Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica

Tesi di laurea

Supporto tecnico con l'aiuto della IA Generativa

20 Settembre 2024

Relatore

Prof. Tullio Vardanega

Laureando

Endi Hysa

Matricola n. 2046424

Anno Accademico 2023-2024

"Allez en avant, et la foi vous viendra"

— Jean d'Alembert

Ringraziamenti

Al termine di questo percorso universitario, desidero esprimere la mia gratitudine a tutte le persone che mi hanno sostenuto e che ho avuto il piacere di conoscere durante questi anni.

Desidero ringraziare il professor Tullio Vardanega, relatore della mia tesi, per il prezioso supporto che mi ha offerto durante lo *stage* e per la sua guida nella stesura di questo documento. I suoi consigli e la sua competenza sono stati una guida fondamentale in tutte le attività legate al lavoro svolto.

Un ringraziamento speciale va a tutti i colleghi di Zero12 s.r.l., con i quali ho instaurato un rapporto di collaborazione e amicizia che ha reso il mio *stage* un'esperienza formativa e professionale indimenticabile. Un grazie di cuore va anche ai compagni di corso con cui ho condiviso l'esperienza di tirocinio, per il reciproco supporto e per le numerose risate in ufficio.

Con immensa gratitudine, ringrazio dal profondo del cuore mamma Saimira, papà Bashkim e mia sorella Sara, per avermi sempre sostenuto, incoraggiato e, in quei momenti difficili, anche sopportato durante questo percorso. L'amore incondizionato che mi avete donato è stata la mia forza più grande. Un ringraziamento speciale va ancora una volta ai miei genitori, per aver compiuto la scommessa più grande della loro vita, permettondomi di costruire un futuro migliore. Sarò sempre riconoscente per il vostro sacrificio e l'amore che mi avete sempre dato.

Un ringraziamento speciale va alla mia cara nonna Lule, che mi ha sempre avvolto nel amore e affetto. E un pensiero pieno di affetto va ai miei nonni, che con orgoglio mi osservano da lassù. Il vostro amore mi accompagna oltre ogni confine, proteggendomi e guidandomi in ogni mia scelta. Mi mancate tanto.

Un grazie speciale va a Tania, che è stata al mio fianco in ogni istante, sostenendomi e incoraggiandomi. Grazie per avermi infuso forza e fiducia in ogni momento, per aver sopportato i miei momenti di stress e per avermi sempre spronato a dare il meglio di me. La tua presenza è stata un pilastro fondamentale in questo viaggio.

Infine, desidero ringraziare tutti gli amici che mi hanno accompagnato in questo percorso. In particolare ci tengo a ringraziare gli amici di sempre, i ragazzi del Concilio e il gruppo *Byte Ops*, per aver reso questi anni universitari divertenti e indimenticabili.

Padova, 20 Settembre 2024 *Endi Hysa*

Sommario

Nel presente documento descrivo il lavoro svolto durante il periodo di *stage* svolto presso l'azienda Zero12 s.r.l, avente sede a Padova, nel periodo compreso tra il 7 maggio e il 1 luglio 2024. Analizzo in primo luogo, la prima parte del progetto che ha portato alla creazione di un sistema di proposte di risoluzioni automatiche per i *ticket* inseriti all'interno del sistema di *ticketing* aziendale. Successivamente analizzo la seconda parte del progetto che ha portato alla creazione di un *chatbot* come strumento aggiuntivo simile a quello che ho presentato precedentemente.
Ho suddiviso la relazione in quattro capitoli principali, ognuno riguardante un aspetto dello *stage* svolto:

- **Capitolo 1**: informazioni generali sull'azienda ospitante. Descrivo la loro organizzazione interna, la metodologia di lavoro e le tecnologie adottate. Presento il profilo della loro clientela e della loro propensione all'innovazione;
- **Capitolo 2**: motivazioni della proposta dello *stage* da parte dell'azienda. Analizzo le loro aspettative, quali vincoli ho dovuto rispettare e le mie motivazioni personali sulla scelta del progetto di *stage*;
- **Capitolo 3**: descrivo il metodo di lavoro adottato, le attività svolte e i risultati ottenuti.
- **Capitolo 4**: obiettivi raggiunti e difficoltà che ho incontrato. Valuto le competenze che ho acquisito e il divario formativo tra università e mondo del lavoro.

Al fine di agevolare la lettura, ho seguito le seguenti convenzioni tipografiche:

- riporto in *corsivo* i termini in lingua diversa dall'italiano;
- accompagno ogni immagine da una didascalia e la relativa fonte, se non di mia creazione;

In appendice ho inserito il glossario dei termini meno comuni utilizzati all'interno del documento, insieme alla lista di abbreviazioni e acronimi e alla bibliografia e sitografia consultata.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Contesto aziendale

1.1 Profilo aziendale

Zero12 s.r.l. è una *startup* italiana fondata nel 2012, con attualmente due sedi operative: una a Padova (dove ho svolto lo *stage*) e una ad Empoli. L'azienda è *partner* Amazon Web Services (AWS), una piattaforma di servizi *cloud* offerta da Amazon, ed è specializzata nello sviluppo di soluzioni *cloud native*, offrendo ai propri clienti, proveniente da ambiti diversificati, soluzioni flessibili e scalibili. Ogni progetto viene svolto in tutte le sue parti dal *team* di Zero12, dalla progettazione fino al suo sviluppo. L'azienda è parte di Var Group S.p.A., l'operatore *leader* nel settore dei servizi e delle soluzioni digitali. Supporta le imprese nel loro percorso di trasformazione digitale, offrendo ai propri *partner* servizi e competenze per lo sviluppo di soluzioni innovative.

1.2 Organizzazione aziendale

Durante il mio periodo di *stage* ho avuto modo osservare in prima persona l'organizzazione in *team* di lavoro che l'azienda adotta.
Ho rappresentato di seguito nella tabella 1.1 i ruoli che ho potuto osservare.

Ruolo	**Descrizione**
CEO	Il Chief Executive Officer (CEO) ovvero l'amministratore delegato, rappresenta il ruolo più alto all'interno di una societa. Si occupa di definire le decisioni più importanti dell'azienda, di supervisionare i processi aziendali e di gestire i contatti con potenziali nuovi clienti.
Project manager	Il *project manager* è la figura che si occupa di gestire il progetto in tutte le sue fasi, di coordinare il *team* di lavoro e di controllare che il progetto rispetti i costi, i tempi e le aspettative del cliente.

Ruolo	Descrizione
Software developer	Il *software developer* è la figura che si occupa della progettazione e dello sviluppo delle varie componenti *software* individuate durante l'attività di analisi. Ognuno di loro si specializza nello sviluppo di funzionalità di *frontend*, *backend* o entrambi (*fullstack*)
Mobile developer	Il *mobile developer* è la figura che si occupa delle stesse mansioni del *software developer*, riguardante un'applicativo *mobile*. Nel contesto aziendale i *mobile developers* si suddividono in due categorie: *Android developers* e *iOS developers*.
AWS specialist	L'*AWS Specialist* è la figura specializzata nella configurazione, gestione e adozione dei servizi offerti da Amazon Web Services (AWS).
HR	La figura dell' Human Resources (HR) è una figura che si occupa della gestione delle risorse umane all'interno dell'azienda.

Tabella 1.1: Organizzazione dei ruoli all'interno di Zero12 s.r.l.

Ad ogni nuovo progetto viene assegnato un *team* di lavoro composto da un *project manager*, un numero variabile di *software developers* o *mobile developers*, a seconda delle esigenze del progetto, e un *AWS specialist* se il progetto prevede l'utilizzo di servizi AWS.

1.3 Servizi e prodotti

Zero12 offre vari servizi e prodotti che vengono richiesti dai propri clienti, provenienti da diversi ambiti. Di seguito mostro una panoramica di tali servizi e prodotti offerti:

- **AWS *professional services***: sin dalla sua nascita nel 2012, l'azienda propone e sviluppa soluzioni che sfruttano il *cloud* e i servizi AWS. Per questo motivo, l'azienda mette a disposizione le competenze dei suoi specialisti per supportare i clienti nella migrazione verso il *cloud* e nell'adozione dei servizi AWS.
- **Sviluppo di *software custom***: questo è un altro servizio offerto da Zero12. L'azienda si occupa della progettazione e dello sviluppo di *software custom* (applicazioni *web* e *mobile*), adattandosi alle richieste ed alle esigenze del cliente. Il mio *stage* ha seguito questa tipologia di progetto in cui l'azienda stessa era il cliente.
- ***Machine learning***: le soluzioni di *machine learning* che Zero12 offre consistono nella creazione di modelli per l'analisi di linguaggio naturale, analisi di immagini e video, creati su misura secondo le esigenze del cliente.

- ***Innovation advisory***: servizio offerto progettato per trasformare le idee dei propri clienti in servizi digitali innovativi, sfruttando al massimo le potenzialità delle tecnologie *cloud*. Questo servizio consente di sviluppare soluzioni innovative e scalabili, adattabili alle esigenze del cliente.

1.4 Processi aziendali

1.4.1 Metodologia di sviluppo

Durante il mio periodo di *stage* ho avuto modo di sperimentare in prima persona i processi istanziati dall'azienda per la realizzazione di un progetto. La metodologia adottata dall'azienda è quella *Agile*, adattandosi in particolare al *framework Scrum*. Ogni progetto viene suddiviso in *sprint* di durata fissa e ad ognuno viene assegnato un *team* di lavoro. Nel contesto del mio *stage*, la durata degli *sprint* era di una settimana e il *team* di lavoro era composta da me e dal mio *tutor* aziendale. Il ruolo che ho ricoperto all'interno del *team* era quello di *software developer*, mentre il mio *tutor* ricopriva il ruolo di *project manager*.

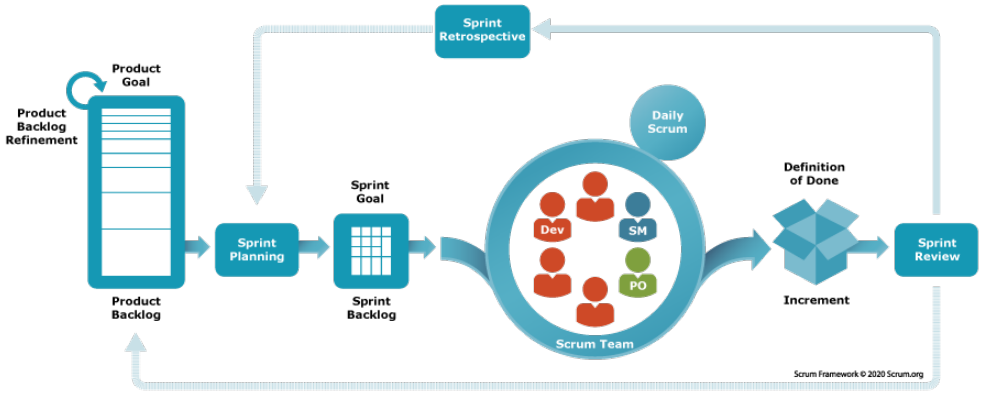

Figura 1.1: Schema di flusso di un progetto con metodologia *Scrum*
Fonte: `scrum.org`

La figura 1.1 mostra il flusso di lavoro di un progetto con metodologia *Scrum* adottato da Zero12. Ogni nuovo progetto segue questo flusso.

- *User stories*: all'inizio viene effettuata una prima sessione di *user stories mapping* che va a definire quello che sarà presente nel *product backlog*, ovvero la lista delle attività da svolgere. Questa sessione viene svolta insieme al cliente, in modo da avere una chiara visione delle aspettative del cliente, definendo cosi anche, le *feature* più importanti da sviluppare.
- ***Sprint planning***: all'inizio di ogni *sprint* vengono selezionate le *task* da svolgere, in base alla priorità, dal *product backlog* e vengono assegnate ai membri del *team* di lavoro. Gli *sprint* hanno una durata di una o due settimane.

- ***Daily stand-up***: ogni giorno, ad un orario prefissato, il *team* di lavoro si riunisce per discutere come procede il lavoro, cosa è stato fatto il giorno precedente e cosa verrà svolto durante la giornata. Hanno una durata di circa 15 minuti.
- ***Sprint review***: alla fine di ogni *sprint* viene effettuata una revisione del lavoro svolto, andando a controllare che le *task* assegnate siano completate e che il lavoro assegnato sia stato svolto in modo conforme alle aspettative del cliente.
- ***Sprint retrospective***: alla fine di ogni *sprint* viene effettuato un controllo interno del lavoro svolto durante lo *sprint*, andando ad evidenziare eventuali problematiche sorte e andando a definire eventuali miglioramenti da apportare al processo di lavoro.

1.4.2 Strumenti di supporto ai processi

Jira

Jira è un Issue Tracking Sytem (ITS) che permette la gestione delle *task* dei progetti in modo *Agile*. All'inizio di ogni *sprint*, lo *scrum master* crea una nuova *board* relativa allo *sprint* in corso e assegna le *task* (crea i *ticket*) da svolgere ai membri del *team* di lavoro. I membri del *team* di lavoro possono notificare lo stato del *ticket*, spostandolo tra le varie colonne quali:

- **Da completare**: *ticket* assegnati ma che devono essere ancora completati.
- **In corso**: *ticket* assegnati ad una o più risorse del *team* di lavoro e che sono in sviluppo.
- **Completato**: *ticket* completati e pronti per la revisione.

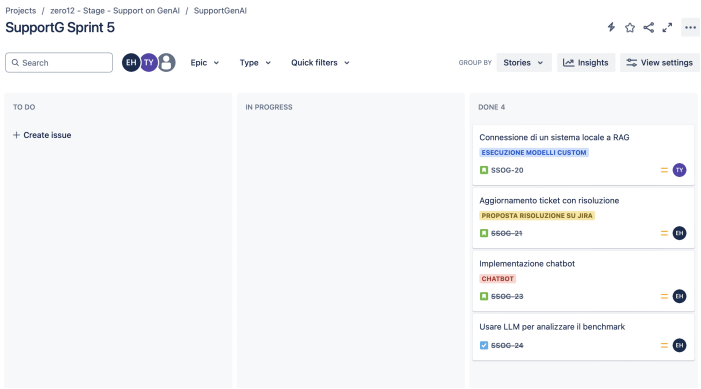

Figura 1.2: *Board* del quinto *sprint* del progetto di *stage*

Visual Studio Code

Visual Studio Code è un *editor* di codice sorgente che permette la scrittura di codice in diversi linguaggi di programmazione.

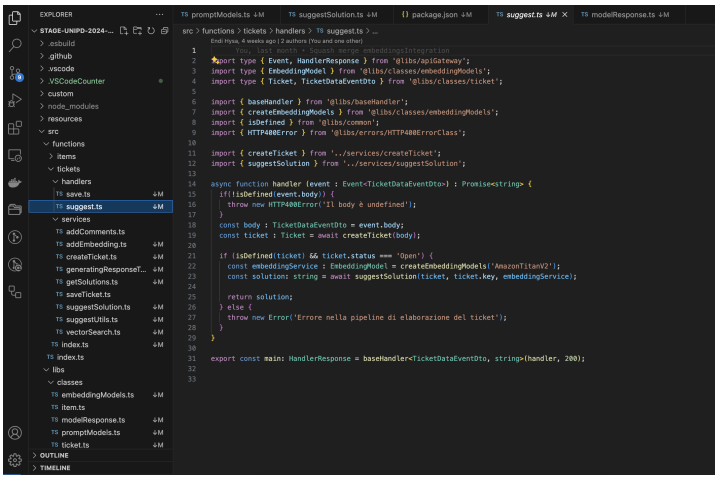

Figura 1.3: Schermata dell'*editor* Visual Studio Code con codice Typescript

Github

Github è una piattaforma *web* che permette di usufruire del sistema di versionamento Git del codice sorgente, da interfaccia *web*. Grazie a questa piattaforma , è possibile creare *repositories* in cui salvare il codice sorgente del progetto. È possibile suddividere il progetto in *branch*, rami di sviluppi separati da quello principale, assegnati a specifiche *feature* da sviluppare, permettendo cosi di lavorare in modo parallelo. Terminata la codifica di una *feature*, tramite una *pull request*, si unisce il codice sviluppato nel *branch* al codice sorgente presente nel *branch* principale. Tramite *pull request* il codice viene revisionato per garantire la sua qualità e la corretta integrazione con il codice sorgente principale.

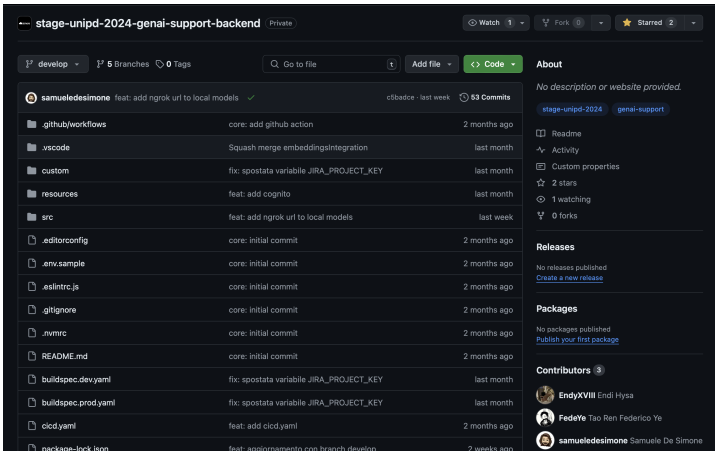

Figura 1.4: Schermata del *repository* Github del progetto di *stage*

Slack

Slack è un'applicazione di messaggistica utilizzata in ambito aziendale. Permette la creazione di canali di comunicazione dedicati a specifici argomenti. Nel contesto

aziendale, viene creato un canale dedicato ad ogni progetto in modo da poter permettere una comunicazione più efficace tra i membri del *team* di lavoro. Nel caso del mio progetto di *stage*, è stato creato un canale dedicato al progetto assieme al mio *tutor* aziendale ed un altra figura all'interno dell'azienda, con lo scopo aiutarmi in caso di problematiche durante la configurazione di alcuni servizi o durante lo sviluppo.

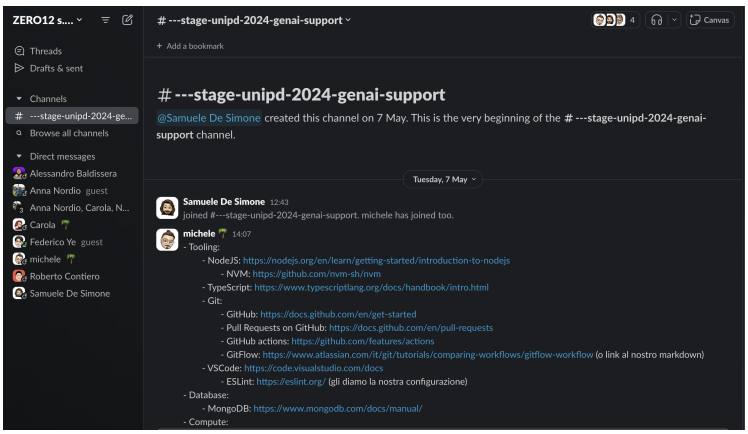

Figura 1.5: Schermata del canale dedicato al progetto di *stage*

Di seguito uno schema riassuntivo in cui mostro l'interazione degli strumenti utilizzati durante il flusso di progetto.

