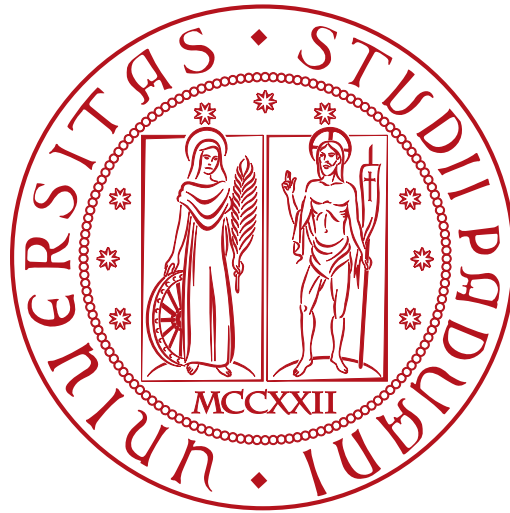


Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA "TULLIO LEVI-CIVITA"

CORSO DI LAUREA IN INFORMATICA



DocumentAI: generare documentazione dal codice sorgente

Tesi di Laurea

Relatore

Prof. Tullio Vardanega

Laureando

Giuseppe Tutino

Matricola 2075515

Ringraziamenti

Prima di tutto ringrazio infinitamente la mia famiglia ed i miei parenti, specialmente i miei genitori e la mia sorellina Irene per avermi sempre compreso e sostenuto.

Desidero ringraziare i miei amici per tutti i bei momenti passati insieme, un grazie di cuore va agli amici siciliani Alessandro, Alessio, Francesco, Johnny, Saverio ed agli amici veneti Alessio, Arbri, Basilio, Filippo, Heldin, Leonardo, Ovidio, Riccardo e Volty.

Un ringraziamento speciale va a Manuel, con cui ho avuto la fortuna di condividere questo bellissimo percorso.

Ringrazio molto Tommaso, con cui ho avuto il piacere di svolgere questo progetto.

Un sentito ringraziamento va a tutti i membri del gruppo Tech Minds, con cui ho avuto il piacere di lavorare al progetto di ingegneria del software.

Ringrazio profondamente il mio relatore, il Prof. Tullio Vardanega, per la sua disponibilità e l'indispensabile supporto durante la stesura della tesi.

Infine, ringrazio il mio tutor aziendale e l'azienda per avermi dato la possibilità di lavorare con loro.

Padova, Settembre 2025

Giuseppe Tutino

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di *stage* curricolare, della durata di trecentoventi ore, dal laureando Giuseppe Tutino presso l'azienda Var Group. Il tutor aziendale Francesco Battistella ha condotto lo stage, mentre il prof. Tullio Vardanega ha ricoperto il ruolo di tutor accademico.

Questa tesi tratta la realizzazione di **DocumentAI**, una piattaforma *web* il cui scopo è generare documentazione tecnica a partire dal codice sorgente. Tale documentazione è generata tramite l'ausilio di AI generativa.

Il documento si divide in quattro capitoli:

1. Contesto aziendale, presentazione dell'azienda con cui ho svolto lo *stage*.
2. Progetto di *stage*, cosa tratta il progetto e gli obiettivi da raggiungere.
3. Realizzazione, processo di creazione dall'analisi fino al prodotto finale.
4. Retrospettiva, resoconto dei risultati raggiunti.

Nel testo adotterò alcune convenzioni stilistiche, ovvero:

- I termini in lingua diversa dall'italiano sono posti *in corsivo*.
- I termini da glossario sono indicati con una G a pedice (es. termine_G).
- I termini che si riferiscono a degli *snippet* di codice sono indicati in `monospace`.

Indice

1. Contesto aziendale	1.
1.1. L'azienda	1.
1.2. Processi produttivi	2.
1.3. Strumenti e tecnologie	3.
1.3.1. Strumenti organizzativi	3.
1.3.2. Tecnologie per lo sviluppo	5.
1.4. Innovazione	7.
2. Progetto di <i>stage</i>	9.
2.1. Visione aziendale	9.
2.2. Idea	10.
2.2.1. Problema iniziale	10.
2.2.2. Strumenti esistenti	10.
2.2.3. DocumentAI	12.
2.3. Aspettative	13.
2.3.1. Obiettivi	13.
2.3.2. Vincoli	14.
2.4. Motivazioni della scelta	15.
3. Realizzazione	17.
3.1. Analisi	17.
3.1.1. Studio dei casi d'uso	17.
3.1.2. Specifica dei requisiti	23.
3.1.3. Analisi dei modelli	28.
3.2. Progettazione	29.
3.2.1. <i>Backend</i>	29.
3.2.2. <i>Frontend</i>	34.
3.2.3. CLI	35.
3.2.4. <i>Database</i>	36.
3.2.5. Configurazione dell'ambiente di produzione	37.

3.3. Implementazione	38.
3.4. Verifica e collaudo	41.
3.5. Prodotto finale	43.
4. Retrospettiva	46.
4.1. Risultati raggiunti	46.
4.1.1. Obiettivi aziendali	46.
4.1.2. Obiettivi personali	48.
4.2. Competenze acquisite	49.
4.2.1. Tecnologie	49.
4.2.2. Metodologie	49.
4.3. Prerequisiti fondamentali	50.
4.3.1. Competenze chiave	50.
4.3.2. Influenza del corso di studio	50.
Glossario	52.
Bibliografia	54.

