzate nelle *GitHub*$_G$ *Actions* per autenticare e autorizzare le operazioni.

6.2 Architettura

Per la realizzazione del sistema di analisi si è utilizzata un'architettura *serverless*, che permette di implementare il progetto senza dover gestire l'infrastruttura sottostante, quindi utilizzando servizi *cloud*, prevalentemente tramite *AWS*$_G$.

6.2.1 Architettura serverless

Un'architettura *serverless* è un modello di progettazione che consente di sviluppare ed eseguire applicazioni senza dover gestire i server, delegando l'infrastruttura sottostante a un provider *cloud*. In questo contesto, il *framework*$_G$ *serverless*$_G$ e i servizi di *AWS*$_G$ giocano un ruolo centrale nel garantire la scalabilità, la flessibilità e l'efficienza del sistema.

Questo progetto sfrutta i servizi gestiti di *AWS*$_G$ per eseguire codice e orchestrare flussi di lavoro senza la necessità di gestire direttamente i server. I componenti *AWS*$_G$ utilizzati sono elencati nel capitolo 5.1.4.
Per la gestione e il *deployment*$_G$ dell'applicazione *AWS*$_G$ è stato utilizzato il *framework*$_G$ *serverless*$_G$ che offre un approccio dichiarativo per definire le risorse dell'infrastruttura.
I vantaggi nel adottare un'architettura *serverless* sono i seguenti:

- Scalabilità automatica: uno dei principali benefici è la scalabilità automatica. *AWS*$_G$ Lambda scala automaticamente in base alla domanda, eliminando il problema di *over-provisioning* o *under-provisioning di risorse*
- Costi ridotti: il modello di pagamento a consumo dei servizi *AWS*$_G$ garantisce che si paghi solo per l'uso effettivo delle risorse
- Gestione semplificata: con *AWS*$_G$ che si occupa della gestione dell'infrastruttura sottostante, gli sviluppatori possono concentrarsi unicamente sul codice dell'applicazione, migliorando la produttività e riducendo i tempi di rilascio

6.2.2 Architettura logica del sistema

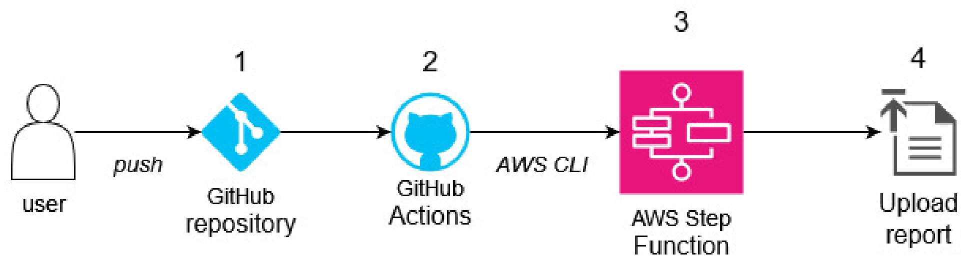


Figura 6.1: Flusso generale del sistema

Il flusso generale del sistema, come mostrato nella Figura 6.1, rappresenta l'interazione tra un'applicazione basata su un'architettura *serverless* su *AWS*$_G$ e *GitHub*$_G$, per l'analisi automatizzata del codice sorgente con la generazione di report di vulnerabilità. Ecco una descrizione dettagliata dei vari passaggi:

1. **Esecuzione del *push* su GitHub**: L'utente, dopo aver effettuato modifiche al codice sorgente nella propria *repository*$_G$ locale, esegue un *push* verso la *repository*$_G$ remota ospitata su *GitHub*$_G$. Questo *push* rappresenta l'inizio del flusso, poiché *GitHub*$_G$ rileva che una modifica è stata effettuata nel codice sorgente e innesca automaticamente una serie di azioni predefinite attraverso *GitHub*$_G$ *Actions*.

2. **GitHub Actions**: Quando si verifica un determinato evento, in questo caso un *psuh*, *GitHub*$_G$ *Actions* esegue uno flusso di istruzioni e, attraverso *AWS*$_G$ *Command Line Interface (CLI)*$_G$, interagisce con la *AWS*$_G$ *Step Fucntion*

3. **Esecuzione della Step Function**: *AWS*$_G$ *Step Function* è un servizio che permette di definire flussi di lavoro basati su stati, in cui ogni stato rappresenta un'attività da eseguire. In questo caso, la *Step Function* è composta da una serie di passaggi che coinvolgono diverse funzioni Lambda e il servizio *S3*$_G$. Il flusso di esecuzione della Step Function prevede i seguenti passi:

   (a) Creazione di *chunk* di codice: una funzione Lambda divide il codice sorgente in *chunk*

   (b) Analisi del codice: il codice sorgente viene analizzato, utilizzando le *CWE*$_G$ per rilevare eventuali vulnerabilità presenti nel codice e *CVSS*$_G$ 3.1 per calcolare lo *score* della gravità della debolezza rilevata

   (c) Creazione del report: una volta analizzato tutto il codice sorgente viene generato un *file* che contiene tutte le vulnerabilità trovate dal sistema con le relative informazioni

4. L'ultimo passaggio consiste nel caricare il report generato nella *repository*$_G$, che contiene il codice sorgente, col nome "report.md"

6.2.2.1 GitHub Actions

Per automatizzare l'analisi del codice sorgente e l'integrazione con il sistema basato su *AWS*$_G$, è stato configurato un flusso all'interno di *GitHub*$_G$ *Actions*.

Il flusso viene eseguito ogni volta che si verifica un *push* nella repository remota. In dettaglio, il processo funziona come segue:

1. Quando l'utente esegue un *push* nella repository locale, il codice viene inviato alla repository remota su *GitHub*$_G$. Questo evento avvia il flusso di *GitHub*$_G$ *Actions*, che è stato configurato per eseguire una serie di istruzioni predefinite.

2. Per garantire la sicurezza nell'accesso ai servizi *AWS*$_G$, sono utilizzati i *GitHub*$_G$ *Secrets*, una funzionalità che permette di archiviare in modo sicuro le credenziali sensibili. In questo caso, l'ID di accesso (*Access Key ID*) e la chiave segreta (*Secret Access Key*) dell'account *AWS*$_G$ devono essere memorizzati nei *secrets* della repository *GitHub*$_G$. Questo permette al flusso di Actions di autenticarsi con i servizi *AWS*$_G$ senza esporre pubblicamente le credenziali.

3. All'interno del flusso, viene utilizzata l'interfaccia a riga di comando di *AWS*$_G$ (*CLI*$_G$) per caricare l'intera repository in un *bucket* *S3*$_G$. In questo modo, il codice sorgente è disponibile per l'analisi da parte della *Step Function* di *AWS*$_G$.

4. Infine, viene anche avviata la *Step Function*, che coordina l'esecuzione delle diverse fasi del processo di analisi del codice. Quando la *Step Function* viene chiamata dal flusso di *GitHub*$_G$ *Actions*, è necessario passare un *GitHub Token* per garantire l'accesso alla *repository*$_G$ remota. Questo token è utilizzato per autenticare le richieste e consentire operazioni come la lettura e la scrittura dei file nella *repository*$_G$.