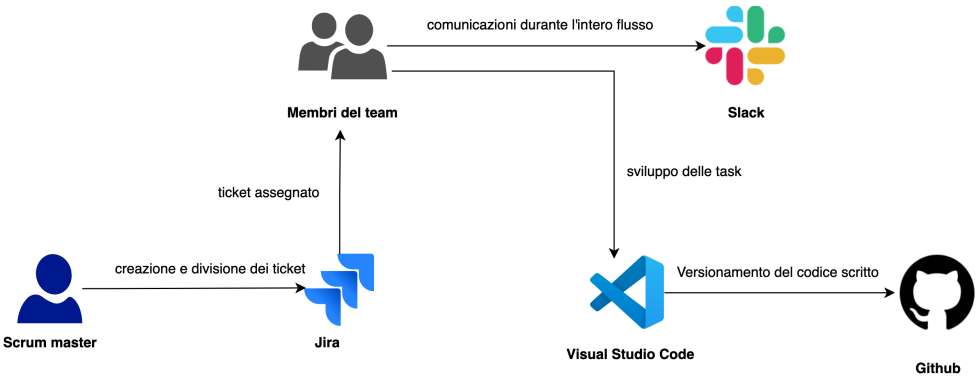

Figura 1.6: Interazione degli strumenti nel processo di sviluppo
Fonte: Visual Studio: https://commons.wikimedia.org

1.5 Tecnologie utilizzate

1.5.1 Tecnologie di sviluppo

Typescript

TypeScript è un linguaggio di programmazione, noto per essere un *superset* di JavaScript che introduce l'annotazione dei tipi. Essendo un *superset* di JavaScript, ogni programma scritto in Javascript può essere eseguito anche in Typescript.

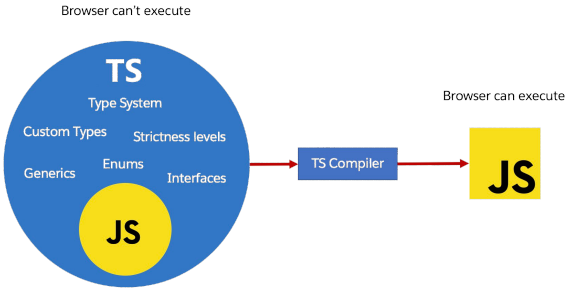

Figura 1.7: Funzionalità aggiuntive di Typescript rispetto a Javascript

Python

Python è linguaggio di programmazione interpretato, ad alto livello. È orientato agli oggetti ed è adatto a molteplici utilizzi. Offre numerose librerie che permettono lo sviluppo di varie tipologie di applicazioni.

Node.js

Node.js è un ambiente *runtime* che consente di eseguire codice Javascript e Typescript al di fuori di un *browser*. È basato sul motore JavaScript V8 e offre una libreria *standard* che comprende vari moduli per le operazioni di base.

MongoDB

MongoDB è un *database* non relazionale (*NoSQL*) dove i dati non vengono strutturati in tabelle con uno schema preciso, ma vengono raggruppati in documenti dove la struttura dei dati può variare da documento a documento. Permette di salvare oltre ai dati anche i loro relativi embedding, ovvero una rappresentazione numerica vettoriale dell'oggetto in uno spazio multidimensionale.

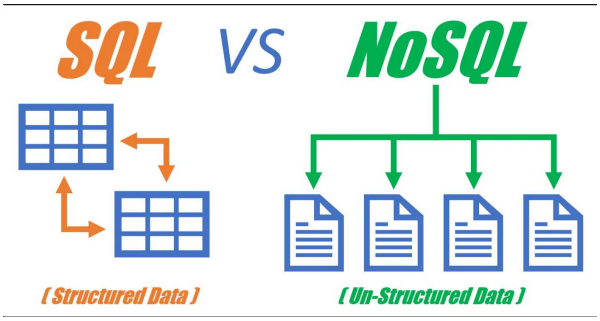

Figura 1.8: Differenza tra la struttura di un *database* relazionale e un *database* non relazionale
Fonte: medium.com

Serverless framework

Il Serverless *framework* è un *framework* che permette la creazione di Application Program Interface (API), un' interfaccia che definisce come due parti (applicazione *client* e *server*) comunicano tra loro usando richieste e risposte, su infrastuttura AWS.

Permette la definizione di funzioni *lambda* con i relativi *trigger* di invocazione e il *deploy* nel *cloud*, attraverso un semplice comando che carica il codice sorgente e configura l'architettura in modo automatico. Supporta diversi *plugin* che permettono l'utilizzo di altri servizi offerti da AWS.

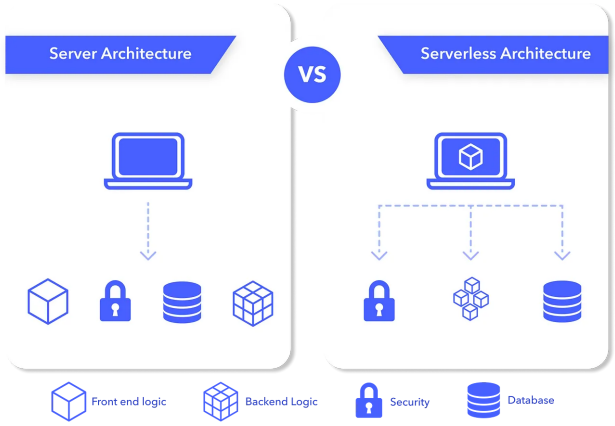

Figura 1.9: Confronto tra un'architettura tradizionale e un' architettura *Serverless*
Fonte: medium.com

Come mostro nella figura 1.9, l'architettura *Serverless* permette all'utente di non doversi preoccupare della gestione dell'infrastruttura sottostante, in quanto viene gestita automaticamente dal fornitore del servizio. Nella controparte dell'architettura tradizionale, l'utente deve gestire l'infrastruttura sottostante.

Streamlit

Streamlit è un *framework open-source* che permette la creazione di applicazioni *web* per la visualizzazione di dati in modo rapido e semplice. Con il suo utilizzo, si possono trasformare *script* Python in applicazioni *web* interattive.

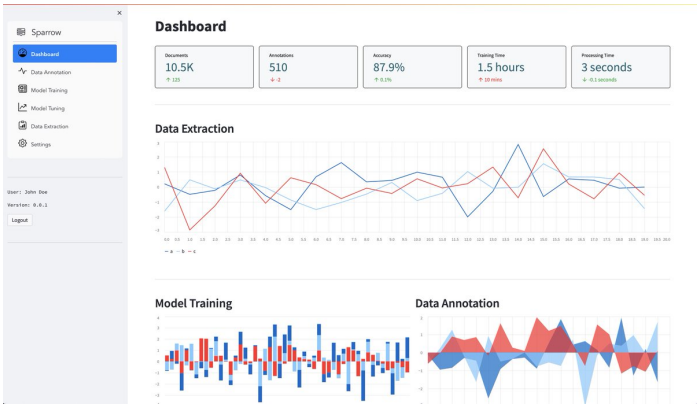

Figura 1.10: Esempio di applicazione *web* creata con il *framework* Streamlit
Fonte: streamlit.io

LangChain

Langchain è un *framework open-source* che fornisce strumenti e componenti modulari per la creazione di applicazioni, come *chatbot*, che sfruttano i Large Language Model (LLM). Questo *framework* permette un' interazione più semplice con questi modelli, permettendo di creare applicazioni che sfruttano le potenzialità dei LLM in modo più semplice. Nel contesto dello *stage*, viene utilizzato assieme al *framework* Streamlit.

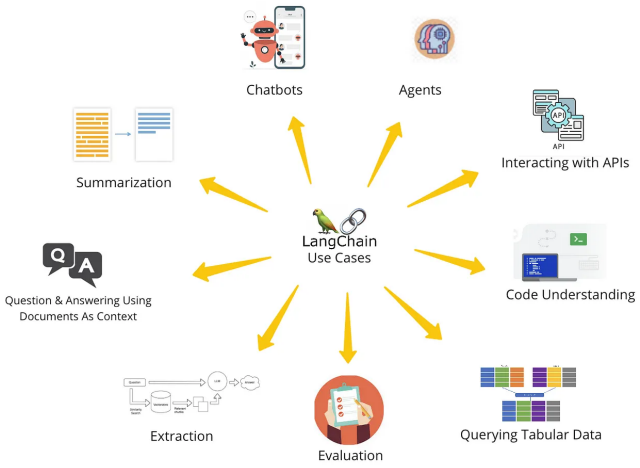

Figura 1.11: Casi d'uso con Langchain
Fonte: https://medium.com

Servizi AWS

Nella realizzazione del progetto di *stage* sono stati utilizzati diversi servizi offerti da AWS. I servizi principali utilizzati sono:

- **AWS Lambda**: servizio che permette l'esecuzione di codice senza la necessità di dover gestire *server* sottostanti. Sono autoscalabili in base al carico di lavoro. Generalmente sono funzioni che seguono il *Simple Responsability Principle*.
- **AWS Bedrock**: servizio che permette l'utilizzo di vari modelli di *machine learning* messi a disposizione, utilizzabili tramite chiamata API.
- **AWS API Gateway**: servizio che permette la creazione, la pubblicazione, la manutenzione, il monitoraggio e la protezione di API su larga scala. Si integra nell'ecosistema AWS offrendo *trigger* di eventi per l'esecuzioni di funzioni *lambda*.
- **AWS Cognito**: servizio che fornisce un sistema di autenticazione e autorizzazione per gli applicativi *web* e *mobile*. Gestisce automaticamente la registrazione di nuovi utenti, il *login*, il recupero della *password* e la gestione delle sessioni.

Alcuni servizi vengono utilizzati per attuare il processo di Continuous Integration & Continuous Delivery (CI/CD), una pratica di sviluppo in cui tutte le modifiche apportate al *software* durante lo sviluppo, vengono integrate e testate automaticamente, garantendo che il codice sia sempre funzionante e pronto per il rilascio.

1.5.2 Tecnologie per la validazione del codice

ESLint

ESLint è un *plugin* installato su Visual Studio Code che permette il controllo del codice in fase di scrittura, segnando eventuali errori di sintassi o di stile. È possibile impostare una configurazione personalizzata (come l'indentazione del codice, la lunghezza delle righe, ecc.), permettendo così di scrivere codice uniforme agli *standard* aziendali.

```
"devDependencies": {
  "@serverless/typescript": "^3.0.0",
  "@types/aws-lambda": "^8.10.71",
  "@types/node": "^14.14.25",
  "@types/uuid": "^9.0.2",
  "@typescript-eslint/eslint-plugin": "^5.55.0",
  "@typescript-eslint/eslint-plugin-tslint": "^5.55.0",
  "@typescript-eslint/parser": "^5.55.0",
  "esbuild": "^0.14.11",
  "eslint": "8.35.0",
  "eslint-config-prettier": "^8.7.0",
  "eslint-plugin-babel": "^5.3.1",
  "eslint-plugin-disable-autofix": "^3.1.1",
  "eslint-plugin-import": "^2.27.5",
  "eslint-plugin-jsdoc": "^40.0.3",
  "eslint-plugin-no-null": "^1.0.2",
  "eslint-plugin-prefer-arrow": "^1.2.3",
  "eslint-plugin-prettier": "^4.2.1",
  "eslint-plugin-promise": "^6.1.1",
  "eslint-plugin-react": "^7.32.2",
  "eslint-plugin-unicorn": "^46.0.0",
  "json-schema-to-ts": "^1.5.0",
  "serverless": "^3.38.0",
  "serverless-esbuild": "^1.23.3",
  "serverless-layers": "^2.7.0",
  "serverless-offline": "^13.5.0",
  "serverless-plugin-split-stacks": "^1.12.0",
  "ts-json-schema-generator": "^1.2.0",
  "ts-node": "^10.4.0",
  "tsconfig-paths": "^3.9.0",
  "typescript": "^4.1.3"
},
```

Figura 1.12: Esempio di configurazione di ESLint

1.5.3 Tecnologie di supporto

Postman

Postman è un *software* che permette il salvataggio e il testing di API in modo semplice. Tramite l'interfaccia è possibile organizzare le richieste e generare documentazione delle API salvate.

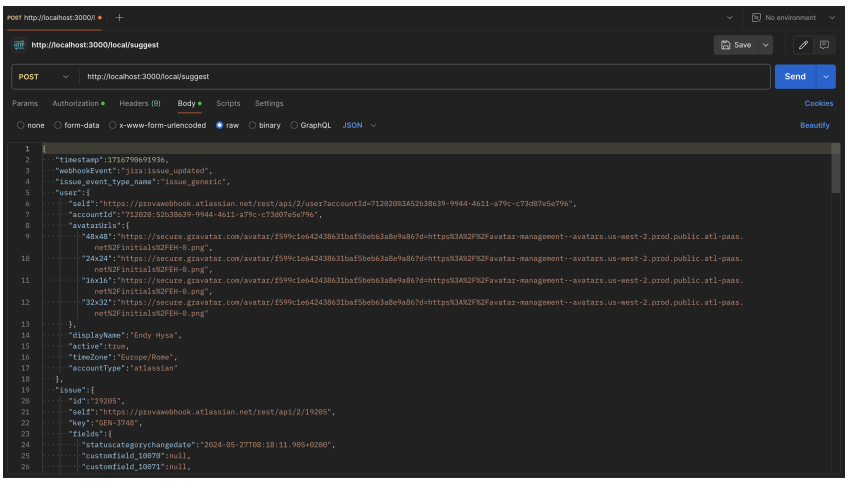

Figura 1.13: Esempio di API di tipo *POST* su Postman con relativo *body*

MongoDB Compass

MongoDB Compass è un *software* che permette la visualizzazione dei documenti salvati nel *database* MongoDB tramite un'interfaccia grafica.

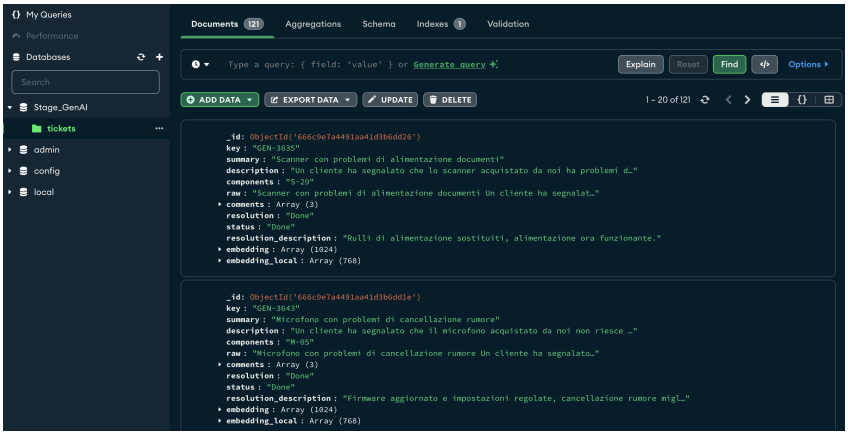

Figura 1.14: Esempio di visualizzazione di documenti su MongoDB Compass

Di seguito uno schema riassuntivo in cui mostro l'utilizzo delle tecnologie nel progetto di *stage*.

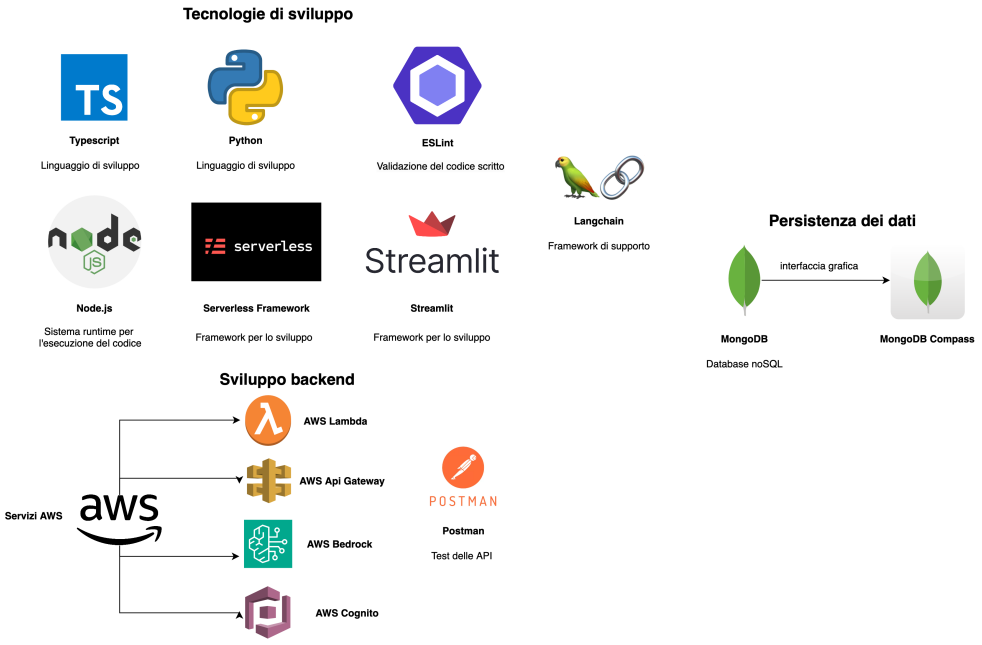

Figura 1.15: Utilizzo delle tecnologie nel progetto di *stage*
Fonti: Typescript: https://iconduck.com Nodejs: https://www.flaticon.com Python: https://www.iconfinder.com Eslint: https://iconduck.com Serverless: https://www.serverless.com AWS: https://iconduck.com Langchain: https://www.nuget.org

1.6 Tipologia di clientela

Durante il mio periodo di *stage* , ho avuto modo di osservare la diversità dei clienti che si rivolgono a Zero12 s.r.l. Tra i committenti dei vari progetti sviluppati dall'azienda, ci sono numerose aziende operanti nel medesimo settore, le quali richiedono consulenze sulle tecnologie di cui l'azienda è *partner*. L'azienda collabora anche con clienti provenienti da settori diversi, come ad esempio il settore della moda, con *brand* di lusso che richiedono lo sviluppo di software *custom* per la gestione dei propri prodotti o servizi.

1.7 Propensione all'innovazione

L'azienda è fortemente orientata all'innovazione e alla sperimentazione di nuove tecnologie. L'azienda si propone ai suoi nuovi clienti come *Innovation advisor* riuscendo a sviluppare soluzioni innovative in linea con le esigenze del cliente. Essendo *parter* AWS, l'azienda è particolarmente incentrata nella ricerca di innovazione in ambito *cloud*. In aggiunta, l'azienda organizza eventi interni per discutere o assistere alla presentazione di nuove tecnologie, portando cosi nuove idee anche non inerenti all'ambito *cloud*. Un esempio, è stato la partecipazione collettiva in sede aziendale, all'evento annuale Apple *Worldwide Developers Conference* organizzato da Apple. In questa occasione, i presenti hanno assistito alla presentazione delle nuove funzionalità presentate e discusso le loro possibili applicazioni in progetti futuri.

Capitolo 2

Lo *stage*

2.1 Il ruolo degli stage nell'azienda

Zero12 crede fortemente nella collaborazione tramite *stage* con gli studenti dell'Università di Padova. Grazie all'evento StageIT, un evento promosso da Confindustria Veneto Est e l'Università degli studi di Padova, l'azienda permette agli studenti di svolgere progetti di *stage* presso la loro sede. Gli stage per l'azienda, rappresentano una doppia opportunità. Da una parte, hanno modo di inserire nuove figure professionali nel proprio organico, dall'altra gli studenti hanno la possibilità di inserirsi nel mondo del lavoro e poter applicare le proprie conoscenze acquisite durante il percorso di studi. Durante le otto settimane di *stage* ho avuto l'occasione di conoscere quasi tutti i membri dell'azienda, comprendere il loro metodo di lavoro e mettere in pratica le mie conoscenze. Il mio percorso è stato guidato dal *tutor* aziendale che mi è stato assegnato, che coordinava le attività da svolgere e che mi forniva supporto in caso di dubbi o problemi. L'accoglienza e l'ambiente di lavoro sono molto positivi, dimostrando una grande attenzione nei confronti degli stagisti. Questa attenzione è testimoniata anche dalla presenza di molti dipendenti che attualmente lavorano in azienda dopo aver svolto uno *stage* in Zero12. I progetti assegnati durante gli *stage* vengono concepiti in base a delle reali esigenze interne all'azienda. Essi sono mirati a migliorare la comunicazione tra i collaboratori, automatizzare delle attività manuali e a migliorare la qualità del lavoro svolto.