Elenco delle Figure

Figura 1.1 Diagramma architetturale AWS.	1.
Figura 1.2 Processo di produzione Scrum.	2.
Figura 1.3 Eventi del processo Scrum.	3.
Figura 1.4 Lista dei <i>working item</i> in formato tabellare in Jira.	4.
Figura 1.5 Interfaccia di Slack.	5.
Figura 1.6 Architettura di un agente AI.	7.
Figura 1.7 Schema LLM + RAG.	8.

Figura 2.8	Documentazione generata da Javadoc.	11.
Figura 2.9	<i>Workflow</i> tramite DocumentAI.	12.
Figura 3.10	Funzionalità amministratore - caso d'uso.	19.
Figura 3.11	Creazione progetto - caso d'uso.	20.
Figura 3.12	Lista progetti - caso d'uso.	21.
Figura 3.13	Esempio di <i>endpoint</i> in architettura esagonale.	33.
Figura 3.14	<i>Design</i> della pagina <i>Dashboard</i>	35.
Figura 3.15	<i>Dashboard</i> - comando per il caricamento dei sorgenti.	36.
Figura 3.16	Diagramma UML del <i>database</i>	37.
Figura 3.17	Configurazione ambiente di produzione.	38.
Figura 3.18	Pagina <i>Home</i> del prodotto.	44.
Figura 3.19	Pagina <i>Auth</i> del prodotto.	44.
Figura 3.20	Pagina <i>Dashboard</i> del prodotto.	45.

Elenco delle Tabelle

Tabella 2.1	Tabella degli obiettivi.	14.
Tabella 3.2	Requisiti funzionali.	23.
Tabella 3.3	Requisiti di qualità.	26.
Tabella 3.4	Requisiti di vincolo.	27.
Tabella 3.5	Risultati analisi dei modelli.	29.
Tabella 3.6	Metriche del prodotto.	39.
Tabella 3.7	Riepilogo copertura dei requisiti.	43.
Tabella 4.8	Risultati obiettivi aziendali.	46.

Elenco dei Codici

Sorgente

Codice 1.1 Esempio di codice TypeScript.	6.
Codice 1.2 Esempio di Dockerfile.	7.
Codice 2.3 Esempio di commenti in formato Javadoc.	11.
Codice 3.4 Classe <i>Project</i>	30.
Codice 3.5 Classe <i>Documentation</i>	31.
Codice 3.6 Classe <i>User</i>	31.
Codice 3.7 Generazione della documentazione in parallelo.	40.
Codice 3.8 Generazione della documentazione in parallelo (gruppi di 5). ...	41.
Codice 3.9 Esempio di <i>test</i> utilizzando Jest.	42.

Capitolo 1.

Contesto aziendale

1.1. L'azienda

Var Group è una società italiana fondata nel 2009, con sede principale ad Empoli e molte altre sedi nel resto d'Italia e nel mondo.

L'azienda fornisce servizi di consulenza nell'ambito dello sviluppo *software*. La maggioranza dei suoi prodotti sono applicazioni *web* che vengono create su misura per soddisfare le esigenze dei clienti. Sono specializzati e certificati nella creazione di prodotti *software* con architetture scalabili, performanti e sicure attraverso l'utilizzo dei servizi *cloud* più conosciuti come AWS_G ed Azure_G.

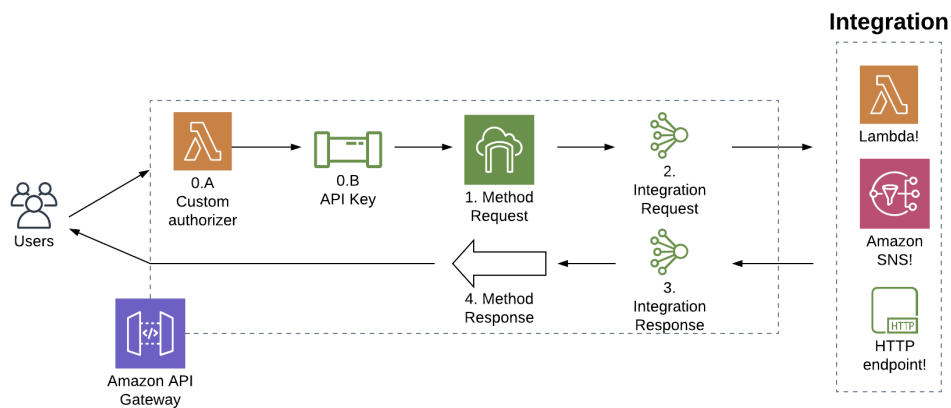


Figura 1.1: Diagramma architetturale AWS.