6.2.2.2 Step Function

Tramite *AWS*$_G$ *Step Function* è stata creata una *state machine*, un flusso di esecuzione che coinvolge diversi servizi *AWS*$_G$ con l'obiettivo di analizzare il codice sorgente e generare un report di vulnerabilità. Il flusso di esecuzione, come mostrato nella figura 6.2, può essere suddiviso nelle seguenti fasi:

1. **Creazione dei chunk di codice:** La prima fase consiste nel suddividere il codice sorgente in blocchi, chiamati *chunk*, ciascuno dei quali rispetta il limite massimo di token che può essere processato dal modello *GPT* **G**. Nel caso specifico, ogni *chunk* è creato per essere lungo circa 4000 token (il limite massimo del modello *GPT* **G**). Una volta generati, i *chunk* vengono salvati in un *bucket* *S3* **G**

2. **Valutazione della rilevanza dei chunk:** Ogni *chunk* viene poi analizzato dal modello *GPT* **G** per verificare se il codice in esso contenuto è rilevante dal punto di vista della sicurezza informatica. Se il modello determina che un *chunk* è rilevante, si procede alla fase successiva di analisi.

3. **Analisi tramite CWE:** Per ogni *chunk* ritenuto rilevante, viene chiesto al modello di analizzare il codice alla ricerca di specifiche *CWE* **G**. Poiché il modello *GPT* **G** ha dei limiti sul numero di token processabili in *input* e delle restrizioni sul numero di chiamate al minuto, è necessario suddividere ulteriormente la richiesta di analisi.
   Nel caso del modello *GPT* **G** *4o* **G**, è possibile inviare una singola richiesta per volta, rispettando il limite di 30.000 *Tokens per Minute (TPM)* **G**. Utilizzando invece il modello *GPT* **G** *4o* **G**-mini, che ha un limite di 200.000 *TPM* **G**, è possibile parallelizzare fino a 18 richieste contemporaneamente [14]. Dato che il sistema deve analizzare un set di 174 *CWE* **G** e ogni richiesta al modello ne contiene circa 10, saranno necessarie 18 chiamate al modello.
   In un progetto di piccole o medie dimensioni (ad esempio con 5-6 *chunk*), l'intero processo di analisi delle *CWE* **G** richiede circa 10 minuti se si utilizza il modello *GPT* **G** *4o* **G**, mentre con il modello *GPT* **G** *4o* **G**-mini il tempo può essere ridotto a circa un minuto grazie alla capacità di parallelizzazione.

4. **Generazione del vettore CVSS:** Dopo che il modello ha identificato eventuali debolezze nel codice tramite le *CWE* **G**, viene richiesto di ge-

nerare il vettore *CVSS*$_G$, che permette di calcolare il *basic score* della vulnerabilità. I risultati vengono salvati in un *bucket* *S3*$_G$.

5. **Aggregazione dei risultati e generazione del report:** Al termine dell'analisi di tutti i *chunk*, il sistema preleva i risultati salvati nel *bucket* *S3*$_G$ e li aggrega in un unico file, generando così il report finale. Questo report viene quindi caricato nella repository *GitHub*$_G$, dove può essere consultato dagli sviluppatori.

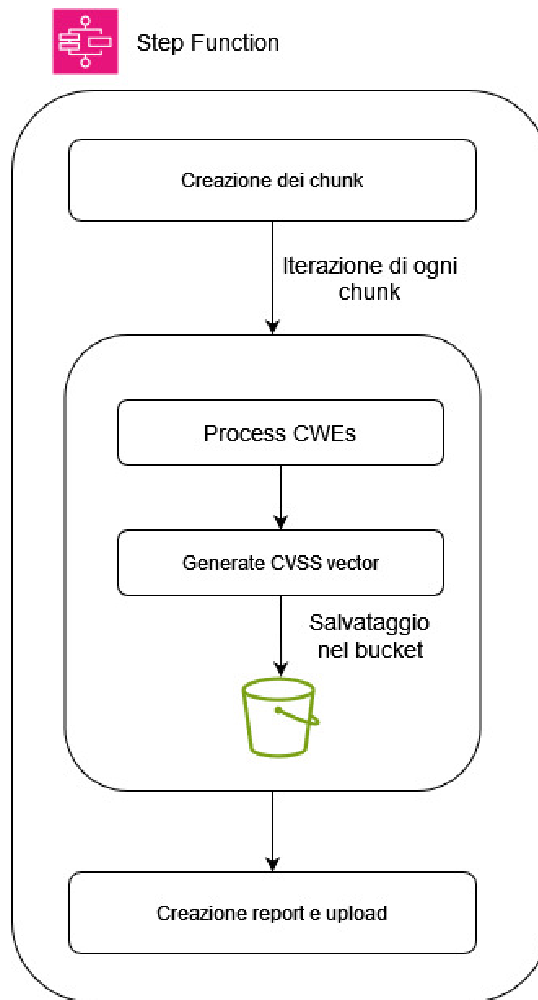


Figura 6.2: Flusso della *Step Function*

Capitolo 7

Implementazione del sistema

Nel capitolo seguente vengono esposte le parti più importanti dell'implementazione del sistema.

7.1 Configurazione serverless

La configurazione del *framework*$_G$ *serverless*$_G$ è essenziale per orchestrare sia la definizione delle funzioni Lambda che l'implementazione del flusso delle *Step Function*. Il file di configurazione standard per il *framework*$_G$ è in formato YAML (come serverless.yml), ma in questo caso è stato utilizzato TypeScript grazie a specifici *plugin*. Questo approccio consente di sfruttare la tipizzazione statica, facilitando la manutenzione e riducendo il rischio di errori.
Il *file* di configurazione principale, chiamato serverless.ts, include informazioni fondamentali come il provider del *cloud*, le risorse da utilizzare, le funzioni Lambda e le Step Functions che coordinano l'intero flusso di esecuzione. Nel codice 7.1 viene riportata la configurazione del *file* serverless.ts.

```
1 const serverlessConfig: CustomServerless = {
2   service: 'genai-report-vulnerability-openai',
3   frameworkVersion: '3',
4   useDotenv: true,
5   provider: {
6     name: 'aws',
7     runtime: 'nodejs18.x',
8     region: 'us-east-1',
9     stage: '${opt:stage, \'dev\'}',
```

```
10      stackName: '${self:provider.stage}-${self:service}
          ',
11      deploymentBucket: {
12        name: '${self:custom.${self:provider.stage}.
             s3DeploymentBucket}',
13      },
14      iam: {
15        role: {
16          name: '${self:provider.stage}-${self:service}-
              role',
17          statements: [
18            {
19              Effect: 'Allow',
20              Action: [
21                'states:*',
22                's3:*',
23                'ssm:*',
24                'sts:*',
25                'lambda:*',
26              ],
27              Resource: '*',
28            },
29          ],
30        },
31      },
32      functions: { ... },
33      resources: { ... },
34      stepFunctions: { ... }
35  };
```

Listing 7.1: Configurazione del file "serveless.ts"

Nel codice 7.1, vengono configurati diversi elementi fondamentali:

- Provider del cloud: In questo esempio, viene utilizzato *AWS*$_{G}$ come *provider* e si specifica la regione in cui verranno eseguite le risorse
- Funzioni Lambda: nella sezione functions vengono definite tutte le Lambda utilizzate dal sistema
- Step Functions: nella sezione stepFunctions viene definito il flusso della *state machine*

In questo modo, il file serverless.ts fornisce una visione completa e strutturata dell'infrastruttura *cloud*, includendo la gestione delle funzioni Lambda e la definizione del flusso di esecuzione con *Step Functions*.