2.2 Introduzione ai progetti

Durante il mio *stage* ho avuto modo di lavorare non solo al progetto inizialmente proposto, ma anche allo sviluppo di un progetto aggiuntivo, strettamente correlato al primo. Lo sviluppo di questi due progetti rappresenterebbe un grande vantaggio per l'azienda, ovvero:

- **Prototipi funzionanti**: grazie allo sviluppo di entrambi i progetti, l'azienda avrebbe a disposizione due prototipi funzionanti che dimostrano tutte le potenzialità dei sistemi. Entrambi i progetti utilizzato tecnologie e servizi con cui l'azienda è *partner*.
- **Presentazione ai cliente**: nel caso in cui un cliente si rivolgesse a Zero12 per implementare sistemi simili a quelli che ho sviluppato, l'azienda avrebbe già un

prototipo da presentare, che grazie alla sua flessibilità, può essere facilmente adattato alle specifiche del cliente.

- **Utilizzo di IA Generativa**: l'utilizzo della IA Generativa, un ramo dell'intelligenza artificale in grado di generare testo coerente al contesto, è un punto importante per entrambi i progetti. Questa tecnologia sta diventando sempre più diffusa in diversi ambiti e grazie al suo utilizzo garantisce che questi sistemi siano all'avanguardia.

Di seguito descrivo il progetto inizialmente proposto dall'azienda all'evento StageIT e il progetto aggiuntivo concepito e sviluppato durante lo *stage*.

2.2.1 Sistema di proposte di risoluzioni a *ticket* Jira

L'idea del progetto nasce come strumento che l'azienda può utilizzare per velocizzare il proprio supporto tecnico interno, con la possibilità di essere estesa anche ai suoi clienti. Per comprendere l'idea del progetto, immaginiamo un'azienda che produce e commercializza componenti *hardware*. Nel caso di problematiche con la componente acquistata, il cliente chiama il servizio clienti dell'azienda e l'operatore inserisce un *ticket* all'interno dell'Issue Tracking System (ITS) aziendale, che nel contesto del progetto è Jira. Immaginando questa iterazione tra cliente e addetto al servizio clienti per molte volte nel corso degli anni, i *ticket* inseriti all'interno del sistema saranno numerosi e molto simili tra loro. Questo implica che in caso di nuovi *ticket* inseriti, è molto probabile che ci siano *ticket* simili già risolti in passato. Ed è qui che da Zero12 nasce l'idea di creare un sistema di proposte di risoluzioni automatiche per i *ticket* inseriti all'interno di un ITS, che permetta di velocizzare il processo di risoluzione del problema. Le proposte di risoluzione vengono generate grazie all'utilizzo della IA Generativa, in base al contesto dei *ticket* risolti in precedenza. Di seguito in figura 2.1 mostro uno schema che rappresenta l'idea del progetto principale.

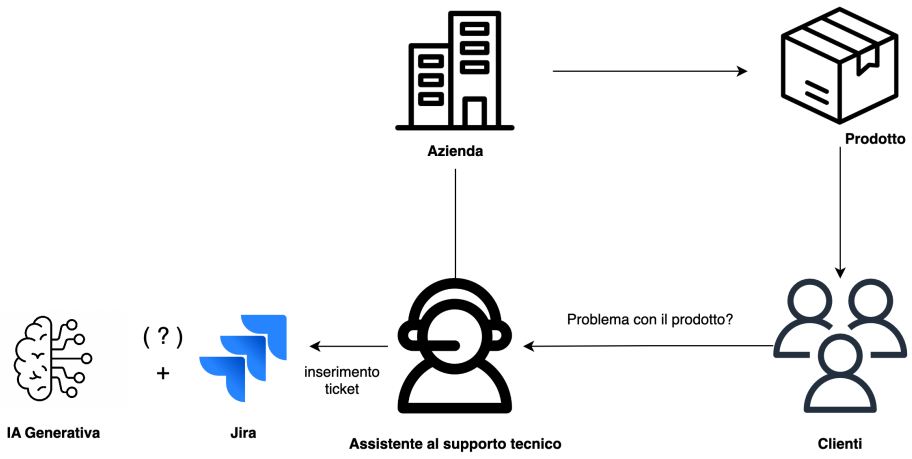

Figura 2.1: Schema che rappresenta l'idea del progetto principale
Fonte: Icona azienda: https://www.flaticon.com Icona prodotto: https://www.flaticon.com Icona assistente: https://www.flaticon.com Icona IA Generativa: https://www.creativefabrica.com

Benefici del sistema di proposte di risoluzioni

Il sistema di proposte di risoluzioni a *ticket* Jira, permette di ottenere diversi benefici, qui di seguito elencati:

- **Risposte più veloci**: gli addetti al supporto tecnico possono risolvere le nuove problematiche più velocemente, grazie alle proposte di risoluzione che il sistema genera.
- **Apprendimento continuo**: il sistema permette di apprendere continuamente dai *ticket* risolti in passato, integrando nuove informazioni e conoscenze per generare proposte di risoluzione sempre più precise e dettagliate.
- **Supporto decisionale**: il sistema permette di supportare gli addetti al supporto tecnico nella presa di decisioni, fornendo proposte di risoluzione che possono essere utilizzate come spunto per risolvere il problema.
- **Vantaggi del *cloud***: il sistema è stato sviluppato utilizzando le tecnologie e i servizi di cui l'azienda è *partner*. Come descritto nella sezione 1.1, l'azienda è specializzata nello sviluppo di applicazioni *cloud*, e il sistema di proposte di risoluzioni sfrutta proprio queste tecnologie. Questo permette di avere un sistema scalabile e flessibile.

2.2.2 *Chatbot* per proposte di risoluzioni

Durante lo sviluppo del progetto descritto nel paragrafo precedente, da parte del mio *tutor* aziendale, è nata l'idea di creare un ulteriore strumento che permetta di svolgere la stessa funzionalità del sistema di proposte di risoluzioni, ma in modo differente. Lo strumento aggiuntivo è un *chatbot* che permette di generare proposte di risoluzione, in base al testo di un *ticket* inserito all'interno della chat. Sempre grazie all'utilizzo della IA Generativa, il *chatbot* genera proposte di risoluzione in base al contesto dei *ticket* risolti in precedenza. A differenza del sistema Jira, il *chatbot* propone le risoluzioni in modo più descrittivo e dettagliato, e permette di interagire con l'assistente virtuale, per ottenere informazioni più dettagliate sulle proposte di risoluzione.

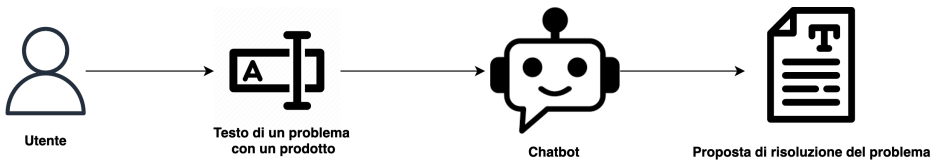

Figura 2.2: Schema che rappresenta l'idea del progetto aggiuntivo
Fonte: Icona chatbot: https://uxwing.com Icona *input* : https://www.iconfinder.com Icona testo: https://www.flaticon.com

Benefici del *chatbot*

Il *chatbot* per proposte di risoluzioni, pur non disponendo dei vantaggi del *cloud*, permette di ottenere diversi benefici oltre a quelli che il sistema di proposte precedentemente descritto possiede. Tali benefici sono qui di seguito elencati:

- **Interazione più naturale e intuitiva**: il *chatbot* permette di interagire con l'assistente virtuale attraverso un'interfaccia intuitiva e semplice. L'utente può

porre domande specifiche, richiedere chiarimenti sulle proposte e ottenere risposte immediate.

- **Flessibilità e adattabilità**: il *chatbot* potrebbe essere integrato in diversi canali di messagistica, rendendolo uno strumento flessibile e adattabile alle esigenze dell'utente. Nel contesto del progetto, il *chatbot* potrebbe assistere il cliente per risolvere le problematiche più semplici senza la necessità di contattare il servizio clienti.
- **Informazioni più dettagliate**: Il chatbot, grazie alla sua capacità di generare risposte descrittive e dettagliate, può fornire agli operatori informazioni più approfondite rispetto alle semplici proposte di risoluzione generate dal sistema Jira.

2.3 Obiettivi

2.3.1 Obiettivi aziendali

Durante i primi giorni di *stage* ho definito assieme al mio *tutor* aziendale gli obiettivi del progetto. Di seguito nella tabella elenco gli obiettivi da raggiungere:

Obiettivo	**Importanza**
Studio delle tecnologie e studio di fattibilità	Obbligatorio
Creazione dati di *mock* da inserire nell' ITS Jira	Obbligatorio
Reperimento dei ticket su Jira e salvataggio su *database* mongoDB	Obbligatorio
Aggiornamento costante del *database* a nuovi *ticket* risolti su Jira	Obbligatorio
Eseguire Retrieval Augmented Generation (RAG), una tecnica avanzata di elaborazione di linguaggio naturale, sui *ticket* salvati	Obbligatorio
Benchmark su vari modelli per la generazione della risposta	Obbligatorio
Creazione del sistema di proposte di risoluzione su Jira	Obbligatorio

Tabella 2.1: Obiettivi da raggiungere per il sistema di proposte di risoluzione su Jira

Durante lo sviluppo del progetto, grazie il buon andamento che ho avuto, al mio *tutor* aziendale è nata l'idea di sviluppare il progetto aggiuntivo relativo al *chatbot* per le proposte di risoluzione. Di seguito elenco gli obiettivi extra da raggiungere:

Obiettivo	Importanza
Studio delle tecnologie e studio di fattibilità	Desiderabile
Implementazione interfaccia *web* per l'interrogazione del *chatbot*	Desiderabile
Impostazione di un formato di domanda dell'utente da seguire	Desiderabile

Tabella 2.2: Obiettivi da raggiungere del progetto *chatbot*

2.3.2 Obiettivi personali

Prima dell'inizio dello *stage*, mi sono prefissato degli obiettivi che riguardassero gli aspetti che volevo curare durante il tirocinio. Di qui seguito elenco gli aspetti che ho tenuto in considerazione:

- **Conoscenza delle metodologie aziendali**: acquisire nuove conoscenza e competenze riguardanti le metodologie di sviluppo *software* e dei processi aziendali.
- **Conoscenze tecniche**: sviluppare competenze in merito alle tecnologie utilizzate, permettondomi di mettere in pratica le conoscenze acquisite durante il mio percorso di studi
- ***Problem solving***: migliorare il mio approccio alla risoluzione dei problemi, migliorando la mia capacità di analisi e di risoluzione rispettando i vincoli imposti.
- **Introduzione al mondo del lavoro**: Comprendere le dinamiche e l'organizzazione lavorativa di un'azienda operante in questo settore.

Questa è stata la mia prima vera esperienza lavorativa in un'azienda del settore informatico, e ho voluto sfruttare al massimo questa opportunità per crescere professionalmente e personalmente. Nella seguente tabella, riporto gli obiettivi personali secondo la seguente notazione:

OP-I

dove:

- **OP**: abbreviazione di Obiettivo Personale
- **I**: numero intero positivo incrementale, identificativo dell'obiettivo

Codice Obiettivo	Descrizione dell'Obiettivo Personale
OP-1	Sviluppare competenze con strumenti di comunicazione e collaborazione aziendali come Slack e GitHub

Codice Obiettivo	Descrizione dell'Obiettivo Personale
OP-2	Sviluppare competenze con i *framework* utilizzati, come Serverless *framework* e Streamlit
OP-3	Sviluppare competenze con nuovi linguaggi di programmazione come TypeScript e Python.
OP-4	Sviluppare competenze con le tecnologie offerte da AWS.
OP-5	Partecipare ai processi di sviluppo *software* in ambito aziendale.
OP-6	Comprendere i ritmi e le dinamiche di un lavoro in questo settore.

Tabella 2.3: Obiettivi personali durante lo *stage*

2.4 Vincoli dello *stage*

2.4.1 Vincoli tecnologici

I vincoli tecnologici imposti dall'azienda includevano l'utilizzo delle tecnologie utilizzate nello *stack* aziendale, descritte nella sezione 1.5. In aggiunta, mi è stato chiesto di utilzzare le API che fornisce Jira, per il caricamento di *ticket* di *mock* all'interno di un progetto Jira, e per il recupero dei *ticket* risolti da inserire nel *database*. Mi è stato inoltre chiesto di utilizzare i *webhook* di Jira, per impostare un *trigger* che invii eventi al sistema che ho sviluppato, in modo che alla chiusura di un *ticket* su Jira, il sistema aggiorna automaticamente il *database* con il nuovo *ticket* completato.

2.4.2 Vincoli di rendicontazione

Durante il mio periodo di *stage*, l'azienda mi ha richiesto la stesura di una tabella riassuntiva dei risultati del *benchmark* dei vari modelli testati. I risultati dei vari modelli sono stati assegnati secondo metriche di valutazione da me impostate e motivate, descritti nella sezione 3.4.1. Mi è stato richiesta la stesura di un documento in cui spiego le scelte adottate durante lo sviluppo del progetto, indicando le componenti sviluppate, le librerie utilizzate e le motivazioni dietro le scelte fatte. Mi è stata richiesta anche la stesura di un manuale utente, in cui spiego come configurare correttamente il sistema su Jira, e come utilizzarlo. Quest'ultimo documento comprende anche una sezione in cui descrivo il corretto utilizzo del *chatbot*.
Durante l'ultima settimana infine, mi è stato richiesto di preparare una presentazione sui progetti sviluppati, che avrei dovuto esporre a tutti i membri presenti dell'azienda. Questa presentazione doveva contenere le idee dei progetti, la loro importanza, i risultati ottenuti e le possibile evoluzioni future.

2.4.3 Vincoli temporali

Il progetto di *stage* è stato ideato per essere sviluppato in otto settimane, limite massimo di tempo per lo svolgimento del tirocinio curricolare. Nello specifico, è stato programmato in giornate lavorative da 8 ore ciascuna, per un totale di non più di 320 ore, suddivise in 40 ore settimanali.

2.4.4 Vincoli organizzativi

L'organizzazione dello *stage* è stata fondamentale per garantire un percorso valido e proficuo. Questa organizzazione provvedeva un costante allineamente tra gli attori che di seguito elenco:

- ***Tutor* aziendale**: durante tutto il periodo di *stage*, il contatto con il *tutor* aziendale doveva essere costante, al fine di garantire un monitoraggio costante dell'avanzamento del progetto che gli obiettivi prefissati venissero raggiunti.
- **Comunicazioni con il relatore**: il contatto con il relatore universitario doveva essere costante, in modo da allineare l'andamento dello *stage* e il suo stato di avanzamento. A tale scopo, ogni 5 giorni lavorativi, inviavo un resoconto delle attività svolte, degli obiettivi raggiunti e delle pianificazioni per il periodo successivo.

2.5 Scelta dello *stage*

Durante l'evento StageIT ho avuto modo di conoscere numerose azienda, e di ascoltare i loro progetti di *stage*. La scelta del progetto di Zero12 è stata basata su una serie di fattori che descrivo qui di seguito:

- **Tema del progetto**: il progetto sul supporto tecnico facilitato con l'aiuto della IA Generativa, come descritto nel paragrafo 2.2.1, mi ha subito colpito, in quanto mi avrebbe permesso di approfondire l'utilizzo di questa tecnologia, sempre più diffusa in vari settori. Ho scelto questo progetto anche per la sua utilità in ambito aziendale e la possibilità di poterlo estendere a clienti reali.
- **Prima impressione**: come ho descritto all'inizio del paragrafo, ho avuto modo l'opportunità di parlare con diverse aziende. Il colloquio che ho avuto con Zero12 è stato quello che mi è rimasto più impresso, in quanto si sono dimostrati molto disponibili nel rispondere alle mie domande, molto chiari nelle loro descrizioni e molto accoglienti. Dopo il colloquio, e a seguito dell'ottima impressione che ho ricevuto, mi sono interessato maggiormente al progetto da loro proposto.
- **Tecnologie utilizzate**: i giorni prima l'evento, ho avuto modo di esaminare le tecnologie richieste per lo sviluppo del progetto. Tra queste, vi erano tecnologie a me già familiari, come Typescript, Node.js e MongoDB. Il restante delle tecnologie, erano legate ai servizi *cloud* offerte da AWS, che ho avuto modo di accennare durante il mio percorso scolastico. Ho scelto questo progetto anche per la possibilità di approfondire queste tecnologie e di poterle utilizzare in ambito aziendale.
- ***Cloud* e IA Generativa**: come descritto nella sezione 1.1, Zero12 è specializzata nello sviluppo di applicazioni *cloud* ed è *partner* AWS. Il progetto inizialmente

proposto, utilizzano tecnologie e servizi AWS, e IA Generativa. Questo aspetto mi ha colpito molto, in quanto mi avrebbe permesso di mettere mano a questo tipo di tecnologie e di poterle utilizzare in un contesto lavorativo.

Nelle motivazioni riguardanti la scelta dello *stage*, non ho tenuto in considerazione la posizione geografica delle aziende. L'obiettivo principale era quello di selezionare un progetto con un aspetto formativo e professionale, che mi permettesse di approfondire le mie conoscenze e di metterle in pratica in un contesto lavorativo. Nel caso specifico di Zero12, la sede aziendale distra a circa 40 minuti di auto da dove risiedo.

Capitolo 3

Svolgimento dello *stage*

3.1 Flusso di lavoro

3.1.1 Pianificazione delle attività

In accordo agli obiettivi descritti in 2.3, ho redatto, insieme al *tutor* aziendale, una pianficazione settimanale delle attività da svolgere. Nella tabella 3.1 presento la pianificazione di tali attività.

Periodo	**Descrizione attività**
Prima settimana	- Studio ed apprendimento delle tecnologie di sviluppo - Definizione delle *user stories*
Seconda settimana	- Creazione *ticket* di *mock* - Caricamento dei *ticket* all'interno dell'ITS Jira
Terza settimana	- Reperire i ticket completati da Jira - Salvataggio *ticket* su *database* MongoDB - Aggiornamento costante del *database*
Quarta settimana	- Studio embedding per RAG - Tokenizzazione, ovvero la divisione del testo in unità che l'LLM elabora, dei *ticket* contenuti nel *database* MongoDB - Connessione tra AWS Bedrock e MongoDB per RAG
Quinta settimana	- Creazione di domande di *benchmark* - Confronto tra i diversi LLM, modelli progettati per generare testo umano
Sesta settimana	- Aggiornamento del *ticket* Jira con la proposta di risoluzione, generata tramite il sistema di IA Generativa - Sviluppo di un *chatbot* per l'interrogazione sui *ticket* Jira

Periodo	Descrizione attività
Settima settimana	- Migliorie al sistema di proposte di risoluzione Jira - Migliorie al *chatbot*, dando all'utente la possibilità di utilizzare LLM diversi - Implementare un sistema di autenticazione nel *chatbot*
Ottava settimana	- Realizzazione di una presentazione sui progetti sviluppati - Redazione documento manuale utente - Redazione documento sull'architettura e sulle scelte progettuali adottate - Miglioria al *chatbot* per proposte di risoluzione più descrittive

Tabella 3.1: Pianificazione del progetto di *stage*

Qui di seguito presento nell'immagine 3.1 che rappresenta la pianificazione delle attività svolte durante il periodo di *stage* tramite un diagramma di Gantt.