7.2 Configurazione GitHub Actions

La configurazione di *GitHub*$_G$ *Actions* è gestita tramite un *file* in formato *YAML*, che deve essere inserito nella *repository*$_G$ del progetto. Questo *file* definisce i *trigger* (eventi che attivano le azioni) e i flussi di lavoro che eseguono operazioni come il caricamento dei *file* su *S3*$_G$ e l'avvio della *step function* su *AWS*$_G$.

```
1  name: Upload Files to S3 and start Step Function
2
3  on:
4    push:
5      branches:
6        - main
```

Listing 7.2: Trigger dell'esecuzione del flusso

Il codice 7.2 definisce che il flusso di lavoro di *GitHub*$_G$ *Actions* deve essere eseguito ogni volta che viene effettuato un *push* sul *branch main*.
Successivamente, quando si verifica un evento *push*, il sistema verifica se nel messaggio del *commit* è presente la stringa *[skip]*. Se questa parola chiave è inclusa nel messaggio, l'esecuzione delle *GitHub*$_G$ *Actions* viene interrotta, permettendo così di apportare modifiche alla *repository*$_G$ senza attivare l'analisi del codice.

```
1  jobs:
2      check-commit:
3      runs-on: ubuntu-latest
4
5      outputs:
6        skip: ${{ steps.check_message.outputs.skip }}
7
8      steps:
9        - name: Check commit message
10         id: check_message
11         run: |
12           if [[ "${{ github.event.head_commit.message
               }}" == *"[skip]"* ]]; then
13             echo "Skipping Step Function"
14             echo "skip=true" >> $GITHUB_OUTPUT
15           else
16             echo "skip=false" >> $GITHUB_OUTPUT
```

```
17            fi
```

Listing 7.3: Controllo del messaggio del *commit*

L'ultima fase consiste nel caricamento del codice sorgente su *S3* **G** e l'avvio della *step function* su *AWS* **G**. Se l'*output* del controllo del *commit* indica che il processo non deve essere saltato, viene inizializzata la *AWS* **G** *CLI* **G** e il codice viene caricato nel *bucket* **G** *S3* **G**.
Infine la *step function* viene avviata tramite l'*Amazon Resource Name (ARN)* **G** specificato, e vengono passati parametri come il nome del progetto e il percorso del *file* di report.

```
1  runs-on: ubuntu-latest
2  needs: check-commit
3  if: needs.check-commit.outputs.skip == 'false'
4
5  steps:
6    - name: Checkout code
7      uses: actions/checkout@v3
8
9    - name: Configure AWS credentials
10     uses: aws-actions/configure-aws-credentials@v3
11     with:
12       aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID
           }}
13       aws-secret-access-key: ${{ secrets.
           AWS_SECRET_ACCESS_KEY }}
14       aws-region: us-east-1
15
16   - name: Upload to S3
17     run: |
18       aws s3 sync ./src s3://${{ secrets.BUCKET }}/
           genai-report-openai --delete
19
20   - name: Trigger Step Function
21     run: |
22       aws stepfunctions start-execution --state-
           machine-arn ${{ secrets.STATE_MACHINE_ARN }}
           --input '{"prefixProject":"genai-report-openai
           /", "githubRepo":"${{ github.repository }}", "
           githubPath":"report.md", "githubBranch":"${{
           github.ref_name }}"}'
```

Listing 7.4: *Upload* del codice e avvio della *Step Function*

7.3 Configurazione Step Function

Nel grafico 7.1 viene riportato il flusso degli stati della *Step Function* creata.

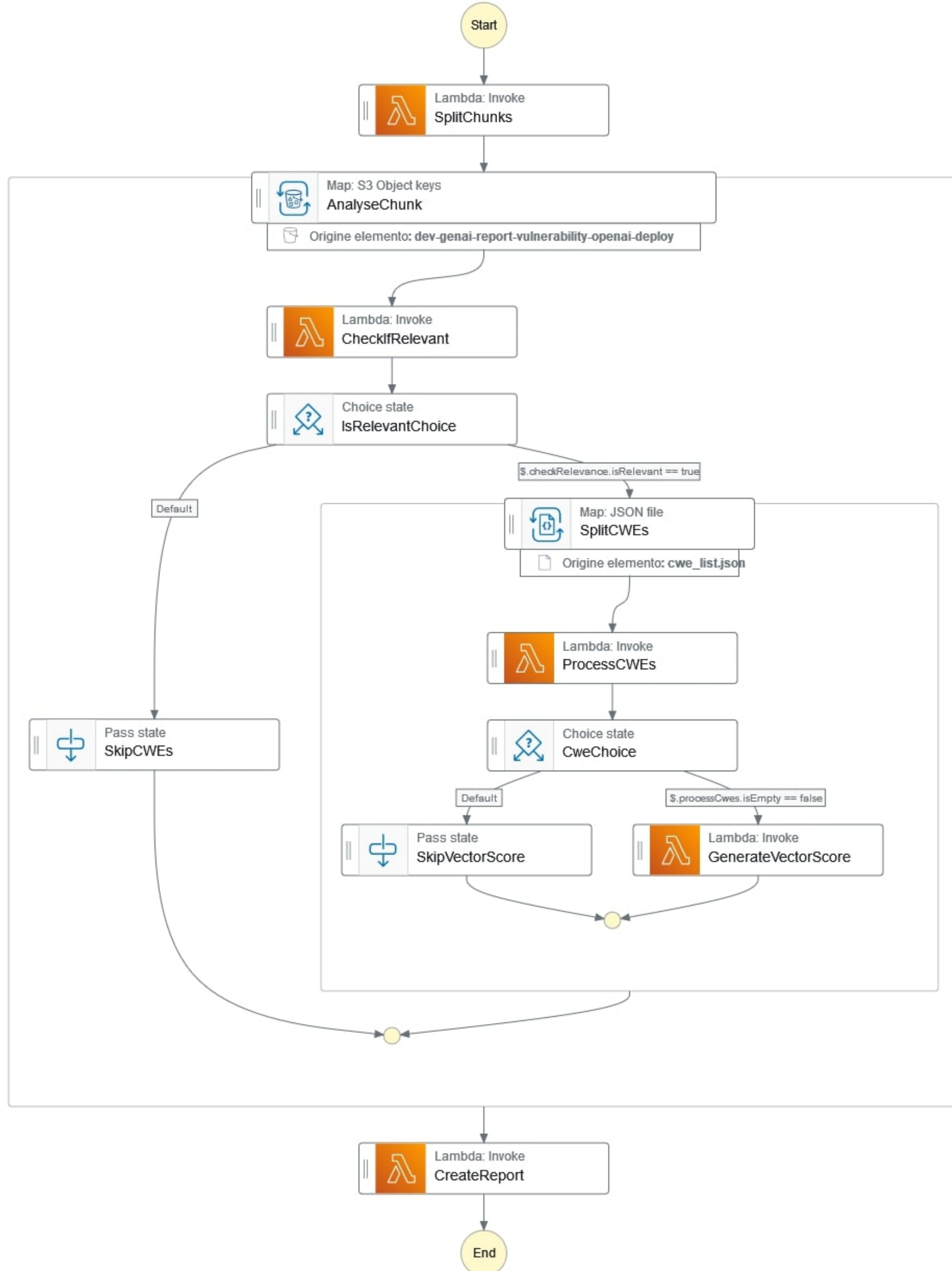


Figura 7.1: Grafico dell'implementazione della *Step Function*

Come si può osservare dall'immagine, si può dividere in tre fasi: la fase iniziale rappresentata dalla Lambda denominata "SplitChunks", la fase intermedia che racchiude il processo d'analisi del codice, il quale viene eseguito per

ogni elemento nel *bucket*$_G$ specificato e, infine, la fase finale rappresentata dalla Lambda "CreateReport".