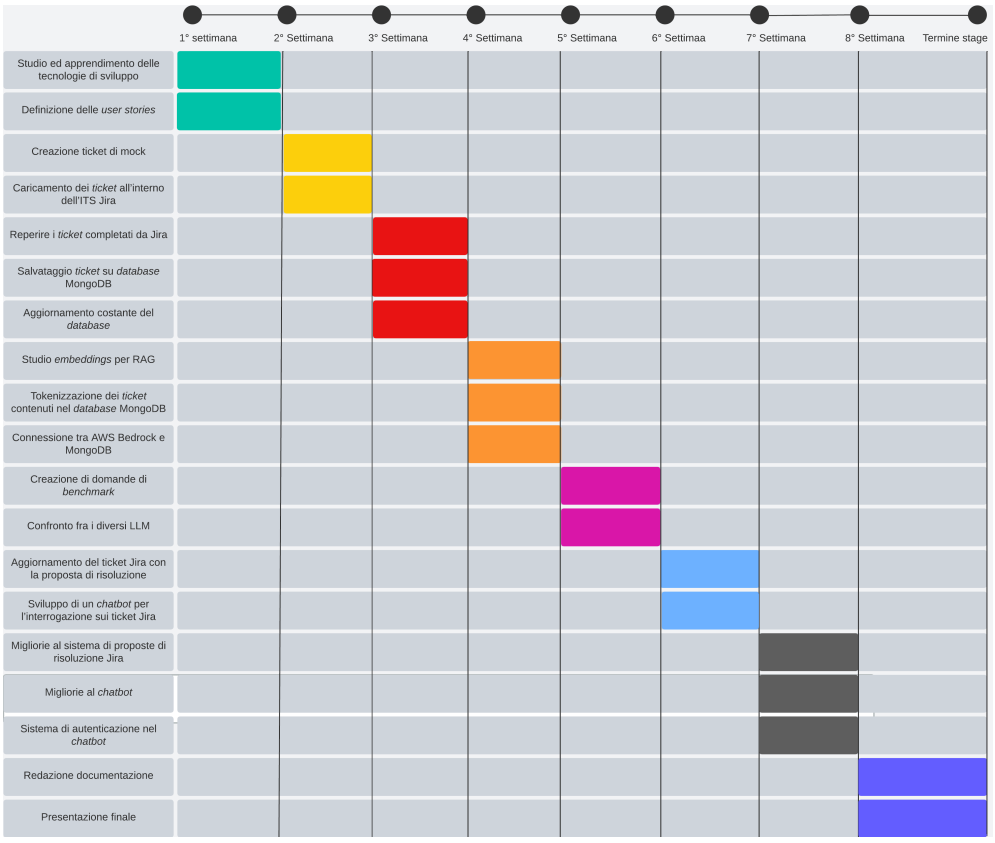

Figura 3.1: Diagramma di Gantt delle attività svolte durante lo *stage*

3.1.2 Revisioni di progetto

L'attività di revisione del progetto ritengo essere stata di fondamentale importanza e utile ai fini formativi dello *stage*. Nel corso del tirocinio ho avuto modo di sperimentare due tipologie di revisione: il *daily*, un incontro giornaliero con il *tutor* aziendale, e lo *sprint review*, un incontro settimanale con il *tutor* aziendale per valutare quanto e cosa avevo svolto durante la settimana e per pianificare le attività per la settimana successiva. L'occorrenza del *daily* è stata purtroppo però limitata a causa di numerosi impegni che il *tutor* aziendale aveva durante la giornata. Sono stati comunque utili, soprattutto all'inizio dello *stage*, per chiarire dubbi su come procedere con le attività e per ricevere feedback immediato su quanto svolto. Lo *sprint review* invece è stato quello presente con regolarità durante tutto il periodo di *stage*, che mi ha permesso di ricevere un feedback più dettagliato su quanto avevo svolto durante la settimana. Questi incontri periodici mi hanno permesso di correggere eventuali errori e di migliorare la qualità del lavoro svolto.

3.1.3 Interazioni con il tutor aziendale

Come descritto nella sezione 3.1.2, le interazioni con il *tutor* aziendale sono state numerose e di fondamentale importanza per il corretto svolgimento delle attività. Nelle giornate in cui non era possibile riferirmi direttamente al *tutor* aziendale, mi sono rivolto ad una figura di supporto, che in caso di assenza o indisponibilità del *tutor* aziendale, era predista all'aiuto degli stagisti in caso di problematiche.

3.2 Analisi dei rischi

I primi giorni di stage sono stati dedicati all'individuazione dei potenziali rischi che avrei potuto incontrare durante lo svolgimento dello *stage*. Si farà riferimento ai rischi individuati seguendo la seguente notazione:

R-[Tipologia]-[Probabilità][Impatto]-[Indice]

dove:

- **R**: abbreviazione di Rischio
- **Tipologia**:
  Natura del rischio:
  - **T**: Tecnologico
  - **O**: Organizzativo
  - **P**: Personale
- **Probabilità**:
  Numero intero positivo che indica la probabilità di occorrenza del rischio:
  - **1**: Alta
  - **2**: Media
  - **3**: Bassa
- **Impatto**:
  Valore alfabetico che indica la probabilità di occorrenza del rischio:

  - **A**: Alto
  - **B**: Medio
  - **C**: Basso

- **Indice**: Numero intero positivo incrementale che determina univocamente il rischio relativamente ad una specifica tipologia

Codice	Descrizione	Mitigazione
R-T-1A-1	Complessità del contesto di uso di servizi di IA Generativa	Chiedere supporto al *tutor* aziendale sul corretto utilizzo dei servizi di IA Generativa che verranno utilizzati
R-T-2B-2	Complessità del contesto di sviluppo e attività di integrazione con la *suite* Atlassian	- Riferimento a documentazione più dettagliata - Supporto da parte del *tutor* aziendale
R-T-1A-3	Ridotta conoscenza dei servizi cloud AWS	- Studio approfondito dei servizi AWS - Documentazione dettagliata dei servizi AWS - Supporto da parte del *tutor* aziendale
R-P-3C-1	Assenze per motivi di salute o personali	Avvisare per tempo il *tutor* aziendale in caso di ritardi, in modo da aggiornare la pianificazione delle attività
R-O-2A-1	Lo sviluppo del progetto potrebbe non essere portato a termine nel periodo pianificato a causa di ritardi	Avvisare per tempo il *tutor* aziendale in modo da pianificare le attività per recuperare i giorni di assenza

Tabella 3.2: Analisi dei rischi

3.3 Analisi dei requisiti

3.3.1 Aspettative del committente

In accordo agli obiettivi descritti in 2.3, mi è stato richiesto di sviluppare, a seguito di uno studio di fattibilità, un sistema integrato su Jira che alla creazione di un nuovo *ticket*, venisse generata una proposta di risoluzione in base ai *ticket* completati in passato e salvati in un *database* MongoDB. Inoltre, mi è stato richiesto di sviluppare un *chatbot* che permettesse all'utente di interrogare l'assistente virtuale sui *ticket* Jira e di ricevere proposte di risoluzione.

3.3.2 Casi d'uso

I diagrammi dei casi d'uso permettono di descrivere le interazione tra gli attori e il sistema. La convezione che ho utilizzato per la stesura dei casi d'uso è stata la seguente:

UC[Numero]: nome del caso d'uso

- **UC**: acronimo di *Use Case*;
- **Numero**: numero progressivo del caso d'uso. I sottocasi vengono rappresentati nella forma [Numero].[SottoNumero];

a cui ho associato una descrizione, la lista degli attori coinvolti, le precondizioni e le postcondizioni affinche il caso d'uso possa avvenire. Per riassumere i casi d'uso, sono presenti dei diagrammi Unified Modeling Language (UML) per una rappresentazione grafica degli scenari. Verrano presentati i diagrammi dei casi d'uso principali e non banali dei progetti.

Attori

Nel contesto dei casi d'uso, con un attore si definisce chi o cosa interagisce con la specifica funzionalità descritta dal caso d'uso. Gli attori che ho individuato si suddividono in 2 categorie:

- **Attori principali**
  - **Utente generico**: utente che non ha effettuato l'autenticazione nel sistema e nel *chatbot*. Ha accesso unicamente alla pagina di *login*;
  - **Utente amministratore**: utente che ha effettuato l'autenticazione. Ha accesso a tutte le funzionalità del sistema.
- **Attori secondari**:
  - **LLM**: è un attore secondario che permette la generazione di embedding e di generazione di testo tramite modelli di linguaggio. Questo attore a sua volta può essere interpretato da AmazonTitanV2 (modello di generazione di embedding offerto da AWS Bedrock) e da Claude3.5Sonnet (LLM offerto da AWS Bedrock).

Figura 3.2: Attori dei casi d'uso

Lista dei casi d'uso

Sistema di proposte di risoluzione Jira

UC0: Creazione di un nuovo ticket

Attori principali: Utente amministratore
Precondizioni: L'utente amministratore è autenticato nel *workspace* Jira

Descrizione: L'utente amministratore crea un nuovo *ticket* all'interno del progetto Jira
Postcondizioni: Il *ticket* è stato creato

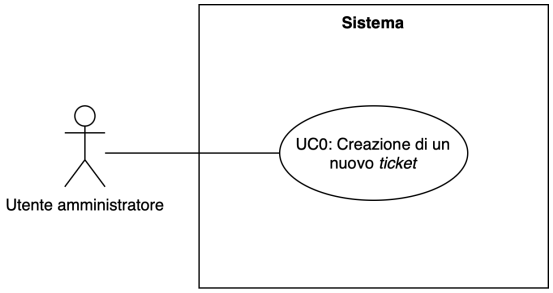

Figura 3.3: Diagramma del caso d'uso UC0

UC1: Richiesta proposta di risoluzione

Attori principali: Utente amministratore
Precondizioni: L'utente amministratore ha creato un nuovo *ticket*
Descrizione: Viene richiesto al sistema di generare una proposta di risoluzione per il *ticket* appena creato
Postcondizioni: Il sistema inizia il processo di generazione della proposta di risoluzione

UC1.1: Generazione embedding del ticket

Attori principali: LLM
Precondizioni: Il *ticket* è stato creato ed è stata richiesta una proposta di risoluzione
Descrizione: Viene richiesto all'LLM di generare l'embedding del *ticket* per effettuare la ricerca dei *ticket* completati più simili
Postcondizioni: L'embedding del *ticket* è stato generato

UC1.2: Ricerca *ticket* più simili

Attori principali: Sistema
Precondizioni: È stato generato l'embedding del *ticket* creato
Descrizione: Il sistema effettua una ricerca vettoriale all'interno del *database* con l'utilizzo dell'embedding, per trovare i *ticket* completati più simili al *ticket* creato
Postcondizioni: I *ticket* completati più simili vengono restuiti dal sistema

UC1.3: Generazione proposta di risoluzione

Attori principali: LLM
Precondizioni: Sono stati restituiti i *ticket* completati più simili al *ticket* creato
Descrizione: Viene richiesto all'LLM di generare una proposta di risoluzione per il *ticket* creato usando come contesto i *ticket* completati più simili
Postcondizioni: La proposta di risoluzione è stata generata ed inserita all'interno del

campo adibito, del *ticket* creato
Estensioni: **UC1.3.1**: I *ticket* completati non sono inerenti al *ticket* creato e non è possibile generare una proposta di risoluzione

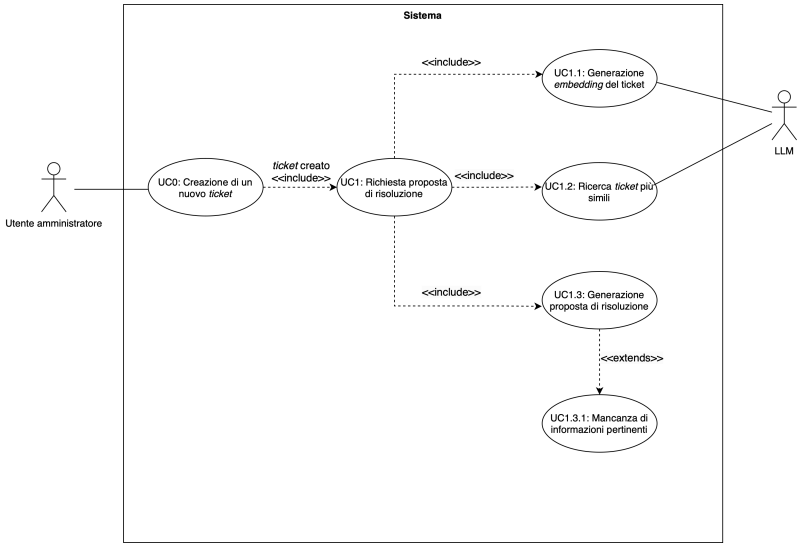

Figura 3.4: Diagramma del caso d'uso UC1 e dei relativi sottocasi

UC2: Chiusura di un *ticket*

Attori principali: Utente amministratore
Precondizioni: Il *ticket* è stato completato
Descrizione: L'utente amministratore chiude il *ticket* mettondolo in *Done*
Postcondizioni: Il *ticket* è stato chiuso

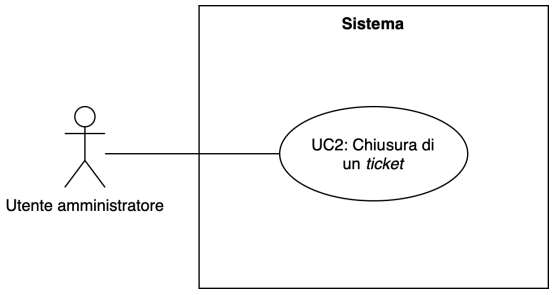

Figura 3.5: Diagramma del caso d'uso UC2

UC3: Generazione embedding del *ticket* completato

Attori principali: LLM
Precondizioni: Il *ticket* è stato chiuso
Descrizione: Viene richiesto all'LLM di generare l'embedding del *ticket* completato per poterlo poi salvare nel *database* MongoDB

Postcondizioni: L'embedding del *ticket* è stato generato

UC3.1: Salvataggio *ticket* su *database* MongoDB

Attori principali: Sistema
Precondizioni: Il *ticket* è stato compleato e in stato *Done*
Descrizione: Il *ticket* viene salvato su *database* MongoDB con il suo relativo *embedding*
Postcondizioni: Il *ticket* è stato salvato su *database* MongoDB con il suo relativo *embedding*

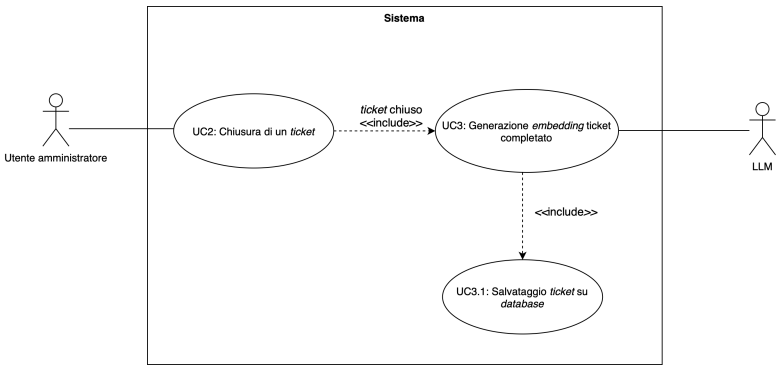

Figura 3.6: Diagramma del caso d'uso UC3 e del relativo sottocaso

Chatbot

UC0: Autenticazione nel *chatbot*

Attori principali: Utente generico
Precondizioni: L'utente vuole accedere al *chatbot*
Descrizione: All'accesso al *chatbot*, viene aperta una schermata di *login* per l'autenticazione dell'utente
Postcondizioni: L'utente è autenticato nel *chatbot*

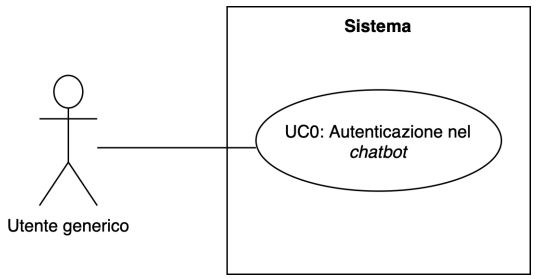

Figura 3.7: Diagramma del caso d'uso UC0 - *Chatbot*

UC1: Autenticazione per il *database* MongoDB

Attori principali: Utente amministratore
Precondizioni: L'utente amministratore è autenticato nel *chatbot*
Descrizione: L'utente amministratore si autentica per poter accedere al *database* MongoDB e poter ricercare i *ticket* più simili
Postcondizioni: L'utente amministratore è autenticato per il *database* MongoDB

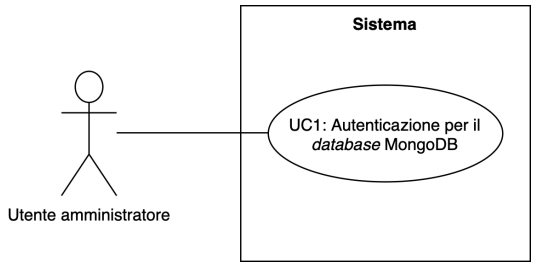

Figura 3.8: Diagramma del caso d'uso UC1 - *Chatbot*

UC2: Scelta del modello LLM

Attori principali: Utente amministratore
Precondizioni: L'utente amministratore è autenticato nel *chatbot*
Descrizione: L'utente amministratore sceglie il modello LLM con cui interrogare il *chatbot*
Postcondizioni: Il modello LLM è stato scelto

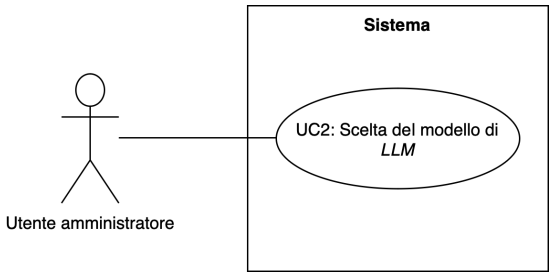

Figura 3.9: Diagramma del caso d'uso UC2 - *Chatbot*

UC3: Inserimento del testo di un *ticket* all'interno della *chat* del *chatbot*

Attori principali: Utente amministratore
Precondizioni: L'utente amministratore ha scelto il modello LLM con cui interrogare il *chatbot*
Descrizione: L'utente amministratore interroga il *chatbot* sui *ticket* Jira
Postcondizioni: Il *chatbot* richiede la richiesta di proposta di risoluzione al sistema

UC3.1: Generazione embedding del testo inserito dall'utente

Attori principali: LLM
Precondizioni: L'utente amministratore ha inserito il testo del *ticket* per interrogare il *chatbot*
Descrizione: Viene generato l'embedding del testo inserito dall'utente per effettuare la ricerca dei *ticket* completati più simili

U3.2: Ricerca *ticket* più simili all'interno del *database* MongoDB

Attori principali: Sistema
Precondizioni: È stato generato l'embedding del testo inserito dall'utente
Descrizione: Il sistema effettua una ricerca vettoriale all'interno del *database* con l'utilizzo dell'embedding, per trovare i *ticket* completati più simili al testo inserito dall'utente
Postcondizioni: I *ticket* completati più simili vengono restuiti dal sistema

UC3.3: Generazione proposta di risoluzione

Attori principali: LLM
Precondizioni: Sono stati restituiti i *ticket* completati più simili al testo inserito dall'utente
Descrizione: Viene richiesto all'LLM di generare una proposta di risoluzione per il testo inserito dall'utente usando come contesto i *ticket* completati più simili
Postcondizioni: La proposta di risoluzione è stata generata e restituita all'utente, nell'interfaccia del *chatbot*

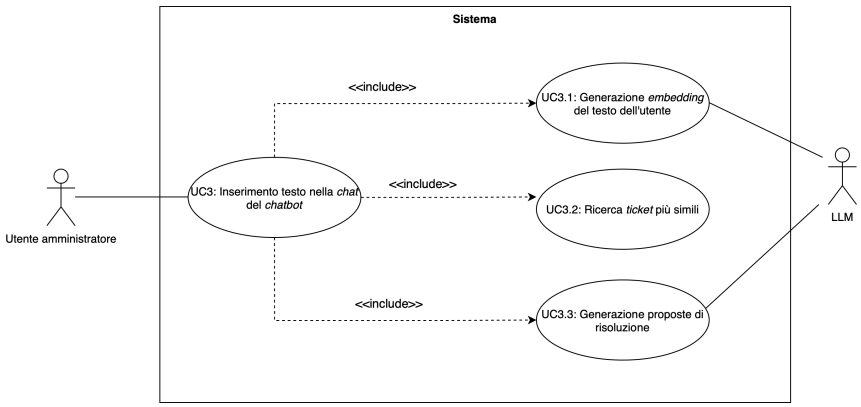

Figura 3.10: Diagramma del caso d'uso UC3 e dei relativi sottocasi - *Chatbot*

3.3.3 Tracciamento dei requisiti

Ho tracciato i requisiti individuati dai casi d'uso con la seguente notazione:

R-[Tipologia][Importanza]-[Numero]

dove:

- **R**: abbreviazione di Requisito
- **Tipologia**: Natura del requisito:
  - **F**: Funzionale
  - **V**: Vincolo
- **Importanza**: Valore alfabetico che indica l'importanza del requisito:
  - **O**: Obbligatorio
  - **D**: Desiderabile
  - **O**: Opzionale
- **Numero**: Numero intero positivo progressivo del requisito

Codice	Descrizione	Importanza	Fonte
R-FO-1	Il sistema deve eseguire una richiesta di generazione di proposta di risoluzione quando un *ticket* viene creato	Obbligatorio	UC1 (Jira)
R-FO-2	Il sistema deve poter interrogare un LLM per la creazione dell'embedding del *ticket* creato	Obbligatorio	UC1.1 (Jira)
R-FO-3	Il sistema deve effettuare una ricerca dei ticket completati più simili al *ticket* creato, nel *database*	Obbligatorio	UC1.2 (Jira)
R-FO-4	Il sistema deve poter interrogare un LLM per generare una proposta di risoluzione utilizzando come contesto i *ticket* completati più simili.	Obbligatorio	UC1.3 (Jira)
R-FO-5	Il sistema deve poter interrogare un LLM per generare una risposta standard in caso di mancanza di *ticket* completati simili.	Obbligatorio	UC1.4 (Jira)
R-FO-6	Il sistema deve poter interrogare un LLM per la generazione dell'embedding del *ticket* completato per salvarlo nel *database*	Obbligatorio	UC3 (Jira)
R-FO-7	Il sistema deve permettere il salvataggio del *ticket* completato e del relativo embedding nel *database*.	Obbligatorio	UC3.1 (Jira)
R-FO-8	Il sistema deve permettere l'autenticazione dell'utente nel *chatbot*.	Obbligatorio	UC0 (Chatbot)

Codice	Descrizione	Importanza	Fonte
R-FO-9	Il sistema deve permettere l'autenticazione dell'utente amministratore per accedere al *database*.	Obbligatorio	UC1 (Chatbot)
R-FO-10	Il sistema deve permettere la scelta del modello LLM con cui interrogare il *chatbot*.	Obbligatorio	UC2 (Chatbot)
R-FO-11	Il sistema deve permettere l'interrogazione al modello LLM tramite una *chat*.	Obbligatorio	UC3 (Chatbot)
R-FO-12	Il sistema deve poter interrogare un LLM per generare l'embedding del testo inserito dall'utente.	Obbligatorio	UC3.1 (Chatbot)
R-FO-13	Il sistema deve permettere la ricerca dei ticket più simili all'interno del *database*.	Obbligatorio	UC3.2 (Chatbot)
R-FO-14	Il sistema deve poter interrogare un LLM per la generazione di una proposta di risoluzione basata sui *ticket* simili.	Obbligatorio	UC3.3 (Chatbot)
R-VO-15	Il sistema Jira deve essere sviluppato utilizzando il Serverless *framework*	Obbligatorio	Interna
R-VO-16	Il *database* utilizzato deve essere MongoDB	Obbligatorio	Interna
R-VO-17	Il *chatbot* deve essere sviluppato utilizzando il *framework* Streamlit	Obbligatorio	Interna
R-VO-18	L'autenticazione nel *chatbot* deve utilizzare il servizio AWS Cognito	Obbligatorio	Interna

Tabella 3.3: Tracciamento dei requisiti

3.4 Progettazione

3.4.1 RAG e selezione del modello LLM

L'integrazione e l'utilizzo di LLM e RAG rappresenta il fulcro dei progetti che ho sviluppato durante lo *stage*. Di seguito descrivo il funzionamento della RAG e mostro che modello ho selezionato per la generazione di testo. Mostrerò anche il *benchmark* creato con il quale ho selezionato il modello migliore per la generazione di testo.

Funzionamento della RAG

Il funzionamento della RAG può essere diviso in più fasi, qui di seguito elencate:

- **Reperimento di dati di addestramento**: per prima cosa è necessario reperire dei dati da fonti esterne o produrli autonomamente in modo da arrichire la

conoscenza del modello. Nel progetto di *stage* i dati sono stati prodotti autonomamente, risultato delle attività svolte durante la 2° settimana come mostrato nella tabella 3.1;

- **Salvataggio dei dati in un *database* vettoriale**: i dati prodotti vengono salvati all'interno di un *database* vettoriale con il loro embedding associato. Questo processo facilita il recupero delle informazioni rilevanti per la generazione di testo. Nel progetti di *stage* i dati sono stati salvati all'interno di un *database* MongoDB;
- **Recupero delle informazioni pertinenti**: quando l'utente crea una *query*, il sistema utilizza il *database* vettoriale per recuperare le informazioni e i dati più pertinenti alla richiesta dell'utente. La ricerca coinvolge l'uso di tecniche di K-Nearest Neighbors (KNN) per ricercare i vettori più simili al vettore della *query*. Tramite questa ricerca si ottengono i dati più pertinenti alla richiesta dell'utente. Nel progetto di *stage* il *database* MongoDB permetteva la creazione di un indice vettoriale adibito alla ricerca vettoriale. Nella creazione dell'indice si poteva specificare il numero di dimensione del vettore, e la funzione di similarità combinata con il KNN utilizzata per la ricerca;

```
{
  "fields": [
    {
      "numDimensions": 1024,
      "path": "embedding",
      "similarity": "cosine",
      "type": "vector"
    }
  ]
}
```

Figura 3.11: Indice vettoriale creato su MongoDB

- **Generazione della risposta**: le informazioni più pertinenti vengono combinate con la *query* data inizialmente dall'utente per creare un *prompt* che verrà inviato al LLM che generà la risposta utilizzando il contesto fornito.

Prompt engineering

Come aspetto cruciale per la generazione di proposte di risoluzione di qualità, ho dovuto creare un *prompt* che permettesse al modello di generare proposte di risoluzioni corrette utilizzando il contesto fornito. Inizialmente per testare la qualità del *prompt* creato, ho utilizzato il modello *MistralLarge* di AWS Bedrock tramite la *console*. La scelta è ricaduta su questo modello in quanto si è dimostrato il più performante tra quelli offerti. La creazione del *prompt* è stata effettuata ispirandosi alle pratiche del Prompt Engineering, ovvero un processo di progettazione e ottimizzazione di *prompt*. L'approccio che ho utilizzato è stato fondamentale un *trial and error* ed è consistito nel testare gli svariati *prompt* creati con il modello *MistralLarge* per valutare la qualità delle proposte di risoluzione generate. In figura 3.12 mostro il prompt finale creato per il sistema di proposte di risoluzione Jira:

```
Ti fornisco le seguenti informazioni estratte dai documenti:
${JSON.stringify(results)}.
La domanda posta è: "${queryText}".

Istruzioni per la risposta:
1. Rispondi ESCLUSIVAMENTE in base alle informazioni contenute nei
   documenti forniti.
2. NON aggiungere interpretazioni, opinioni o ragionamenti personali.
3. Se i documenti trattano problemi simili, fornisci TUTTE le possibili
   soluzioni basate sulle risoluzioni presenti.
4. Descrivi in modo chiaro ed esaustivo OGNI soluzione possibile,
   senza limitarti a una sola opzione.
5. Se le informazioni necessarie non sono presenti nei documenti e i
   problemi non sono simili, dichiara ESPLICITAMENTE l'impossibilità di
   rispondere.
6. La risposta deve essere in lingua italiana.
7. Inizia SEMPRE la risposta con "In base ai precedenti ticket:".
8. Includi link nel formato:
   https://${process.env.JIRA_DOMAIN}/jira/servicedesk/projects/
   ${process.env.JIRA_PROJECT_KEY}/queues/custom/66/${key}
9. Utilizza il seguente formato per ogni soluzione, se disponibile:
    -[Chiave del ticket][Descrizione dettagliata della risoluzione][Link]

10. Se non è possibile fornire una risposta, specificalo
    chiaramente SENZA aggiungere ulteriori informazioni o speculazioni.

Attieniti RIGOROSAMENTE a queste istruzioni senza deviazioni.
```

Figura 3.12: *Prompt* per il sistema di proposte di risoluzione Jira

Per quanto riguarda la creazione del *prompt* per il *chatbot*, ho seguito lo stesso approccio, ma con delle modifiche per adattarlo al contesto del *chatbot*. In figura 3.13 mostro il prompt finale creato per il *chatbot*:

```
Sei un assistente specializzato che fornisce informazioni e soluzioni
basandosi esclusivamente sul database e sui documenti di supporto
disponibili.  Il tuo compito è analizzare le domande degli
utenti e rispondere in modo appropriato seguendo
queste linee guida:

1. Formato della domanda:
- Verifica che la domanda dell'utente sia nel formato corretto:
[Domanda, codice della componente]
- Se manca il codice alfanumerico della componente, rispondi solo che
 non puoi procedere senza queste informazioni e richiedi i dettagli
 nel formato corretto
- Non fornire soluzioni o informazioni dai documenti se il formato
non è corretto

2. Analisi dei documenti:
- Se il formato è corretto, cerca nei documenti componenti uguali o
simili,
 anche se non trattano esattamente lo stesso problema
- Verifica attentamente la pertinenza delle informazioni trovate
```

```
3. Formulazione della risposta:
- Basa la tua risposta rigorosamente sulle informazioni presenti nei
documenti, senza aggiungere interpretazioni o ragionamenti personali
- La risposta deve essere in italiano, esaustiva e descrittiva
- Inizia sempre con "In base ai precedenti ticket:"
- Se pertinente, includi:
    • Riassunto della descrizione del ticket
    • Codice della componente
    • Soluzione proposta
    • Chiave del ticket
- Termina la risposta dopo aver suggerito una soluzione o dichiarato
l'impossibilità di rispondere

4. In caso di mancanza di informazioni:
- Se non trovi informazioni pertinenti o componenti simili, dichiara
chiaramente che non puoi fornire una risposta alla domanda

Context: {context}
User question: {input_text}
Ricorda: analizza attentamente il formato della domanda prima di
procedere con la ricerca nei documenti e la formulazione della
risposta.
```

Figura 3.13: *Prompt* per il *chatbot*

Selezione del modello LLM

La selezione del modello rappresenta una scelta cruciale per una proposta di risoluzione coerente e di qualità. Come ho mostrato nella tabella 3.1 ho creato un *benchmark* per selezionare quale tra i modelli più performanti offerti da AWS Bedrock fosse il migliore per il contesto dei due progetti. Per la creazione del *benchmark* ho creato una serie di *ticket* fittizzi a cui ho associato delle proposte di risoluzione in base ai *ticket* completati contenuti nel *database*. In seguito, per ognuno di questi ticket veniva richiesta una proposta di risoluzione ai vari modelli da testare e le risposte venivano valutato da un ulteriore modello, di cui ho avuto modo di poterne testare l'accuratezza tramite la *console* di AWS Bedrock.

```
const tickets : Ticket[] = [
  {
    //Domanda 1 – Richiesta non del tutto presente, ma si controlla la possibilità di suggerire soluzioni per problemi
    //di surriscaldamento del laptop

    //Expected : Non trovo informazioni precise ma suggerisce di controllare la scheda video(ci sono ticket che parlano
    //di problemi con la scheda video) e di pulire il laptop dai residui di polvere
    summary: 'Laptop si surriscalda',
    description: 'Il mio laptop mentre lo utilizzo si surriscalda in modo anomalo',
    components: 'L-07',
    comments: ['Laptop preso in assitenza'],
    resolution_description: 'Unresolved',
    status: 'Open',
    resolution: 'Unresolved',
  },
  {
    //Domanda 2 – Richiesta presente, con una descrizione simile ma non del tutto uguale a quello presente nel db

    //Expected: Mmeoria video sostituita
    summary: 'Scheda video con artefatti grafici',
    description: 'Nello schermo sono presenti linee colorate sull\'immagine, rendendo difficile l\'utilizzo del laptop per attività grafiche',
    components: 'V-24',
    comments: ['Scheda video presa in assitenza'],
    resolution_description: 'Unresolved',
    status: 'Open',
    resolution: 'Unresolved',
  },
```

Figura 3.14: Esempio *ticket* fittizzi creati per il *benchmark*

Come mostro nell'immagine 3.14, i *ticket* creati possiedono tutti i dati che il sistema riceve quando un *ticket* viene creato su Jira. Alcuni di questi dati sono lasciati con valori *default* in quanto lo scopo era quello di valutare la qualità delle proposte di risoluzione dei vari modelli. Come sistema di *scoring*, come ho accennato in precedenza, ho utilizzato il modello *MistralLarge* di AWS Bedrock. Tramite l'utilizzo di questo modello ho potuto attribuire un punteggio di similarità tra la proposta di risoluzione generata dal modello e la proposta di risoluzione attesa in modo automatico. La scala di valutazione del punteggio di similarità varia tra 0 (risposte totalmente diverse) e 1 (risposte identiche).
In figura 3.15 mostro l'architettura del *benchmark*:

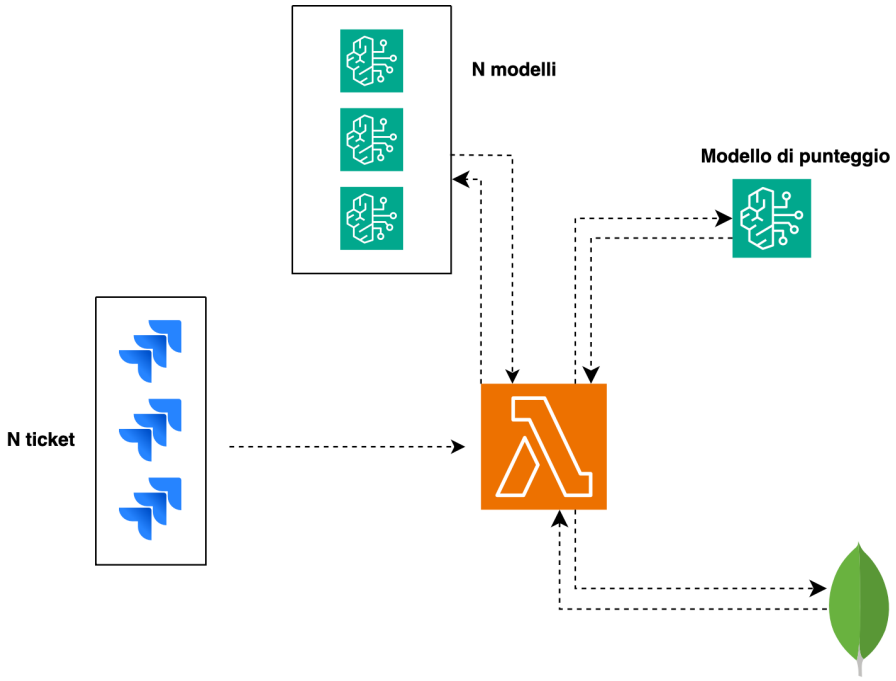

Figura 3.15: Architettura del *benchmark*

Come mostro nell'immagine 3.15, il *benchmark* è composto da quattro componenti principali:

- **N *ticket* fittizzi**: insieme di *ticket* creati per valutare la qualità delle proposte di risoluzione dei vari modelli, come mostrato nell'immagine 3.14;
- **N Modelli**: insieme di LLM offerti da AWS Bedrock che interrogo per generare proposte di risoluzione per gli N *ticket* fittizzi e li restituisce alla funzione *lambda*;
- ***Database***: componente che tramite la ricerca vettoriale trova i *ticket* completati più simili ai *ticket* fittizzi creati.
- **Modello di punteggio**: modello che valuta la qualità delle proposte di risoluzione generate dai vari modelli interrogati. Restituisce un numero intero positivo compreso tra 0 (risposta totalmente diversa) e 1 (risposta identica).

In figura 3.16 mostro il risultato del *benchmark*:

		0,57	0,65	0,72
		Punteggio LLM: MistralLarge		
Modello	**Tipo**	**Low**	**Medium**	**Strict**
ClaudeAI	AWS	21~23	21~21	18~19
MistralLarge	AWS	25	22~24	21~24
Titan Text G1 Premium	AWS	20~23	18~19	17~18
llama3:70B Instruct	AWS	24~25	23~24	22~23
Claude 3.5 Sonnet	AWS	24~25	22~23	22~23

Figura 3.16: Risultato del *benchmark*

Il punteggio relativo ad ogni modello è suddiviso in tre categorie:

- **Low**: punteggio di similarita maggiore di 0,57 incluso;
- **Medium**: punteggio di similarità maggiore di 0,65 incluso;
- **Strict**: punteggio di similarità maggiore di 0,72 incluso;

Il punteggio attribuito ai vari modelli però è un indicatore di qualità rispetto alle proposte di risoluzione attese da me create. Questo implica che modelli che hanno ottenuto un punteggio più basso generano comunque proposte di risoluzione corrette, ma non nel formato da me richiesto. Questo è un aspetto che ho dovuto considerare nella scelta del modello. Come si può notare dall'immagine 3.16, il modello *Claude3.5 Sonnet* e *MistralLarge* hanno ottenuto i migliori risultati. Tuttavia la scelta del modello è ricaduta su Claude3.5Sonnet in quanto ha dimostrato di generare proposte di risoluzione di qualità superiore rispetto a *MistralLarge*.
Come modello per la generazione di embedding invece la scelta è ricaduta su *Titan Text Embeddings V2* in quanto rispetto alle alternative offerte da AWS Bedrock, permetteva di generare embedding con una dimensione minore ma con qualità superiore rispetto agli altri modelli.

3.4.2 Architettura del sistema Jira

La progettazione delle componenti del sistema Jira si è basata su un'architettura *event-driven*, tipica del *framework Serverless*. Questo tipo di architettura permette di creare applicazioni scalabili e flessibili, in quanto le funzioni *lambda* create vengono eseguite solo quando gli eventi vengono scatenati. Durante le prime settimane di *stage* mi è stato fornito un *template* di progetto Serverless che ho utilizzato come base per la creazione del sistema Jira. Di seguito elenco le componenti principali del *template* di progetto.

Handlers

Gli *handlers* sono le funzioni *lambda* che vengono eseguite quando un evento viene scatenato. Esse rappresentano il fulcro del sistema Jira in quanto gestiscono la richiesta di proposte di risoluzione per i nuovi *ticket* e il salvataggio automatico dei *ticket* chiusi in Jira nel *database*. Gli *handlers* vengono configurati all'interno del file *index.ts* e vengono associati ad un evento specifico. Di seguito mostro un il file di configurazione delle due funzioni *lambda* del sistema Jira

```
1    export const tickets: AWS['functions'] = {
2      save: {
```

```
3         handler: `${handlerPath(__dirname)}/handlers/save.main`,
4         name: '${self:provider.stage}-${self:service}-save',
5         events: [
6           {
7             http: {
8               method: 'any',
9               path: 'save',
10              cors: true,
11            },
12          },
13        ],
14        timeout: 60,
15        memorySize: 128,
16      },
17      suggest: {
18        handler: `${handlerPath(__dirname)}/handlers/suggest.main`,
19        name: '${self:provider.stage}-${self:service}-suggest',
20        events: [
21          {
22            http: {
23              method: 'any',
24              path: 'suggest',
25              cors: true,
26            },
27          },
28        ],
29        timeout: 60,
30        memorySize: 128,
31      },
32    };
```

Figura 3.17: Configurazione delle funzioni *lambda* del sistema Jira

Come mostro nel frammento di codice 3.17, sono presenti le configurazione di due funzioni *lambda*, una rispettivamente per il salvataggio del *ticket* chiuso (funzione *save*) e una per la richiesta di proposta di risoluzione per il *ticket* appena creato (funzione *suggest*). Entrambe le funzioni sono configurate per essere eseguite quando un evento Hypertext Transfer Protocol (HTTP) viene scatenato. Questo evento viene scatenato da AWS API Gateway che riceve la notifica da Jira quando un *ticket* viene creato o chiuso. Definiscono inoltre il tempo massimo di esecuzione e la quantità di memoria allocata per l'esecuzione.