7.3.1 Fase iniziale

7.3.1.1 Lambda: SplitChunks

La prima fase consiste nel prelevare il codice sorgente dal *bucket*$_G$ *S3*$_G$ per creare dei blocchi (*chunks*) di circa 4000 token. Questo limite è determinato dalla capacità massima di *input* del modello *GPT*$_G$. Ogni *chunk* viene successivamente salvato in una cartella dedicata all'interno del *bucket*$_G$.
Poiché un progetto può includere *file* di configurazione o altri file che non contengono codice sorgente, la funzione controlla l'estensione dei *file* prima di processarli. In questo modo, vengono ignorati i file che non sono associabili a linguaggi di programmazione, riducendo il carico di lavoro e assicurando che vengano analizzati solo i *file* di interesse per l'individuazione delle vulnerabilità.

7.3.2 Analisi del codice

La fase successiva prevede l'analisi del codice vero e proprio. In questa fase, il sistema comunica ripetutamente con il modello *GPT*$_G$ per eseguire l'analisi di ogni *chunk* di codice.

7.3.2.1 Comunicazione con il modello

Poiché la comunicazione con il modello *GPT*$_G$ avviene più volte, è stata implementata una funzione specifica per semplificare questo processo. Il codice 7.5 mostra come viene effettuata una richiesta al modello *AI*$_G$.

```
1 export async function invokeGPT(
2   systemPrompt: string,
3   userPrompt: string,
4   model: string,
5 ): Promise<ModelOutput> {
6   const completion: OpenAI.Chat.ChatCompletion = await
        client.chat.completions.create({
7     messages: [
8       { role: 'system', content: systemPrompt },
```

```
      { role: 'user', content: userPrompt },
    ],
    model: model,
    temperature: 0.2,
    n: 1,
  });

  const response: string | null = completion.choices
     [0]?.message.content;
  if (response === null || response === '') {
    throw new Error('Failed to generate response');
  }

  return {
    content: response.trim(),
    totalTokens: completion.usage?.total_tokens ?? 0,
    promptTokens: completion.usage?.prompt_tokens ??
       0,
    responseTokens: completion.usage?.
       completion_tokens ?? 0,
  };
}
```

Listing 7.5: Codice che permette di fare una richiesta al modello *GPT*

Questa funzione permette di fare una richiesta al modello *GPT* $_G$ con un *system prompt* e un *user prompt* personalizzati. La funzione ritorna la risposta generata dal modello, insieme al conteggio dei token utilizzati: i *totalTokens*, i *promptTokens* e i *responseTokens*, così da poter monitorare il consumo dei token durante l'intero processo di comunicazione.

Si nota inoltre che la temperatura del modello è impostata a 0.2. La temperatura controlla la variabilità delle risposte generate dal modello: valori più bassi rendono le risposte più deterministiche, mentre valori più alti rendono le risposte più creative e meno prevedibili. In questo caso, una temperatura di 0.2 garantisce che il modello produca risposte altamente coerenti e vicine alle attese, riducendo il rischio di errori o *output* non formattati correttamente. Dopo diversi tentativi, è stato stabilito che 0.2 fosse il valore ottimale per ottenere risposte accurate con i modelli *GPT* $_G$.

7.3.2.2 AnalyzeChunk

L'attività *AnalyzeChunk* riguarda l'iterazione degli elementi nel *bucket*$_G$ specificato, infatti per ogni blocco di codice precedentemente creato viene eseguita ogni attività contenuta in questo blocco (con parallelismo impostato ad uno per rispettare i limiti di *GPT*$_G$), ovvero:

1. La Lambda "CheckIfRelevant"
2. Lo *choice state* "IsRelevantChoice"
3. L'iterazione "SplitCWEs"

7.3.2.3 Lambda CheckIfRelevant

La Lambda "CheckIfRevelant" controlla se il blocco di codice è rilevante per la sicurezza informatica, per farlo chiede al modello *4o*$_G$-mini attraverso il *system prompt* 7.6.

```
Analyze the following file content and determine if
it contains code relevant for cybersecurity
vulnerability assessment or if it is just
configuration code.
If the file is relevant for cybersecurity assessment
respond with "relevant".
If the file is not relevant respond with "not
relevant".
Do not provide any additional explanations or content.
```

Listing 7.6: *System promot* per la rilevanza del blocco di codice

Successivamente, viene estrapolata la risposta del modello e si passa alla fase successiva il valore *true* se il modello ha restituito *relevance*, altrimenti *false*.

7.3.2.4 Choice state: IsRelevantChoice

Lo *choice state* permette di definire come procedere con il flusso tramite l'esito di un controllo, in questo caso se il parametro preso in *input* dall'attività precedente risulta uguale a *true* si procede con l'analisi del blocco di codice, altrimenti il blocco in non viene analizzato. Il codice 7.7 rappresenta l'implementazione dello *choice state*.

```
IsRelevantChoice: {
  Type: 'Choice',
  Choices: [
    {
      Variable: '$.checkRelevance.isRelevant',
      BooleanEquals: true,
      Next: 'SplitCWEs',
    },
  ],
  Default: 'SkipCWEs',
},
```

Listing 7.7: Definizione da inserire nel file serverless.ts per definire lo stato IsRelevantChoice

7.3.2.5 SplitCWEs

"SplitCWEs" consiste nell'iterare le attività al suo interno per ogni *set* di CWE_G definite, quindi per ogni *set* vengono eseguenti le seguenti attività:

1. Lambda ProccessCWEs
2. Choice state CweChoice
3. Lambda GenerateVectorScore

7.3.2.6 Lambda ProccessCWEs

La Lambda "ProccessCWEs" chiede al modello GPT_G di individuare nel blocco di codice delle specifiche CWE_G, per analizzare la risposta la funzione utilizza un'espressione regolare, ovvero una sequenza di caratteri utilizzata per estrarre stringhe di testo. L'espressione regolare utilizzata è riportata nel codice 7.8.

```
/CWE-?\[?(\d+)\]?:?\s*(present|not
present)\s*(File:([\s\S]*?)\n
Area:([\s\S]*?)\n
Explanation:([\s\S]*?)
Resolution:(.*))?/ig,
```