Services

La componente *services* rappresenta la logica di business. Sono una serie di funzioni di supporto che vengono utilizzate dalle funzioni *lambda* per eseguire operazioni specifiche. Queste funzioni vengono utilizzate per eseguire operazione ad esempio di connessione al *database* o per l'interfacciamento con altri servizi quali ad esempio AWS Bedrock. Includono funzioni di *parsing* per la gestione delle risposte. Questo approccio è vantaggioso in quanto:

- **Modularità**: facilità la manutenzione e la gestione del codice. Ogni funzione ha un compito specifico e ben definito;

- **Riusabilità**: le funzioni di supporto possono essere riutilizzate in altre funzioni *lambda* che si vuole creare;
- **Pulizia del codice**: il codice delle funzioni *lambda* risulta più pulito e leggibile in quanto le operazioni complesse vengono gestite da funzioni di supporto.

Per migliorare ulteriormente la struttura e la manutenibilità del codice, ho adotattato diversi *design patterns*. Uno di questo è il Data Transfer Object (DTO), che viene utilizzato per trasferire i dati tra il *client* e il *server* e viceversa. Grazie a questo *design pattern* è possibile selezionare solo i campi necessari che si vogliono trasferire, alleggerendo così il *payload* della chiamata API.

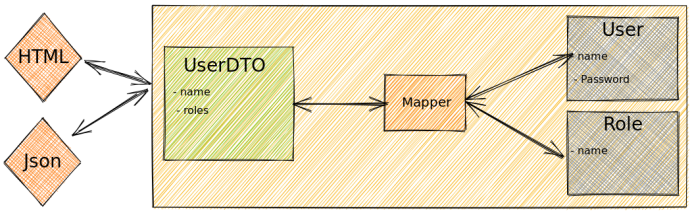

Figura 3.18: Esempio di utilizzo del *design pattern* DTO
Fonte: www.baeldung.com

Un altro *desig pattern* che ho utilizzato è il *Mapper pattern*, che è strettamente collegato al DTO *pattern*. Come mostro nell'immagine 3.18, il *mapper* consiste in uno o più metodi per facilitare la trasformazione dei DTO in oggetti utilizzati dalla logica di *business*, mappando i diversi campi da un tipo all'altro. All'interno del sistema Jira, il *mapper* è stato utilizzato per trasformare i dati ricevuti dalle Application Program Interface di Jira in oggetti *ticket* che vengono utilizzati dalle funzioni *lambda* per eseguire le operazioni di salvataggio e di richiesta di proposta di risoluzione. Questo *design pattern* contribuisce a mantere il codice organizzato, garantendo che i dati siano nel formato corretto. Ho adottato il *Factory Method pattern* per la creazione delle interfacce di connessione con i vari modelli offerti da AWS Bedrock. Questo *design pattern* permette di astrarre la logica di creazione degli oggetti, permettendo l'aggiunta di nuovi modelli senza la necessità di modificare il codice esistente. La flessibilità e la scalabilità offerta da questo *design pattern* sono state fondamentali per garantire che il sistema possa evolvere nel tempo, permettendo l'integrazione di nuovi modelli che verranno resi disponibili in futuro.

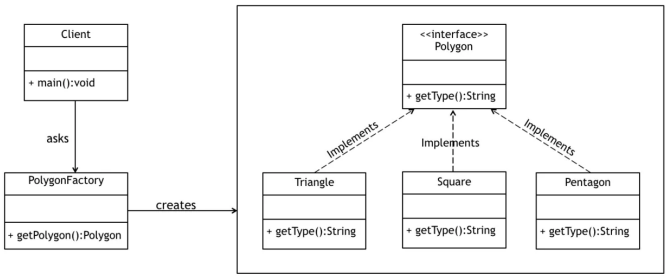

Figura 3.19: Illustrazione del *design pattern* Factory Method
Fonte: www.medium.com

Infine, ho adottato il *design pattern Strategy* per la creazione delle strategie di generazione del *prompt* e di risposta dei vari LLM offerti da AWS Bedrock. Definisco una strategia di generazione *prompt* ad ogni modello in quanto ognuno di essi possiede delle *keywords* e una formattazione del testo specifico. Il *prompt* mostrato nella sezione 3.4.1 è la base per ognuno dei modelli.

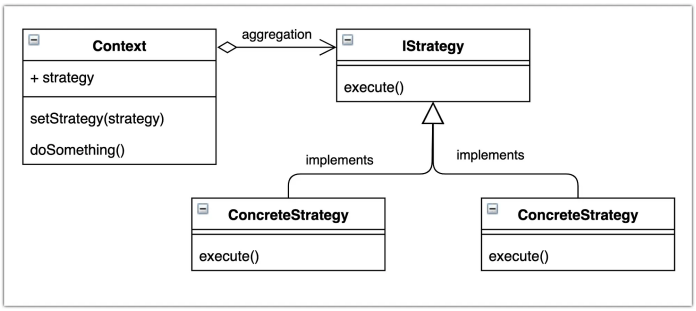

Figura 3.20: Illustrazione del *design pattern* Strategy
Fonte: www.medium.com

In figura 3.21 mostro lo schema dell'architettura del sistema Jira:

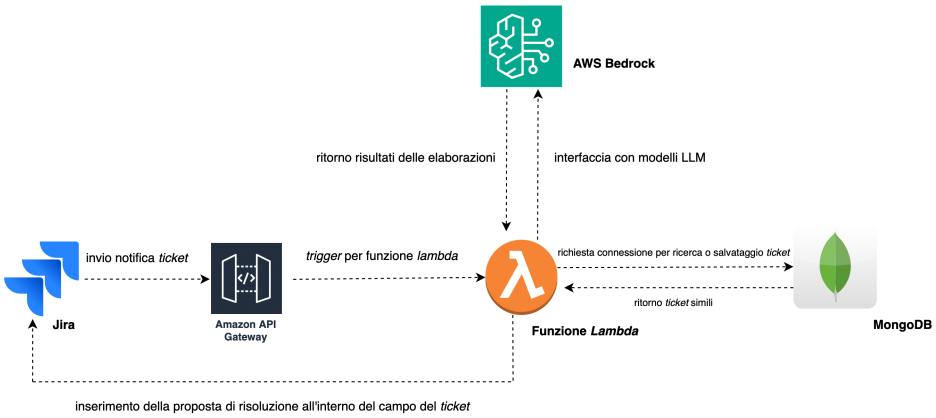

Figura 3.21: Schema dell'architettura del sistema Jira

3.4.3 Architettura del *Chatbot*

La progettazione del *chatbot* si è basata sull'architettura *client-server* del *framework* Streamlit. Il *client* viene rappresentato dal *browser* che l'utente utilizza per interagire con il *chatbot*, mentre il *server* è rappresentato dall'esecuzione del codice Python che in risposta alle azioni dell'utente, invia i risultati al *client* sotto forma di Hypertext Markup Language (HTML)/Cascading Style Sheets (CSS)/JavaScript che vengono visualizzati nel *browser*.

Back-end

Essendo l'architettura del *chatbot* basata sul *framework* Streamlit, il *back-end* è rappresentato dal codice Python che viene eseguito in risposta alle azioni dell'utente.

Una componente chiave del *back-end* è l'utilizzo del *framework* Langchain. Esso viene impiegato in diverse parti cruciali del *back-end*, ovvero:

- **Integrazione con il *database* vettoriale**: con l'utilizzo di Langchain, è possibile stabilire una connessione con il *database* vettoriale MongoDB per effettuare la ricerca vettoriale dei *ticket* completati più simili al testo inserito dall'utente;
- **Gestione degli embedding**: tramite LangChain viene facilitato l'uso di diversi modelli di embedding messi a disposizione da AWS.
- **Orchestrazione dei modelli**: come da requisito da soddisfare presentato nella tabella 3.3, il *chatbot* deve poter dare la possibilità all'utente di selezionare l' LLM con cui interrogare il *chatbot*. LangChain permette di integrare ed orchestrare i diversi modelli consentendo una flessibilità nella scelta e nell'utilizzo dei modelli scelti dall'utente.

Anche in questo caso ho adotatto il *Factory Method pattern* per la creazione delle interfacce di connessione con i vari modelli offerti da AWS Bedrock e il *design pattern Strategy* per la creazione delle strategie di generazione del *prompt*.

Front-end

Allo stesso modo, il *front-end* del *chatbot* è anch'esso rappresentato da codice Python. Tramite la libreria Streamlit, è possibile creare un'interfaccia grafica interattive per l'utente. Possiede una serie di componenti per la creazione di *widget*. Grazie all' utilizzo *framework*, l'interfaccia viene aggiornata automaticamente in risposta alle azioni dell'utente.

In figura 3.22 mostro lo schema dell'architettura del *chatbot*:

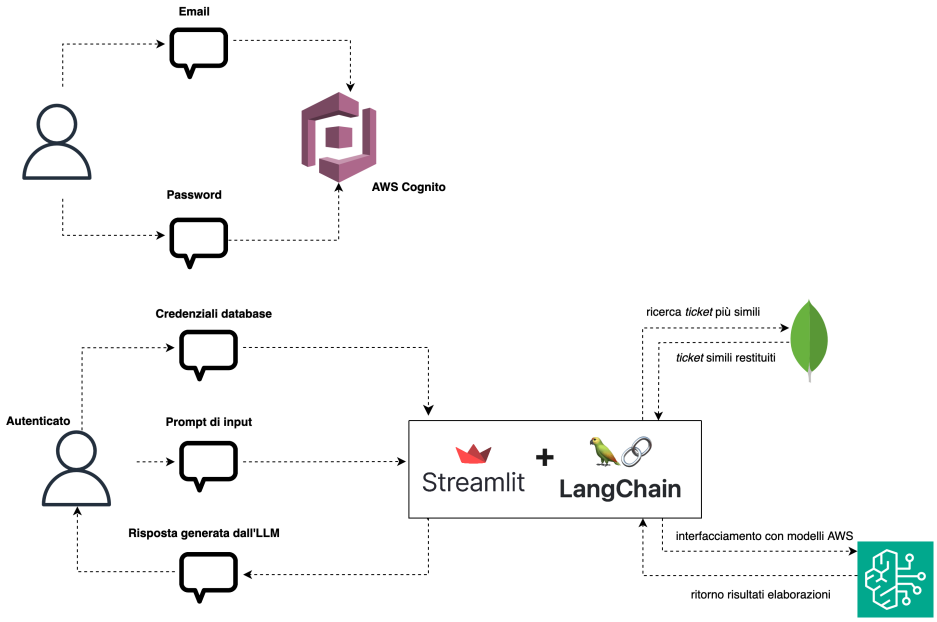

Figura 3.22: Schema dell'architettura del *chatbot*

3.5 Codifica

3.5.1 Sistema Jira

Al fine di sviluppare un sistema con le funzionalità definite durante il processo di progettazione, ho sviluppato due *handlers* (ovvero due funzioni *lambda*), che descrivo qui di seguito:

- ***suggest***: funzione *lambda* che viene eseguita quando un evento HTTP viene scatenato da AWS API Gateway. Questo evento si verifica quando un *ticket* viene creato, attraverso i *webhook* di Jira. La funzione *suggest* si occupa di interrogare il *database* vettoriale per la ricerca dei *ticket* più simili al *ticket* appena creato e di generare una proposta di risoluzione utilizzando l'LLM impostato. La proposta di risoluzione viene poi inserita all'interno di un campo personalizzato dedicato al *ticket* creato in Jira. All'interno della funzione *suggest*, ho sviluppato delle funzioni *services* che si occupano di effettuare la ricerca vettoriale, di interrogare l'LLM e di generare la proposta di risoluzione. Questo approccio permette di mantenere il codice pulito e organizzato, garantendo che ogni funzione abbia un compito specifico e ben definito.

```
1  async function handler(event: Event<TicketDataEventDto>):
   ↪  Promise<undefined> {
2      if (!isDefined(event.body)) {
3          throw new HTTP400Error('Il body e\' undefined');
4      }
5      const body: TicketDataEventDto = event.body;
6      const ticket: Ticket = await createTicket(body);
7
8      if (isDefined(ticket) && ticket.status === 'Open') {
9          const embeddingService: EmbeddingModel =
           ↪  createEmbeddingModels('AmazonTitanV2');
10         await suggestSolution(ticket, ticket.key, embeddingService);
11     } else {
12         throw new Error('Errore nella pipeline di elaborazione del
           ↪  ticket');
13     }
14 }
15
16 export const main: HandlerResponse = baseHandler<TicketDataEventDto,
   ↪  undefined>(handler, 200);
```

Figura 3.23: Funzione *lambda suggest*

- ***save***: funzione *lambda* che viene eseguita quando un evento HTTP viene scatenato da AWS API Gateway. Questo evento si verifica quando un *ticket* viene chiuso, sempre con l'ausilio dei *webhook* di Jira. La funzione *save* si occupa di salvare il *ticket* chiuso e il suo relativo embedding all'interno del *database*. Questo permette di poter ricercare in futuro i *ticket* completati più simili ai *ticket* creati, arricchendo così la conoscenza del *database*. Anche in questo caso, ho sviluppato delle funzioni *services* che si occupano di inserire il *ticket* chiuso e di calcolare il relativo embedding all'interno del *database*.

```
async function handler(event: Event<TicketDataEventDto>):
↪  Promise<undefined> {
    if (!isDefined(event.body)) {
        throw new HTTP400Error('Il body e\' undefined');
    }
    const body: TicketDataEventDto = event.body;
    const ticket: Ticket = await createTicket(body);

    const embeddingService: EmbeddingModel =
    ↪  createEmbeddingModels('AmazonTitanV2');

    if (isDefined(ticket) && ticket.status === 'Done') {
        await processEmbedding(ticket, embeddingService, false);
        await saveTicket(ticket);
    } else {
        throw new Error('Errore nella pipeline di elaborazione del
        ↪  ticket');
    }
}

export const main: HandlerResponse = baseHandler<TicketDataEventDto,
↪  undefined>(handler, 200);
```

Figura 3.24: Funzione *lambda save*

3.5.2 *Chatbot*

Per lo sviluppo del *chabot* come ho descritto precedentemente ho utilizzato il *framework* Streamlit. Questo *framework* mi ha permesso di sviluppare il *back-end* e il *front-end* del *chatbot* in semplici *file* Python. Qui di seguito descrivo la suddivisione dei *file* Python che compongono il *chatbot*:

- ***Back-end***: per quanto riguarda il *back-end* del *chatbot*, ho creato diversi *file* Python che si occupano di gestire l'*input* dell'utente, di interrogare il *database* vettoriale per la ricerca dei *ticket* più simili e l'interfacciamento con i vari modelli LLM offerti da AWS Bedrock. Ho gestito queste diverse operazioni con l'ausilio del *framework* LangChain. Qui di seguito mostro un esempio di uno dei *file* Python che compongono il *back-end* del *chatbot*.

```

from langchain_community.vectorstores import MongoDBAtlasVectorSearch

def cloudVectorSearch(database_url, database_name, collection_name,
↪  query_embedding):
    namespace = f"{database_name}.{collection_name}"

    vector_search = MongoDBAtlasVectorSearch.from_connection_string(
        database_url,
        namespace,
        embeddings,
        embedding_key="embedding",
        index_name="vector_index",
```

```
13          text_key="raw"
14      )
15      search_result =
        ↪  vector_search.similarity_search_with_relevance_scores(
16          query_embedding,
17          k=3,
18      )
19
20      desired_fields = ['key','summary', 'description',
        ↪  'resolution_description', 'components']
21      filtered_result = filter_fields(search_result, desired_fields)
22
23      return filtered_result
```

Figura 3.25: Funzione di ricerca vettoriale utilizzata nel *chatbot*

Nel frammento di codice 3.25 mostro la funzione di ricerca vettoriale utilizzata nel *chatbot*. Questa funzione si occupa di interrogare il *database* vettoriale MongoDB fornendogli il nome dell'indice, il nome del campo contenente l'embedding e l'embedding del testo della *query* dell'utente. La funzione restituisce i tre *ticket* più simili alla *query* dell'utente filtrandoli per i campi desiderati.

- ***Front-end***: per lo sviluppo del *front-end* del *chatbot* ho creato un unico *file* Python che si occupa di creare l'interfaccia grafica interattiva utilizzando le componenti offerte dal *framework* Streamlit. Questo *file* Python si occupa di gestire l'interfaccia grafica, di inviare le richieste al *back-end* e di visualizzare i risultati all'utente.

```
1  input_text = st.chat_input()
2
3  if input_text:
4      with st.chat_message("user"):
5          st.write(input_text)
6
7      st.session_state.chat_history.append(HumanMessage(input_text))
8
9          search_result =
           ↪  cloudVectorSearch(st.session_state['database_url'],
           ↪  st.session_state['database_name'],
           ↪  st.session_state['collection_name'], input_text)
10     else:
11     st.session_state.search_results.append(search_result)
12
13     with st.chat_message("assistant"):
14         chat_response= st.write_stream(conversation(input_text,
           ↪  st.session_state.chat_history,
15         st.session_state.search_results, model_choice))
16     st.session_state.chat_history.append(AIMessage(chat_response))
```

Figura 3.26: Frammento del codice Python relativo al *front-end*

Nel frammento di codice 3.26 mostro un esempio di come viene gestita la richiesta dell'utente e come vengono visualizzati i relativi risultati. Viene effettuata una ricerca vettoriale che restituisce i *ticket* più simili alla *query* dell'utente. Successivamanete inserisco i risultati all'interno della *st.session_state* che rappresenta la sessione del-

l'utente. Infine viene generata la proposta di risoluzione con l'ausilio del LLM scelto dall'utente con i risultati della ricerca vettoriale e visualizzata all'utente.

3.5.3 *Best practice*

Durante il mio periodo di *stage* ho fatto uso di diversi strumenti utilizzati in ambito aziendali e ho potuto apprendere alcune *best practice* che ho applicato durante l'attività di codifica per lavorare allo stato dell'arte. Di seguito elenco alcune delle *best practice* che ho applicato:

- ***Editor* di codice**: all'inzio dell'attività di codifica mi è stato richiesto di utilizzare la configurazione aziendale del *plugin* ESLint. Come ho descritto in 1.5, ESLint è uno strumento di analisi statica del codice che permette di identificare e segnalare gli errori di sintassi e di stile presenti nel codice. Questo strumento mi ha permesso di scrivere codice uniforme e pulito, seguendo lo *standard* aziendale. Inoltre, ha permesso di ridurre il numero di errori di sintassi e di stile presenti nel codice, velocizzando le *code review* necessarie per l'avanzamento del progetto.

- ***Code reviews***: l'intera attività di codifica è stata sottoposta a *code reviews* da parte del mio *tutor* aziendale e da un altro membro dell'azienda. Anche in questo caso mi sono attenuto al processo già ben consolidato in azienda. Per ogni funzionalità sviluppata, ho creato una *feature branch* all'interno di un *repository* Git dedicato. Il *feature branch* veniva denominato con la seguente notazione:

  feature/nome_funzionalità

  in modo da rendere chiaro il contenuto del *branch*. Una volta che completavo lo sviluppo della funzionalità, aprivo una *pull request* per la *code review*. Questo processo mi ha permesso di ricevere *feedback* costante sul codice che stavo sviluppando, permettondomi di correggere eventuali errori e di migliorare la qualità del codice. Infine, mi ha permesso di approfondire in modo più dettagliato l'utilizzo di Github come strumento di versionamento del codice e di osservare come si sviluppa codice seguendo dei processi consolidati e ben definiti.

- ***Interfaces***: Typescrit richiede la dichiarazione esplicita dei tipi delle variabili e degli oggetti utilizzati all'interno del codice. Per questo motivo, ho definito delle *interfaces* per definire i tipi di tutti gli oggetti utilizzati a cui non potevano essere assegnati i tipi primitivi. A differenza delle classi, le *interfaces* non vengono instanziate, ma presentano solo i tipi degli attributi e la firma dei metodi che un oggetto avrà una volta implementato. Questo mi ha permesso di definire in modo chiaro e preciso i tipi degli oggetti utilizzati all'interno del codice, in modo da non incorrere in errori di tipo durante l'esecuzione del codice.

- **Documentazione**: durante tutta l'attività di codifica ho documentato il codice sviluppato attraverso l'utilizzo di commenti. Durante l'ultima settimana di *stage* ho redatto una documentazione tecnica dettagliata contenente le scelte progettuali fatte per entrambi i progetti, con le funzionalità implementate, e un manuale utente per il corretto utilizzo del sistema Jira e del *chatbot*.

3.6 Verifica e validazione

3.6.1 Sistema Jira

Test tramite eventi

Per verificare il corretto funzionamento delle due funzioni *lambda* che ho sviluppato per il sistema Jira, ho utilizzato il servizio di *testing* tramite eventi offerto da AWS Lambda. Tramite questo servizio ho potuto creare un evento di test che simulava l'evento scatenato da AWS API Gateway quando un *ticket* viene creato o chiuso. Tramite questo approccio ho potuto verificare che le funzioni *lambda* ricevessero correttamente il *ticket* dall'evento. Le operazioni di proposta di risoluzione e di salvataggio del *ticket* sono state testate a parte, in quanto richiedevano l'interrogazione del *database* vettoriale e l'interfacciamento con i vari modelli LLM offerti da AWS Bedrock.

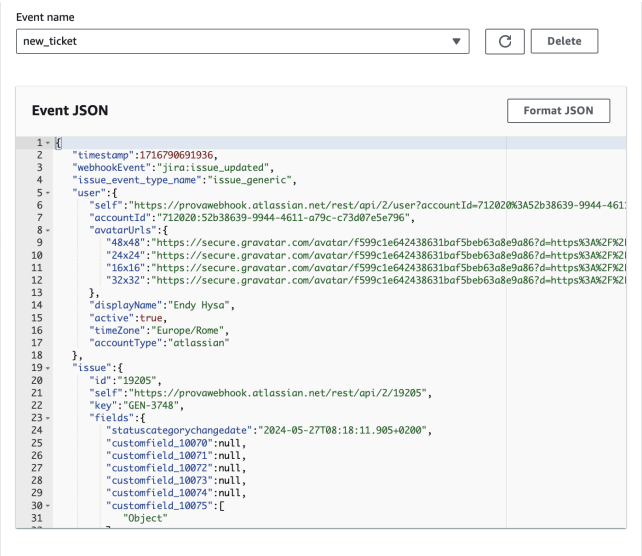

Figura 3.27: Esempio di evento di *test*

Nell'immagine 3.27 mostro un esempio di evento di test creato per verificare il corretto funzionamento della funzione *lambda suggest*. L'evento contiene i dati di un *ticket* fittizio creato per il *testing*. Tramite questo evento ho potuto verificare che il *ticket* contenuto nell'evento venisse correttamente ricevuto dalla funzione *lambda*.

Test di integrazione

I *test* di integrazione si occupano di verificare che le varie componenti del sistema funzionino correttamente, garantendo che le varie parti dell'applicativo interagiscano nel modo corretto. Durante il mio periodo di *stage* ho potuto svolgere i *test* di integrazione per verificare che le due funzioni *lambda* sviluppate per il sistema Jira interagissero correttamente con il *database* vettoriale e con l'LLM impostato per la generazione delle proposte di risoluzione. Questi *test* li ho svolti manualmente. Il *framework* Serverless offre la possibilità di esporre le funzioni *lambda* in locale, chiamabili tramite API con il comando *serverless-offline*. Questo mi ha permesso di simulare l'interazione tra le varie componenti e servizi del sistema Jira, testati attraverso Postman. Come descritto nella

sezione 1.5, Postman è uno strumento di *testing* delle API che permette di effettuare richieste HTTP e di verificare le risposte ricevute.
Con i *test* di integrazione si conclude il processo di verifica e validazione del sistem Jira, che mi ha permesso di verificare il corretto funzionamento del prodotto sviluppato e che soddisfi le aspettative dell'azienda.

3.6.2 *Chatbot*

Per quanto riguarda il *chatbot*, ho svolto solamente i *test* di integrazione. Questi *test* li ho svolti manualmente per verificare che il *chatbot* funzionasse correttamente e che le varie componenti interagissero nel modo corretto. Essendo nata l'idea di sviluppare il *chatbot* durante il mio periodo di *stage*, non è stato possibile dedicare molto tempo alla creazione di *test*, in quanto il suo scopo principale era quello di dimostrare la fattibilità e l'utilità di un *chatbot* per l'interrogazione dei *ticket* di Jira.

3.7 Risultato finale

3.7.1 Il prodotto realizzato

I prodotti che ho realizzato durante il mio periodo di *stage* soddisfano quanto atteso dall'azienda, risultando funzionanti e rispettando i requisiti richiesti.

Sistema Jira

Quando l'utente si sarà autenticato nel *workspace* Jira, potrà creare un *ticket* e il sistema da me sviluppato si occuperà di generare una proposta di risoluzione per il *ticket* creato

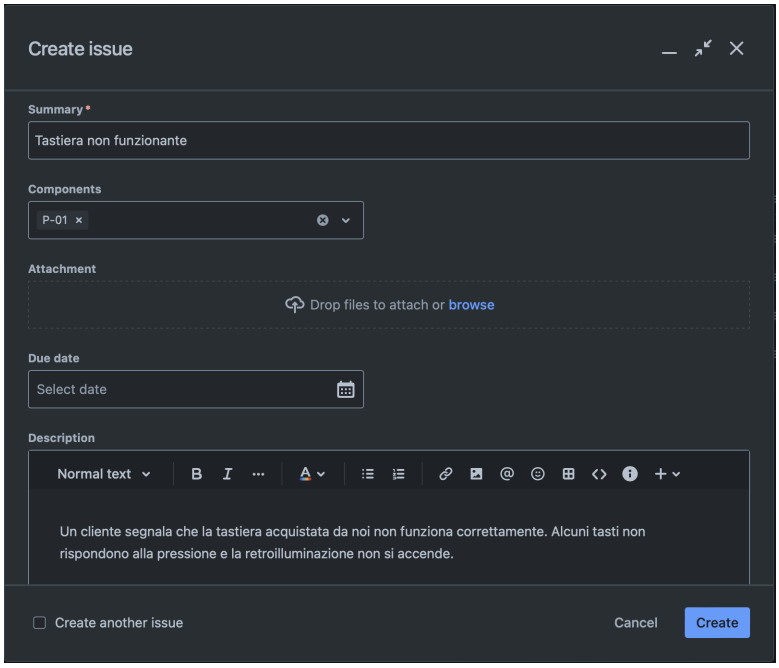

Figura 3.28: Creazione di un nuovo *ticket* su Jira

Come mostro nell'immagine 3.28, l'utente premendo il pulsante *Create*, creerà un nuovo *ticket* su Jira con i relativi campi compilati. Una volta creato il *ticket*, il sistema Jira comincerà a generare una proposta di risoluzione per il *ticket* creato.

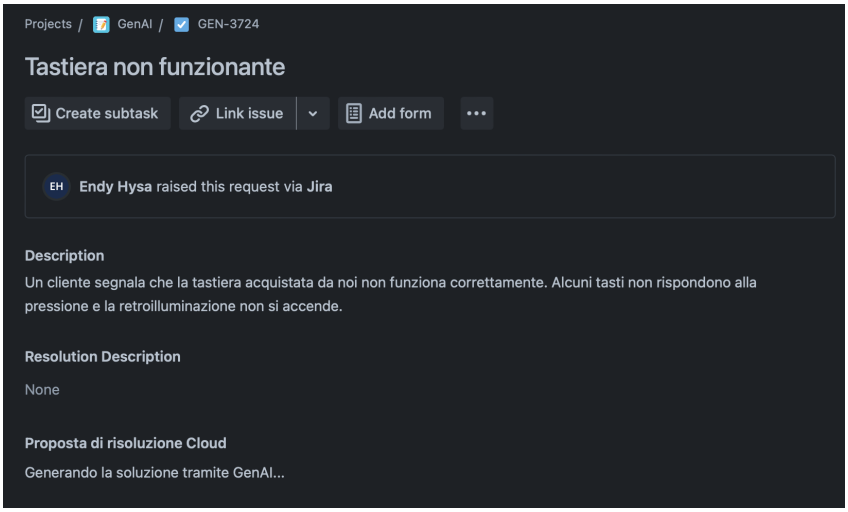

Figura 3.29: Messaggio che indica che il sistma sta generando la proposta di risoluzione

Nell'immagine 3.29 mostro il messaggio che indica che il sistema sta generando la proposta di risoluzione per il *ticket* creato. Una volta che la proposta di risoluzione è stata generata, sovrascriverà il messaggio di generazione.

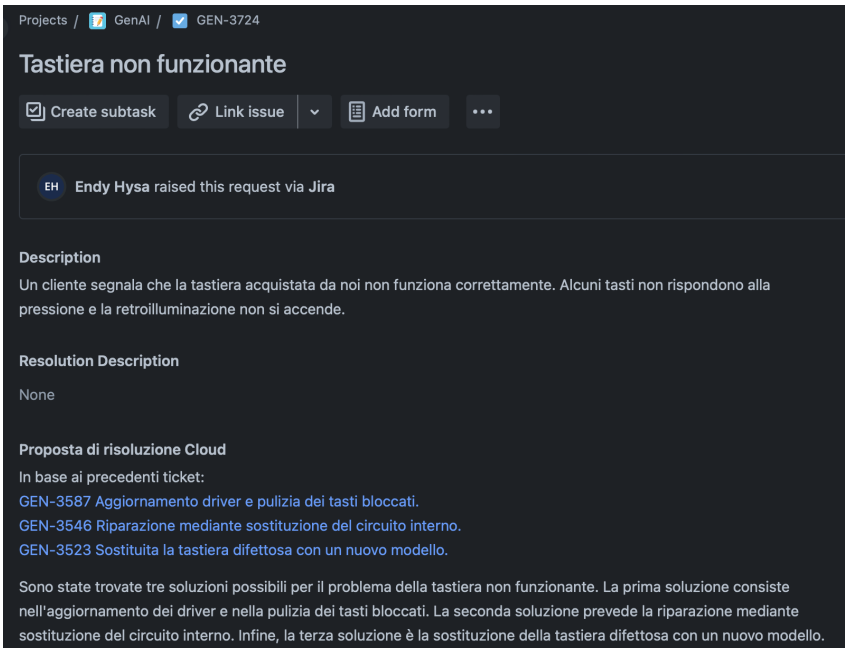

Figura 3.30: Proposta di risoluzione generata dal sistema ed inserita nel *ticket* Jira

La proposta di risoluzione che mostro nell'immagine 3.30, che viene generata è inerente al *ticket* creato e ai *ticket* completati più simili. Nella proposta di risoluzione vengono inseriti anche i *link* ai *ticket* completati più simili, in modo da poterli consultare per avere ulteriori informazioni sulla risoluzione del problema. Nel caso in cui l'utente crei un *ticket* e le informazioni contenute non siano inerenti al problema, la proposta di risoluzione generata dal sistema Jira dichiarerà che non è possibile generare una proposta di risoluzione in quanto non sono presenti informazioni sufficienti.

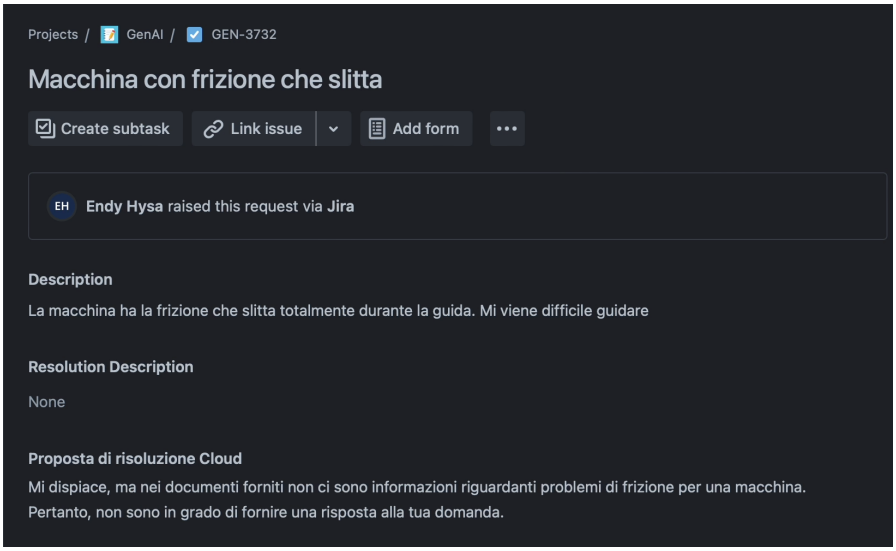

Figura 3.31: Messaggio che indica che non è possibile generare una proposta di risoluzione

Nell'immagine 3.31 mostro il messaggio che indica che non è possibile generare una proposta di risoluzione per il *ticket* aperto. Questo viene mostrato in quanto il *database* non contiene *ticket* completati simili a quello aperto e pertanto non è possibile generare una proposta di risoluzione.
Mettendo il *ticket* in stato *Done*, il sistema Jira chiederà di inserire una risoluzione per il *ticket* chiuso. Una volta inserita la risoluzione, il sistema Jira si occuperà di salvare il *ticket* chiuso e il suo relativo embedding all'interno del *database* vettoriale. Questo viene fatto per arricchire la conoscenza del *database* e permettere di poter ricercare i *ticket* completati più simili ai *ticket* creati in futuro.

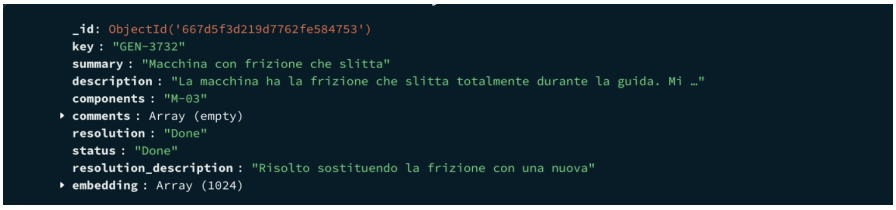

Figura 3.32: Schermata raffigurante il *ticket* chiuso in MongoDB

Chatbot

Per poter interagire con il *chatbot* che ho sviluppato durante il mio periodo di *stage*, l'utente dovrà accedere tramite credenziali al *chatbot* stesso. Come descritto nei requisiti di vincolo descritti nella tabella 3.3, l'autenticazione avviene tramite AWS Cognito.

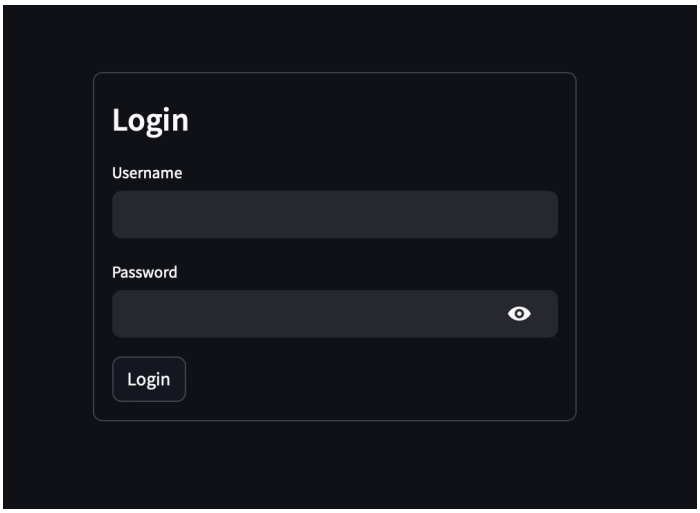

Figura 3.33: Schermata di login del *chatbot*

Una volta che l'utente si sarà autenticato, dovrà inserire le credenziali di accesso al *database* vettoriale MongoDB. Questo passaggio è necessario per poter effettuare la ricerca vettoriale dei *ticket* completati più simili a quello inserito dall'utente.

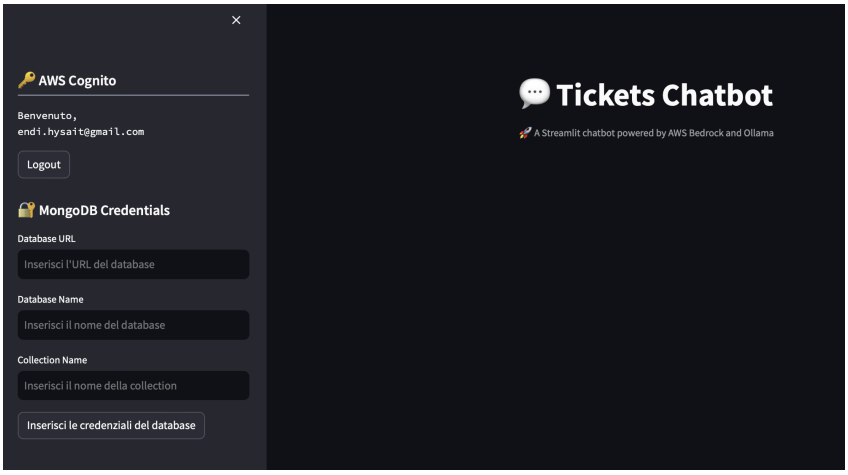

Figura 3.34: Schermata di *login* del *chatbot*

Inserite dall'utente le credenziali necessarie, il *chatbot* è pronto per essere utilizzato. L'utente potrà inserire il testo del *ticket* di cui vuole avere una proposta di risoluzione e selezionare l'LLM con cui interrogare il *chatbot*. L'utente dovrà inserire oltre alla

problematica del *ticket*, anche la sua relativa componente. Questo è necessario per poter effettuare una ricerca vettoriale più precisa e ottenere una proposta di risoluzione di qualità.

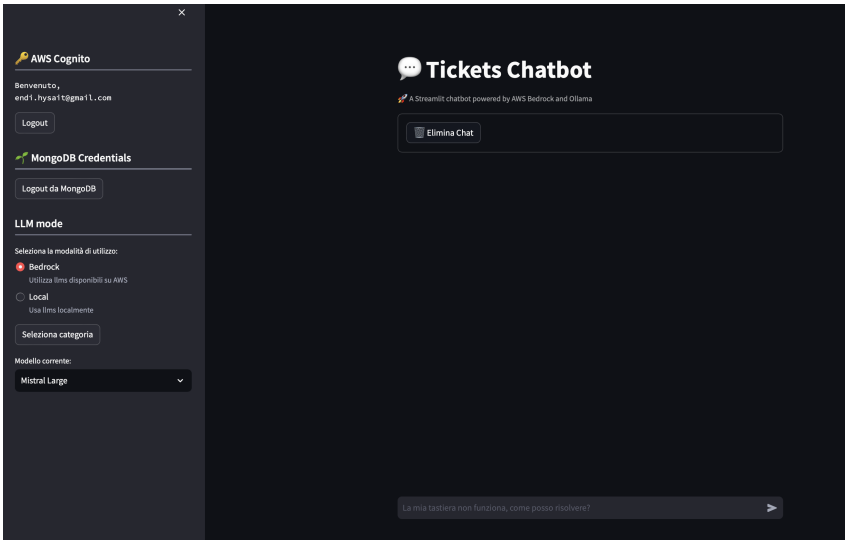

Figura 3.35: Intefaccia del *chatbot*

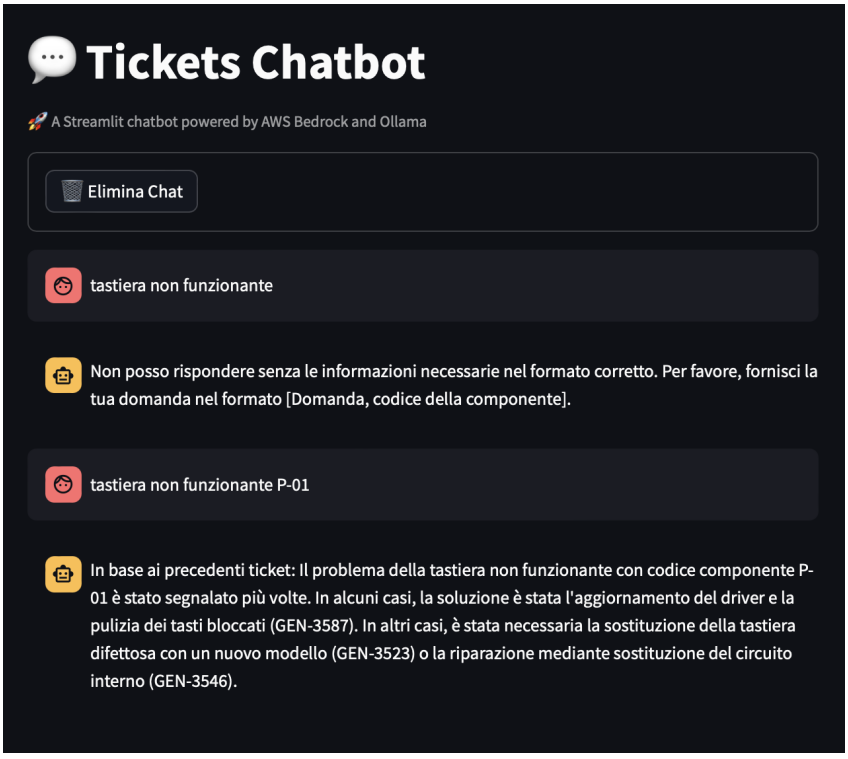

Figura 3.36: Proposta di risoluzione generata dal *chatbot*

3.7.2 Copertura dei requisiti

Il sistema di proposte di risoluzione Jira e il *chatbot* coprono l'interezza dei requisiti descritti nella tabella 3.3, garantendo che i prodotti sviluppati rispettassero le specifiche definite durante l' analisi e che soddisfasse le aspettative dell'azienda. Non avendo svolto dei *test* codificati ma solo dei *test* effettuati manualmente, non ho un valore di *code coverage* che mi permetta di quantificare il numero di righe di codice coperte dai *test*. La decisione di non svolgere dei *test* codificati è stata dettata dalla necessità di concentrarmi sullo sviluppo dei progetti e di dimostrare la fattibilità e l'utilità dei prodotti sviluppati al termine delle otto settimane di *stage*.