Listing 7.8: Espressione regolare per estrapolare le informazioni delle debolezze rilevate

Nel dettaglio l'espressione regolare è composta dalle seguenti parti:

- **CWE-?**: Cerca la stringa "CWE" seguita facoltativamente da un trattino (-? significa "zero o un trattino).
- **[?(\d+)]?**: Cerca facoltativamente una parentesi quadra di apertura `[?`. `(\d+)` è un gruppo di cattura che cattura una o più cifre (\d+ rappresenta un numero intero). Questo rappresenta il numero *CWE*$_{G}$. Segue facoltativamente una parentesi quadra di chiusura `]?`.
- **:?**: Cerca facoltativamente un due punti (`:`).
- **\s***: Cerca uno o più spazi bianchi (tabulazioni, spazi o nuove righe), che potrebbero esserci tra il numero CWE e lo stato.
- **(present|not present)**: Un altro gruppo di cattura. Cerca la stringa esattamente uguale a "present" o "not present", che indica se la vulnerabilità è presente o no.
- **\s***: Ancora una volta, uno o più spazi bianchi per permettere una separazione tra i dati.
- **File:(\[\s\S]*?)**: Cerca la stringa "File:", seguita da qualsiasi carattere (spazio bianco o non) `\s\S` con cattura non *greedy* `*?`.
- **\nArea:(\[\s\S]*?)**: Cerca una nuova riga (\n), seguita da "Area:", e cattura l'area associata con la vulnerabilità.
- **\nExplanation:(\[\s\S]*?)**: Cerca un'altra nuova riga seguita dalla stringa "Explanation:", e cattura la spiegazione.
- **Resolution:(.*)**: Cerca la stringa "Resolution:" e cattura tutto ciò che segue dopo di essa, rappresentando la risoluzione del problema.
- **i**: Indica che la regex è case-insensitive (non distingue maiuscole e minuscole).
- **g**: Indica la modalità *global*, che cerca tutte le corrispondenze nella stringa di input, non solo la prima.

7.3.2.7 Choice state: CweChoice

Lo *choice state* "CweChoice" controlla se la Lambda "ProccessCWEs" (7.3.2.6) ha rilevato che nel blocco di codice vi sono delle debolezze allora il flusso dell'esecuzione verrà indirizzato verso la Lambda "GenerateVectorScore", altrimenti il blocco di codice viene ignorato. Il codice 7.9 riporta l'implementazione di "CweChoice".

```
CweChoice: {
    Type: 'Choice',
    Choices: [
      {
        Variable: '$.processCwes.isEmpty',
        BooleanEquals: false,
        Next: 'GenerateVectorScore',
      },
    ],
    Default: 'SkipVectorScore',
},
```

Listing 7.9: Definizione di CweChoice nel file serverless.ts

7.3.2.8 Lambda GenerateVectorScore

La Lambda "GenerateVectorScope" ha l'obittivo di chiedere al modello *AI* **G** di generare il vettore *CVSS* **G** 3.1 per poi calcolare lo *basic score*.
Per estrarre il risultato dalla risposta del modello si usa l'espressione regolare 7.10.

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

Listing 7.10: Espressione regolare per estrarre il vettore *Common Vulnerability Scoring System*

- `CWE\s*-?\s*(\d+)`: Cerca la stringa "CWE", opzionalmente seguita da un trattino e spazi bianchi, poi cattura un numero (il codice *CWE*).
- `.*\s+Vector:\s+`: Cerca qualsiasi testo (`.*`), seguito da uno o più spazi bianchi e dalla stringa "Vector:".

- `[?((CVSS:3i)??AV:.AC:.PR:.UI:.S:.C:.I:.A:.)]?`: Opzionalmente cattura il punteggio *CVSS*$_G$ 3.1 seguito dai suoi sottopunteggi (`AV`, `AC`, `PR`, `UI`, `S`, `C`, `I`, `A`).
- `AV:([\s\S]*?)` a `A:\s*(.+)`: Cattura i singoli sottopunteggi (`AV`, `AC`, `PR`, `UI`, `S`, `C`, `I`, `A`) con ciascun campo seguito da qualsiasi carattere, separato da spazi bianchi.

Questa *regex* permette di estrarre i dati essenziali relativi ai vettori di vulnerabilità e i loro punteggi in un formato standard.
Successivamente i dati relativi alla *CWE*$_G$ rilevata e al suo vettore *CVSS*$_G$ vengono caricati in un *bucket*$_G$ apposito.

7.3.3 Generazione del report

L'ultima Lambda che conclude il flusso della *step function* è "CreateReport". Questa Lambda preleva tutte le informazioni delle debolezze rilevata dal *bucket*$_G$ *S3*$_G$, per poi creare un *file* con estensione ".md" che riporta tali informazioni. La parte del codice che genera il testo che viene inserito nel *file* viene riportato nel *listing* 7.11.

```
1  for (const vectorGroup of vectorGroups) {
2      report += ` File ${++counter}: ${vectorGroup.
          fileName}\n\n`;
3      for (const [index, vector] of vectorGroup.vectors.
          entries()) {
4        report += ` Evidence ${index + 1}\n`;
5        report += `| CWE | ${vector.cweId}, ${vector.
           cweName}\n`;
6        report += '| --- | --- |\n';
7        report += `| Gravity | ${vector.severity}, ${
           vector.vector} |\n`;
8        report += '| **Metric** | **Explanation** |\n';
9        report += `| Attack Vector (AV) | ${vector.av}
           |\n`;
10       report += `| Attack Complexity (AC) | ${vector.
           ac} |\n`;
11       report += `| Privileges Required (PR) | ${vector
           .pr} |\n`;
12       report += `| User Interaction (UI) | ${vector.ui
           } |\n`;
```

```
13          report += `| Scope (S) | ${vector.s} |\n`;
14          report += `| Confidentiality Impact (C) | ${
               vector.c} |\n`;
15          report += `| Integrity Impact (I) | ${vector.i}
               |\n`;
16          report += `| Availability Impact (A) | ${vector.
               a} |\n`;
17          report += `| **Score** | ${vector.baseScore} |\n
               \n`;
18          report += `**Explanation**: ${vector.cweDetails.
               explanation}\n\n`;
19          report += `**Affected Area**:\n${vector.
               cweDetails.area}\n\n`;
20          report += `**Resolution**: ${vector.cweDetails.
               resolution}\n\n`;
21      }
22  }
```

Listing 7.11: Parte del codice della Lambda "CreateReport"

Come si può notare dal codice 7.11, nel report viene salvato, per ogni *CWE*$_G$ individuata il vettore *CVSS*$_G$ con il relativo *score*, la spiegazione, l'area d'effetto e una possibile soluzione.

Infine, una volta completato il report, quest'ultimo viene caricato sul *repository*$_G$ del progetto analizzato.

Capitolo 8

Uso del sistema

Questo capitolo spiega come integrare il sistema di analisi in una *repository*$_G$ e come iniziare il processo di analisi del codice sorgente.