3.7.3 Materiali prodotti

Il mio periodo di *stage* ha visto la produzione di diversi materiali al fine di documentare il lavoro svolto e le scelte progettuali fatte. Di seguito la tabella riporta una valutazione quantitativa dell'esperienza di *stage*, in base ai diversi aspetti considerati:

Documentale	
Documenti	2
Sprint review svolte	8
Tecnico	
Handlers implementati	2
Services implementati	9
Interfacce	14
Classi implementate	3

Tabella 3.4: Materiali prodotti durante il periodo di *stage*

Capitolo 4

Valutazione retrospettiva

4.1 Raggiungimento obiettivi

4.1.1 Obiettivi aziendali

Durante il mio percorso di *stage* sono riuscito a sviluppare le funzionalità richieste dall'azienda, garantendo che i prodotti rispettassero i requisiti definiti durante l'analisi e che fossero conformi alle loro aspettative. Ho raggiunto il 100% degli obiettivi desritti nelle tabelle 3.3 e 2.2. Inoltre ho redatto la documentazione tecnica e il manuale utente per entrambi i progetti, come richiesto dall'azienda.
Nelle tabelle 4.1 e 4.2, riporto il resoconto deli obiettivi aziendali dei due progetti:

Sistema di proposte di risoluzioni

Obiettivo	**Stato**
Studio delle tecnologie e studio di fattibilità	Raggiunto
Creazione dati di *mock* da inserire nell'ITS Jira	Raggiunto
Reperimento dei ticket su Jira e salvataggio su *database* mongoDB	Raggiunto
Aggiornamento costante del *database* a nuovi *ticket* risolti su Jira	Raggiunto
Eseguire RAG sui *ticket* salvati	Raggiunto
Benchmark su vari modelli per la generazione della risposta	Raggiunto
Creazione del sistema di proposte di risoluzione su Jira	Raggiunto

Tabella 4.1: Resoconto obiettivi aziendali del sistema di proposte di risoluzione

Chatbot

Obiettivo	Stato
Studio delle tecnologie e studio di fattibilità	Raggiunto
Implementazione interfaccia *web* per l'interrogazione del *chatbot*	Raggiunto
Impostazione di un formato di domanda dell'utente da seguire	Raggiunto

Tabella 4.2: Resoconto obiettivi aziendali del *chatbot*

4.1.2 Obiettivi personali

Gli obiettivi personali che mi sono prefissato prima di iniziare lo *stage*, descritti nella sezione 2.3.2, erano volti a sviluppare competenze tecniche e personali che mi permettessero di crescere dal punto di vista personale e professionale. Durante il mio percorso di *stage* ho raggiunto tutti gli obiettivi personali che mi ero prefissato.

- **Metodologia aziendale**: ho partecipato attivamente ai processi e alle attività aziendali, inerenti allo sviluppo di un prodotto *software*, mettendo in pratica le conoscenze acquisite durante il corso di "Ingegneria del Software". Ho avuto modo di vedere come fosse strutturato il modello di sviluppo *agile* con il *framework* Scum. Ho partecipato alle attività di *stand-up meeting*, in numero limitato a causa dei numerosi impegni che il mio *tutor* aziendale aveva e *sprint review*. Quest'ultima attività è stata svolta in modo regolare, con cadenza settimanale, con il supporto di strumenti di comunicazione, collaborazione e tracciamento come Slack, Jira e Github.
- **Conoscenze tecniche**: lo *stage* mi ha permesso di approfondire le mie conoscenze attraverso lo studio di nuove tecnologie e l'applicazione pratica di queste ultime. Lo sviluppo dei due progetti mi ha permesso di lavorare attivamente con *framework* nuovi come Serverless e Streamlit e di approfondire le mie conoscenze con i linguaggi di programmazione come Typescript e Python. Inoltre, ho avuto modo di acquisire nuove conoscenze e competenze con l'utilizzo dei servizi *cloud* offerti da AWS come AWS Lambda e AWS Bedrock.
- ***Problem solving***: durante il mio percorso di *stage* ho avuto modo di affrontare problemi e sfide che mi hanno permesso di migliorare le mie capacità di *problem solving*

Nella tabella 4.3 riporto il resoconto degli obiettivi personali raggiunti durante il mio percorso di *stage*:

Codice Obiettivo	Descrizione dell'Obiettivo Personale	Stato
OP-1	Sviluppare competenze con strumenti di comunicazione e collaborazione aziendali come Slack e GitHub	Raggiunto
OP-2	Sviluppare competenze con i *framework* utilizzati, come Serverless *framework* e Streamlit	Raggiunto
OP-3	Sviluppare competenze con nuovi linguaggi di programmazione come TypeScript e Python.	Raggiunto
OP-4	Sviluppare competenze con le tecnologie offerte da AWS.	Raggiunto
OP-5	Partecipare ai processi di sviluppo *software* in ambito aziendale.	Raggiunto
OP-6	Comprendere i ritmi e le dinamiche di un lavoro in questo settore.	Raggiunto

Tabella 4.3: Resoconto obiettivi personali

4.2 Difficoltà incontrate

Durante il mio percorso di *stage* ho incontrato alcune difficoltà che si sono presentate durante le prime settimane di lavoro. Le principali difficoltà incontrate sono qui di seguito descritto con le relative soluzioni adottate:

- **Timore nel chiedere aiuto**: essendo stata la mia prima esperienza lavorativa in un'azienda operante nel settore informatico, ho avuto timore nel chiedere aiuto ai membri più esperti, per paura di disturbare il loro lavoro.
  - **Soluzione**: ho superato questa difficoltà grazie al clima di collaborazione presente in azienda, che mi ha permesso di liberarmi di ogni timore. Ho avuto modo di confrontarmi con il mio *tutor* aziendale e con gli altri membri dell'azienda per chiarire i miei dubbi
- **Difficoltà con l'utilizzo delle API di Jira**: durante la seconda e terza settimana di lavoro, come descritto nella tabella 3.1, dovevo creare dei *ticket* di *mock*, inserirli in un progetto Jira e infine reperirli per poterli salvare all'interno del *database*. Ho incontrato delle difficoltà nell'utilizzo delle API di Jira adibite all'inserimento e reperimento dei *ticket*.
  - **Soluzione**: ho superato questa difficoltà grazie al riferimento alla documentazione dettagliata delle API di Jira e dal supporto del mio *tutor* aziendale.

La strategia di mitigazione del rischio descritto nella sezione 3.2 si è rivelata efficace.

4.3 Competenze acquisite

L'esperienza di *stage* è stata molto importante per la mia crescita professionale e personale. Ho categorizzato le competenze acquisite in tre macroaree:

- **Competenze tecniche**: Durante le settimane di stage, ho utilizzato nuove tecnologie e strumenti di sviluppo come Serverless *framework*, il *framework* Streamlit, linguaggi di programmazione come TypeScript e Python, e i servizi *cloud* offerti da AWS. Lavorando costantemente su entrambi i progetti con queste tecnologie, ho potuto apprendere e applicare le conoscenze acquisite durante il corso di studi, osservando come vengono implementate in un contesto aziendale.
- **Competenze metodologiche**: L'approccio a nuove tecnologie e strumenti ha richiesto una solida organizzazione, fondamentale per apprendere e applicare le conoscenze, rispettando le tempistiche aziendali. Inoltre, ho partecipato attivamente ai processi di sviluppo *software* in un contesto aziendale, il che mi ha permesso di osservare processi ben strutturati e organizzati, comprendendo appieno i ritmi e le dinamiche del lavoro in questo settore.
- **Competenze personali**: Con il passare delle settimane in azienda, ho acquisito una maggiore sicurezza nelle mie capacità di comunicazione e di lavoro in *team*. Partecipare settimanalmente alle *sprint review* mi ha permesso di sviluppare la capacità di presentare il lavoro svolto in modo chiaro e conciso, affinando le mie capacità di comunicazione. Quest'ultima è stata ulteriormente rafforzata grazie alla presentazione che ho realizzato e presentato all'azienda, la quale mi ha permesso di affinare ulteriormente le mie abilità comunicative. Spiegare il lavoro svolto e le tecnologie utilizzate allo stagista che ha preso in mano il progetto dopo di me è stato un altro momento chiave per mettere in pratica questa abilità. Ancora grazie alle *sprint review*, ho sviluppato una maggiore capacità di ascolto e di adattamento, imparando a gestire critiche e suggerimenti in modo costruttivo. In conclusione, ho migliorato anche il mio approccio al *problem solving*, affrontando e risolvendo in modo efficace le sfide che si sono presentate durante il mio percorso di *stage*.

4.4 Divario formativo università e lavoro

L'esperienza *stage* mi ha aiutato a cogliere le connessioni tra il mondo accademico e il mondo professionale. Personalmente, ritengo che l'università e il mondo del lavoro siano due realtà complementari, ciascuna con il proprio ruolo e i propri obiettivi. L'università fornisce le basi teoriche e le competenze necessarie per affrontare il mondo del lavoro. In particolare, il corso di "Ingegneria del Software" ha avuto un ruolo fondamentale nella mia preparazione allo *stage*, grazie alle solide conoscenze teorico-pratiche acquisite, arricchite dal progetto didattico. Il mondo del lavoro, invece, consente di applicare le conoscenze apprese in un contesto reale, affrontando problemi e sfide che difficilmente emergono durante gli studi. Ritengo che questa esperienza sia fondamentale per rafforzare e ampliare le competenze che si sviluppano durante il percorso universitario. Lo *stage*, inoltre, è un'opportunità per migliorare abilità

essenziali come il lavoro in *team*, cruciale nello sviluppo di applicativi *software*, dove collaborazione e comunicazione sono alla base di qualsiasi progetto aziendale. Questo aspetto è stato trattato in modo limitato durante il corso di studi.
Il collocamento dello *stage*, avvenuto dopo il completamento del percorso accademico e del progetto didattico, mi ha permesso di affrontare questa esperienza con maggiore consapevolezza delle metodologie di lavoro, con abilità comunicative rafforzate grazie ai "Diari di bordo" del corso di "Ingegneria del Software", e competenze tecniche acquisite durante gli studi. Considerato quanto ho descritto, ritengo che il percorso di studi offra una solida base di conoscenze e competenze per affrontare il mondo del lavoro. In aggiunta, ritengo che l'università offre una formazione ampia e generale, focalizzata sullo sviluppo delle competenze trasversali e del pensiero critico, essenziali per orientarsi nel mondo professionale anche in settori diversi.

4.5 Considerazioni finali

L'esperienza di *stage* presso Zero12 ritengo essere stata estremamente positiva e formativa. Mi ha consentito di sviluppare e migliorare le mie competenze tecniche, metodologiche e personali, applicandole concretamente in un contesto aziendale. Ho avuto modo di lavorare su due progetti molto interessanti, che mi hanno permesso di approfondire le mie conoscenze sui servizi *cloud* offerti da AWS, sui *framework* Serverless e Streamlit e sui linguaggi di programmazione TypeScript e Python. Inoltre, ho potuto esplorare da vicino il tema della IA Generativa, che mi ha sempre affascinato, e che ho potuto approfondire grazie ai due progetti. L'ambiente di lavoro in cui sono state inserito è stato altamente stimolante, grazie alla presenza di membri esperti e competenti, che mi hanno supportato e con i quali ho potuto scambiare discutere sulle tematiche che ho affrontato durante il mio percorso di *stage*. In conclusione, sono molto soddisfatto dell'esperienza di *stage* svolta presso Zero12, che mi ha permesso di crescere dal punto di vista professionale e personale, preparandomi al meglio per le sfide future nel mondo del lavoro.

Acronimi e abbreviazioni

API Application Program Interface. 7, 9, 10, 18, 39, 46, 47, 55, 59

AWS Amazon Web Services. 2, 7–9, 12, 18–20, 24, 25, 32, 33, 35–43, 46, 50, 54–57, 59

CEO Chief Executive Officer. 1

CI/CD Continuous Integration & Continuous Delivery. 9, 59

CSS Cascading Style Sheets. 40

DTO Data Transfer Object. 39

HR Human Resources. 2

HTML Hypertext Markup Language. 40

HTTP Hypertext Transfer Protocol. 38, 42, 47

ITS Issue Tracking System. 14, 16, 21, 53, 59

KNN K-Nearest Neighbors. 33, 60

LLM Large Language Model. 9, 21, 22, 25–27, 29–33, 35, 36, 40–43, 45, 46, 50, 60

RAG Retrieval Augmented Generation. 16, 21, 32, 53, 60

UML Unified Modeling Language. 25

Glossario

API In informatica, con il termine *API, Application Programming Interface* si intende un insieme di protocolli che permette la comunicazione tra diverse applicazioni software. Funge da contratto di come una componente software deve interagire con un'altra, attraverso richieste e risposte. 39, 58

AWS Con il termine *AWS, Amazon Web Services* ci si riferisce ad una piattaforma di servizi cloud offerta da Amazon che offre un'ampia gamma di servizi di calcolo, archiviazione, database, analisi, applicazioni e distribuzione, permettendo alle aziende di scalare e crescere rapidamente senza dover gestire infrastrutture fisiche. 1, 58

Backlog Con il termine *Backlog* ci si riferisce ad una lista di attività da svolgere per raggiungere un determinato obiettivo. 3

Branch Con il termine *Branch* ci si riferisce ad un ramo di sviluppo, ovvero una linea di sviluppo separata dal ramo principale della *repository*. Questo permette di lavorare su nuove funzionalità o correzioni di bug senza influenzare il codice principale. 5, 45

CI/CD In informatica, con il termine *CI/CD, Continuous Integration & Continuous Delivery* ci si riferisce ad una pratica di sviluppo in cui tutte le modifiche apportate al *software* durante lo sviluppo, vengono integrate e testate automaticamente, in modo da garantire che il codice sia sempre funzionante e pronto per il rilascio. 58

Embedding Un *Embedding* è una rappresentazione numerica di oggetti, come parole o immagini in uno spazio vettoriale multidimensionale. Questa rappresentazione permette di catturare le relazioni semantiche tra gli oggetti, permettendo cosi di il confronto e l'analisi di similarità in modo più efficiente. 7, 21, 25–28, 30–33, 37, 41, 42, 44, 49

IA Generativa Con il termine *IA (Intelligenza Artificiale) Generativa* ci si riferisce ad un ramo dell'intelligenza artificiale che dato un contesto, è in grado di generare nuovi dati coerenti con esso. Questi modelli sono utilizzati per generare testo, immagini, musica e video. 14, 15, 19, 20, 57

ITS Con il termine *Issue Tracking System (ITS)* ci si riferisce ad un sistema software utilizzato per registrare, monitorare e gestire *ticket* che rappresentano *task* da svolgere, problemi, richieste di miglioramento o altre attività da risolvere all'interno di un'organizzazione. Facilita la comunicazione tra i membri del team e migliora l'efficienza nella risoluzione dei problemi, consentendo di organizzare e prioritizzare il lavoro in modo efficace. 4, 58

KNN La tecnica *KNN, K-Nearest Neighbors* è una tecnica utilizzata per la classificazione e la regressione che si basa sul concetto di vicinanza tra i dati. Viene etichettato un nuovo punto all'interno dello spazio vettoriale, in base all'etichetta dei K punti più vicini ad esso. Viene comunemente usata in ambito di *machine learning* e sistemi di raccomandazione. 33, 58

LLM Con il termine *LLM, Large Language Model* ci si riferisce ad un modello di intelligenza artificiale progettato per generare testo in linguaggio naturale. Questi modelli sono addestrati su grandi quantità di dati testuali ed utilizzano tecniche di apprendimento automatico per generare risposte coerenti al contesto della domanda. 9, 58

Prompt Engineering Il *Prompt Engineering* è un processo di progettazione e ottimizzazione dei *prompt*, ovvero delle istruzioni o domande, utilizzate per guidare i modelli di intelligenza artificiale. L'obiettivo di tale processo è ottenere un *prompt* che permetta di ottenere risposte più accurate, pertinenti e coerenti. 33

Pull Request Una *Pull Request* è una richiesta di incorporare le modifiche apportate ad un progetto da un ramo di sviluppo nel ramo principale della *repository*. Questo processo permette di revisionare le modifiche apportate e garantire che il codice sia conforme agli standard del progetto. 5, 45

RAG Con il termine *RAG, Retrieval Augmented Generation* ci si riferisce ad una tecnica avanzata nell'ambito nell'elaborazione di linguaggio naturale, che combina il recupero di informazioni con la generazione di testo, per produrre risposte con la conoscenza del contesto, creando risposte coerenti e pertinenti alla domanda dell'utente. 21, 58

Serverless *Serverless* è un modello di sviluppo di applicazioni *cloud* in cui il fornitore del servizio (AWS nel contesto aziendale), gestisce automaticamente l'allocazione delle risorse richieste dall'applicazione. L'utente pagherà soltanto per il tempo in cui il codice è in esecuzione (*pay as you go*), senza dover gestire l'infrastruttura sottostante. vii, 8, 37

StageIT *StageIT* è un evento promosso da Confindustria Veneto Est con la collaborazione dei dipartimenti di matematica e scienze statistiche dell'Università di Padova, per favorire l'incontro tra le aziende e gli studenti dei corsi di laurea di informatica, ingegneria informatica e statistica. 13, 14, 19

Tokenizzazione Con il termine *Tokenizzazione* ci si riferisce al processo di suddivisione di un testo in *token*. In ambito di LLM, un *token* possono essere parole, parti di parole o caratteri che rappresentano un'unità di testo che il modello elabora. Questo processo è fondamentale per il pre-processamento del testo, dato che determina come il modello interpreta e gestisce il contenuto linguistico durante l'addestramento e la generazione di risposte. 21

User stories Le *User stories* rappresentano una pratica *Agile*, che viene utilizzata per comprendere le esigenze dell'utente, mediante una descrizione informale, semplice e concisa della funzionalità. 3

Bibliografia

Siti web consultati

AWS API Gateway. https://docs.aws.amazon.com/apigateway/latest/developerguide/welcome.html (Visitato il 23/07/2024).

AWS Bedrock. https://docs.aws.amazon.com/bedrock/latest/userguide/what-is-bedrock.html (Visitato il 23/07/2024).

AWS Lambda. https://docs.aws.amazon.com/lambda/latest/dg/welcome.html (Visitato il 23/07/2024).

Serverless Framework. https://www.serverless.com/framework/docs (Visitato il 23/07/2024).

Sito VarGroup. https://www.vargroup.com (Visitato il 23/07/2024).

Sito Zero12. https://www.zero12.it (Visitato il 23/07/2024).