8.1 Impostazioni iniziali

Il sistema deve poter accedere alla *repository*$_G$ di *GitHub*$_G$ quindi, quando si crea una nuova *repository*$_G$, è necessario generare un *GitHub*$_G$ token utilizzato con i permessi di lettura e scrittura sul progetto che si intende analizzare.
Inoltre, si deve configurare nelle impostazioni della *repository*$_G$ i seguenti *secrets* (il nome deve corrispondere esattamente al nome della chiave da utilizzare):

- AWS_ACCESS_KEY_ID: ID di un account per accedere ai servizi *AWS*$_G$
- AWS_SECRET_ACCESS_KEY: chiave associata all'ID precedentemente inserito
- BUCKET: *ARN*$_G$ del *bucket*$_G$ che si desidera utilizzare per caricare il progetto su *AWS*$_G$ *S3*$_G$
- STATE_MACHINE_ARN: *ARN*$_G$ della *state machine* che gestisce il flusso di esecuzione del sistema

8.2 GitHub Action

Una volta creata una *repository*$_G$, è necessario creare una *GitHub*$_G$ *Action* nel seguente modo:

1. Creare una cartella chiamata *workflows* all'interno di una cartella *.github* nella *root directory* del progetto.
2. Creare un file con estensione ".yml" nella cartella appena creata.

All'interno di questo *file* inserire il codice presente nella sezione 7.2.

8.3 Avvio dell'analisi

Per eseguire un'analisi del codice è necessario effettuare un'azione di *push*, ovvero:

1. Dopo aver apportato modifiche al codice sorgente, eseguire il seguente comando: ‘git add .‘
2. Successivamente, eseguire il comando: ‘git commit -m "*inserire un breve messaggio*"‘
3. Infine, utilizzare ‘git push origin main‘ per salvare le modifiche nel branch "main" e avviare l'analisi del codice.

Se si desidera evitare l'analisi del codice durante un'azione di *push*, è sufficiente aggiungere "[skip]" al messaggio del *commit*. Ad esempio:

```
git commit -m "small fixes [skip]"
```

Listing 8.1: Impedire l'avvio dell'analisi del codice

8.4 Consultazione del report

Quando si esegue l'analisi del codice sorgente, una volta che la *state machine* ha terminato (circa un minuto per il modello *GPT*$_G$-*4o*$_G$-mini, circa 10 minuti per il modello *GPT*$_G$-*4o*$_G$ per progetti medio-piccoli), sarà possibile notare che

la *state machine* ha caricato il *file* del report nella *repository*$_G$.
Aprendo il *file*, sarà possibile visualizzare, come illustra la figura 8.1, le seguenti informazioni per ogni debolezza rilevata:

- ID della *CWE*$_G$ rilevata con una breve descrizione
- Gravità della vulnerabilità, che può variare tra *low, medium* e *high*
- Informazioni riguardanti il *CVSS*$_G$ 3.1
- Spiegazione: una breve spiegazione della vulnerabilità trovata
- Area affetta: l'area del codice che riguarda la vulnerabilità
- Risoluzione: una breve descrizione su come risolvere la vulnerabilità
- Uno *score* numerico che può variare tra 0 e 10; più alto è il punteggio, maggiore è l'importanza della vulnerabilità

File 1: genai-report-openai/functions/handlers/chunks/parse_file_extension.ts

Evidence 1

CWE	**646, Reliance on File Name or Extension of Externally-Supplied File**
Gravity	medium, AV:N/AC:L/PR:N/UI:N/S:U/C:N/I:N/A:L
Metric	**Explanation**
Attack Vector (AV)	Network - The vulnerability can be exploited over a network.
Attack Complexity (AC)	Low - The attacker can exploit the vulnerability with minimal effort.
Privileges Required (PR)	None - No privileges are required to exploit the vulnerability.
User Interaction (UI)	None - User interaction is not required for exploitation.
Scope (S)	Unchanged - The exploitation of the vulnerability does not affect the scope of the system.
Confidentiality Impact (C)	None - There is no impact on confidentiality.
Integrity Impact (I)	None - There is no impact on integrity.
Availability Impact (A)	Low - There is a potential impact on availability, as the application may process unwanted files.
Score	5.3

Explanation: The code relies on the file extension to determine if a file is relevant, which can be manipulated by an attacker by changing the file name or extension. This can lead to the application processing files that should be ignored based on their content.

Affected Area: The function `isFileExtensionRelevant` checks if a file extension is relevant based on a predefined list.

Resolution: Implement a more robust validation mechanism that checks the actual content of the file rather than relying solely on the file extension. This could involve analyzing the file's content type or structure.

Figura 8.1: Esempio di come viene visualizzato il *file* report

Capitolo 9

Conclusioni

Nel corso del progetto, sono stati valutati diversi modelli di *Generative AI* **G** di *OpenAI* **G**, considerando costi e prestazioni. Per bilanciare accuratezza e costi è stato deciso di utilizzare il modello *4o* **G**-mini per la valutazione preliminare del codice e la generazione del vettore *CVSS* **G** e riservare il modello *4o* **G** per l'analisi approfondita delle vulnerabilità.
Inoltre, tramite i servizi di *AWS* **G**, in particolare Lambda, *S3* **G** e *Step Functions* e con l'utilizzo di *GitHub* **G** *Actions* e *Secrets* si è realizzato un sistema che analizza il codice sorgente presente in una *repository* **G** di *GitHub* **G**, generando un report che riporta eventuali vulnerabilità individuate.

9.1 Valutazione dei modelli

I modelli *GPT* **G** analizzati nel corso del progetto sono stati: 3.5-turbo, 4, 4-turbo, *4o* **G**-mini e *4o* **G**. Analizzando il costo e le prestazioni, si è osservato quanto segue:

- **3.5-turbo**: considerato obsoleto, superato dal modello *4o* **G**-mini, che offre prestazioni superiori a costi inferiori [12]
- **4 e 4-turbo**: entrambi superati dal modello *4o* **G**, ora considerato il più avanzato tra quelli di *OpenAI* **G** [11]

- **4o-mini**: il modello più economico offerto da *OpenAI*$_G$, con un costo di 0.15 dollari per un milione di token in *input* e 0.6 dollari per un milione di token in *output*

- **4o**: il modello più avanzato, con un costo di 5 dollari per un milione di token in *input* e 15 dollari per un milione di token in *output*, rispetto ai 10 e 30 dollari del modello 4-turbo

9.1.1 Costi relativi all'esecuzione

Considerando che i modelli *4o*$_G$ e *4o*$_G$-mini sono attualmente i più avanzati di *OpenAI*$_G$, è stato calcolato il costo per l'analisi di codice sorgente, valutando la proporzione tra costo ed efficacia del sistema. Per un progetto di piccole-medie dimensioni, con 5 blocchi di codice e ipotizzando che su ogni blocco venga rilevata ogni *set* di *CWE*$_G$, l'utilizzo dei token si distribuisce come segue:

1. Fase 1: identificazione della rilevanza del blocco di codice, con circa 2500 token in *input* e 2 in *output*

2. Fase 2: individuazione delle *CWE*$_G$, richiedendo 18 analisi con circa 3000 token in *input* e 200 in *output* per ogni richiesta

3. Fase 3: generazione del vettore *CVSS*$_G$, con richieste equivalenti alla fase precedente e un utilizzo di circa 300 token in *input* e 200 in *output* per richiesta

In totale, l'utilizzo di token per i 5 blocchi di codice sarà di 61900 in *input* e 6302 in *output*. Per il modello *4o*$_G$, questo si traduce in un costo di circa 0,31 dollari in *input* e 0,01 dollari in *output*, per un totale di 0,32 dollari per blocco. Per il modello *4o*$_G$-mini, il costo scende a circa 0,01 dollari in *input* e 0,004 dollari in *output*, per un totale di 0,014 dollari.
Sebbene il modello *4o*$_G$-mini sia significativamente più economico, la sua affidabilità è inferiore rispetto al modello *4o*$_G$. Infatti, pur identificando correttamente le vulnerabilità associate alle *CWE*$_G$, le spiegazioni, le aree di impatto, le soluzioni e il vettore *CVSS*$_G$ spesso risultano incoerenti. L'analisi ripetuta

dello stesso codice produce report con differenze significative, soprattutto nei dettagli delle vulnerabilità. Di conseguenza, il modello *4o*$_G$-mini non è considerato sufficientemente affidabile per un'analisi approfondita, mentre il modello *4o*$_G$, pur più accurato, comporta costi elevati.

Per ottimizzare i costi e mantenere un buon livello di accuratezza, si è optato per l'uso del modello *4o*$_G$-mini per la valutazione preliminare della rilevanza del codice e la generazione del vettore *CVSS*$_G$, riservando il modello *4o*$_G$ per l'analisi delle *CWE*$_G$, con un costo complessivo di circa 0,1 dollari per ogni blocco di codice.

L'utilizzo del modello *4o*$_G$ però non risolve pienamente il problema di inconsistenza nei report; infatti si verificano delle variazioni nei risultati tra analisi ripetute eseguite su una *repository*$_G$ di *GitHub*$_G$ che contiene codice con vulnerabilità note. Ciò suggerisce che i modelli *GPT*$_G$ attuali non sono ancora in grado di garantire un'analisi del codice perfettamente affidabile, ma aggiornamenti futuri potrebbero migliorare le loro prestazioni.

Un'ulteriore opzione per migliorare l'analisi sarebbe l'uso del *fine-tuning*, il quale consiste nel processo di riaddestramento del modello su un corpus di dati specifici per adattarlo meglio all'analisi delle vulnerabilità nel codice. Tuttavia, il *fine-tuning* non è stato implementato in questo progetto a causa dei costi elevati e della complessità nell'individuazione dei migliori esempi per l'addestramento.

9.2 Raggiungimento degli obiettivi

Gli obiettivi delineati nella sezione 3.3 sono stati raggiunti attraverso l'implementazione di un sistema automatizzato che integra *GitHub*$_G$ e *AWS*$_G$ per l'analisi delle vulnerabilità nel codice sorgente. Il sistema carica il codice su *S3*$_G$ e avvia automaticamente l'analisi, producendo un report dettagliato che elenca tutte le vulnerabilità identificate, accompagnate da spiegazioni e suggerimenti per la risoluzione. Grazie all'uso dei modelli *GPT*$_G$, è stato possibile classificare le vulnerabilità assegnando a ciascuna uno *score* che ne riflette la gravità.

Il progetto è stato completato entro il limite delle otto settimane lavorative previsto, tuttavia, con più tempo e un budget maggiore, sarebbe stato possi-

bile eseguire operazioni di *fine-tuning* sui modelli, migliorando ulteriormente l'efficacia dell'analisi del codice.

9.3 Conoscenze acquisite

Lo stage presso l'azienda zero12 s.r.l. ha offerto l'opportunità di acquisire competenze in nuove tecnologie, in particolare approfondendo l'uso dei servizi *AWS* ***G*** e la realizzazione di sistemi *serverless*, oltre all'integrazione della *Generative AI* ***G*** in applicazioni pratiche. Questa esperienza mi ha permesso di lavorare in un ambiente professionale, sperimentando la collaborazione tra colleghi e il supporto reciproco.
Dal punto di vista personale e professionale, ho trovato lo stage estremamente utile. Ho avuto l'opportunità di interagire con altri professionisti, migliorando sia le mie competenze tecniche sia le mie capacità di lavoro in team, tutte competenze che considero molto preziose per la mia crescita futura.

Acronimi e abbreviazioni

4o 4-omni. 4, 25, 33, 42, 49, 51–53

AI Artificial Intelligence. iv, 2, 6, 9, 17, 23, 24, 26, 28, 40, 45

API *Application Programming Interface*$_{G}$. 9, 12, 14

ARN Amazon Resource Name. 38, 48

AWS *Amazon Web Service*$_{G}$. iv, 1, 4–7, 11, 29–32, 36–38, 48, 51, 53, 54, 57

CLI Command Line Interface. 31, 32, 38

CVSS *Common Vulnerability Scoring System*$_{G}$. 19, 21–26, 28, 31, 34, 45–47, 50–53

CWE *Common Weakness Enumartion*$_{G}$. 16, 17, 19, 23, 26–28, 31, 33, 43, 44, 46, 47, 50, 52, 53

Generative AI *Generative Artificial Intelligence*$_{G}$. 2, 4, 5, 8, 26, 51, 54

GPT *Generative Pre-trained Transformer*$_{G}$. iv, 4, 7–9, 17, 23, 25, 33, 40–43, 49, 51, 53

IoT *Internet of Things*$_{G}$. 1

NPM *Node Package Manager*$_{G}$. 11

OWASP *Open Web Application Security Project*$_{G}$. 17

S3 Simple Storage Service. iv, 4, 7, 12, 31–34, 37, 38, 40, 46, 48, 51, 53, 57

TPM Tokens per Minute. 33

Glossario

Application Programming Interface Le Application Programming Interfaces sono insiemi di protocolli, strumenti e definizioni che permettono a diversi software di comunicare tra loro. Esse forniscono metodi standardizzati per richiedere e scambiare dati, consentendo l'integrazione e l'interoperabilità tra applicazioni diverse. 56

Amazon Web Service Servizio on demand di cloud computing fornito da Amazon. 56

bucket Un *bucket* di *AWS*$_G$ *S3*$_G$ è una risorsa di archiviazione nel *cloud*. Ogni *bucket* rappresenta un contenitore logico che può memorizzare dati.. 7, 12, 13, 38, 40, 42, 46, 48

Cloud Computing Cloud computing indica un'erogazione di servizi offerti su richiesta da un fornitore a un utente finale attraverso la rete internet. 1, 5

Common Vulnerability Scoring System Il Common Vulnerability Scorting System è una norma tecnica aperta per valutare la gravità delle vulnerabilità di sicurezza di un sistema informatico. CVSS assegna un punteggio di gravità alle vulnerabilità, consentendo a chi si occupa di rispondere all'emergenza di stabilire la priorità di risposte e risorse in base al livello di minaccia. 56

Common Weakness Enumeration È un catalogo di vulnerabilità software che descrive le debolezze di sicurezza comuni che possono esistere nel codice, fornisce una terminologia standard per identificare, categorizzare e comunicare le vulnerabilità, facilitando la comprensione e la gestione

dei rischi da parte degli sviluppatori, dei ricercatori di sicurezza e delle organizzazioni. 56

deployment In informatica si intende il processo di distribuzione software, ovvero l'insieme di attività che rendono il sistema software disponibile per l'utilizzo. 4, 12, 14, 30

framework In informatica e specificamente nello sviluppo software, un framework è un'architettura logica di supporto sulla quale un software può essere progettato e realizzato, facilitandone lo sviluppo da parte del programmatore. 4, 29, 30, 35

Generative Artificial Intelligence L'intelligenza artificiale generativa è un tipo di intelligenza artificiale che è in grado di generare testo, immagini, video, musica o altri media in risposta a delle richieste dette prompt. 56

Generative Pre-trained Transformer Modello di linguaggio sviluppato da OpenAI, basato sull'architettura Transformer. GPT è progettato per comprendere e generare testo naturale ed è in grado di rispondere a domande, completare frasi, tradurre testi e svolgere molte altre attività di elaborazione del linguaggio naturale. 56, 59

GitHub GitHub è un servizio di hosting per progetti software. 4, 7, 28–32, 34, 37, 48, 49, 51, 53, 59

Internet of Things Con Internet of Things, si intendono tutti i dispositivi che sono in grado di collegarsi e comunicare con altri dispositivi attraverso una rete, per esempio internet. 56

linting Linting è il processo di analisi automatizzata del codice sorgente per individuare errori di sintassi, problemi di stile, e potenziali bug. Uno strumento di linting, chiamato *linter*, esamina il codice secondo una serie di regole predefinite o configurabili, e fornisce avvisi o suggerimenti su come migliorare il codice. 14

Machine Learning Con Machine Learning, in italiano apprendimento automatico, si intende la predisposizione di una macchina ad apprendere dai dati in maniera autonoma. 1

MITRE MITRE è un'organizzazione non profit statunitense che gestisce centri di ricerca e sviluppo finanziati dal governo federale degli Stati Uniti. Fondata nel 1958, MITRE lavora principalmente per supportare le agenzie governative in aree critiche come la difesa, la sicurezza nazionale, la sanità, l'aviazione e la sicurezza informatica. L'organizzazione è conosciuta per il suo ruolo nella gestione e sviluppo di importanti framework e standard nel campo della sicurezza informatica, come ad esempio CWE (Common Weakness Enumeration) e CVE (Common Vulnerabilities and Exposures). 16

Node Package Manager NPM (Node Package Manager) è il gestore di pacchetti predefinito per Node.js. Facilita l'installazione, l'aggiornamento e la gestione delle dipendenze nei progetti JavaScript. 56, 59

Node.js Node.js è un runtime JavaScript open-source, basato sul motore V8 di Google Chrome, che consente di eseguire codice JavaScript lato server, offrendo un'ampia libreria di moduli attraverso il gestore dei pacchetti *Node Package Manager*$_{\boldsymbol{G}}$. 9

Open Web Application Security Project Comunità globale e non *profit* dedicata alla sicurezza delle applicazioni web. Fondata nel 2001, OWASP fornisce risorse aperte e strumenti gratuiti per aiutare sviluppatori e organizzazioni a costruire applicazioni più sicure. 56

OpenAI Organizzazione di ricerca e sviluppo che si occupa di intelligenza artificiale, nota per la creazione di modelli avanzati di intelligenza artificiale, tra cui i modelli di linguaggio *Generative Pre-trained Transformer*$_{\boldsymbol{G}}$. iv, 2, 4–6, 9, 10, 17, 25, 51, 52

repository Repository (di *GitHub*$_{\boldsymbol{G}}$) è un archivio strutturato che gestisce e conserva l'intero stato e la cronologia di un progetto software, incluse tutte

le versioni dei file, le modifiche apportate nel tempo e le informazioni sui contributi dei diversi sviluppatori. 4, 7, 14, 28, 29, 31, 32, 37, 47–51, 53

serverless Il framework serverless, da non confondere con il termine *serverless*, permette la gestione e il deployment di applicazioni *serverless*, dove le risorse di calcolo vengono automaticamente allocate e gestite dal provider *cloud*. 4, 29, 30, 35

Bibliografia

Siti web

[1] *AWS Lambda*. URL: https://aws.amazon.com/it/lambda/.

[2] *AWS S3*. URL: https://docs.aws.amazon.com/s3/.

[3] *AWS Step Functions*. URL: https://aws.amazon.com/it/step-functions/.

[4] *Common Weakness Enumeration*. URL: https://cwe.mitre.org/.

[5] *CVSS 3.1*. URL: https://www.first.org/cvss/v3.1/specification-document.

[6] *Framework Serverless*. URL: https://www.serverless.com/framework/docs.

[7] *GitHub Actions*. URL: https://docs.github.com/en/actions.

[8] *OpenAI - Prompt Engineering*. URL: https://platform.openai.com/docs/guides/prompt-engineering (cit. a p. 26).

[9] *OpenAI API pricing*. URL: https://openai.com/api/pricing/.

[10] *OpenAI Fine-Tuning*. URL: https://platform.openai.com/docs/guides/fine-tuning.

[11] *OpenAI model 4o*. URL: https://platform.openai.com/docs/models/gpt-4o (cit. alle pp. 25, 51).

[12] *OpenAI model 4o-mini*. URL: https://platform.openai.com/docs/models/gpt-4o-mini (cit. alle pp. 25, 51).

[13] *OpenAI models*. URL: https://platform.openai.com/docs/models (cit. a p. 10).

[14] *OpenAI rate limits*. URL: https://platform.openai.com/docs/guides/rate-limits/usage-tiers?context=tier-one (cit. a p. 33).

[15] *OWASP Top Ten 2021*. URL: https://owasp.org/Top10/.

Sitografia

[1] *AWS Lambda*. URL: https://aws.amazon.com/it/lambda/.

[2] *AWS S3*. URL: https://docs.aws.amazon.com/s3/.

[3] *AWS Step Functions*. URL: https://aws.amazon.com/it/step-functions/.

[4] *Common Weakness Enumeration*. URL: https://cwe.mitre.org/.

[5] *CVSS 3.1*. URL: https://www.first.org/cvss/v3.1/specification-document.

[6] *Framework Serverless*. URL: https://www.serverless.com/framework/docs.

[7] *GitHub Actions*. URL: https://docs.github.com/en/actions.

[8] *OpenAI - Prompt Engineering*. URL: https://platform.openai.com/docs/guides/prompt-engineering (cit. a p. 26).

[9] *OpenAI API pricing*. URL: https://openai.com/api/pricing/.

[10] *OpenAI Fine-Tuning*. URL: https://platform.openai.com/docs/guides/fine-tuning.

[11] *OpenAI model 4o*. URL: https://platform.openai.com/docs/models/gpt-4o (cit. alle pp. 25, 51).

[12] *OpenAI model 4o-mini*. URL: https://platform.openai.com/docs/models/gpt-4o-mini (cit. alle pp. 25, 51).

[13] *OpenAI models*. URL: https://platform.openai.com/docs/models (cit. a p. 10).

[14] *OpenAI rate limits*. URL: https://platform.openai.com/docs/guides/rate-limits/usage-tiers?context=tier-one (cit. a p. 33).

[15] *OWASP Top Ten 2021*. URL: https://owasp.org/Top10/.