Fonte: <https://mitelman.engineering/blog>

Figura 1.1 presenta un esempio di architettura utilizzando i servizi di AWS_G. Si possono visionare vari servizi, come Api Gateway che facilita la creazione e manutenzione delle API_G, e Lambda che permette l'esecuzione Serverless_G di un servizio *cloud*, ovvero consente di gestire le richieste eseguendo il codice al volo eliminando la necessità di un *server* sempre attivo ed in ascolto. Utilizzando bene i giusti servizi e facendoli comunicare adeguatamente si possono ottenere *performance* ottime con un alto livello di sicurezza.

Inoltre l'azienda è molto affermata nel settore e ne sono prova concreta le numerose *partnership* che possiede con altre grandi aziende come AMD, Intel, Amazon, Microsoft, Google e molte altre.

1.2. Processi produttivi

L'azienda si occupa dei processi di sviluppo e manutenzione del *software*, mentre la messa in produzione è gestita da terzi o dal cliente. Lo sviluppo del *software* segue metodologie Agile, ovvero un insieme di tecniche di sviluppo flessibili che hanno l'obiettivo di consegnare al cliente il prodotto funzionante in tempi brevi e frequentemente.

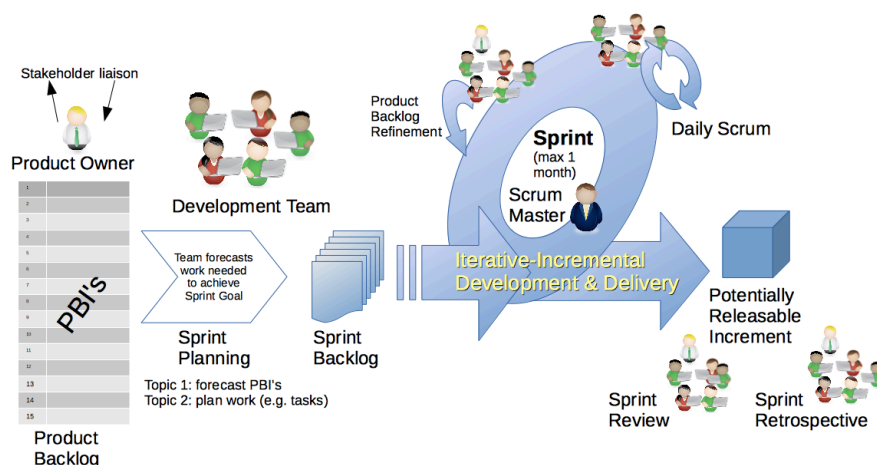


Figura 1.2: Processo di produzione Scrum.

Fonte: [https://en.wikipedia.org/wiki/Scrum_\(software_development\)](https://en.wikipedia.org/wiki/Scrum_(software_development))

Esistono varie metodologie di sviluppo Agile, nello specifico in azienda è utilizzata una metodologia chiamata Scrum.

Figura 1.2 rappresenta l'intero processo del *framework* Scrum. Un concetto fondamentale è quello di *sprint*, che rappresenta un periodo di tempo fissato, solitamente fra una e quattro settimane nel quale il *team* lavora per raggiungere certi obiettivi.

Scrum si basa su 3 principi fondamentali: responsabilità, artefatti ed eventi.

Nel *team* sono presenti tre categorie di individui, di cui ognuno ha le proprie responsabilità. Il *Product owner* interagisce in modo diretto con il cliente ed ha la responsabilità del conseguimento dei risultati aziendali. Gli sviluppatori, la categoria più popolosa, si occupano di sviluppare e supportare il prodotto.

Infine lo *Scrum master* è un supervisore che si assicura che vengano rispettate le pratiche Scrum.

Gli artefatti principali sono il *Product backlog*, che contiene la lista ordinata delle attività da svolgere, e lo *Sprint backlog*, un sottoinsieme di attività del *Product backlog* da completare in un determinato *sprint*.

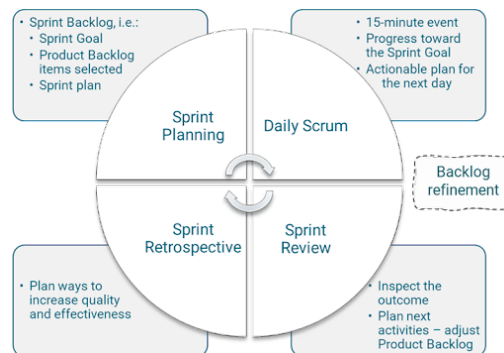


Figura 1.3: Eventi del processo Scrum.

Fonte: [https://en.wikipedia.org/wiki/Scrum_\(software_development\)](https://en.wikipedia.org/wiki/Scrum_(software_development))

Figura 1.3 mostra gli eventi principali in Scrum, che si ripetono ciclicamente ad ogni *sprint* e sono:

1. *Sprint planning*: incontro che avviene all'inizio dello *sprint*, nel quale i membri del *team* pianificano il lavoro da svolgere, spostando le attività da svolgere dal *Product backlog* allo *Sprint backlog*.
2. *Daily scrum*: incontro giornaliero breve (meno di 15 minuti) in cui ogni membro del *team* aggiorna gli altri riguardo i suoi progressi, cosa deve svolgere e se ha riscontrato problemi bloccanti.
3. *Sprint review*: revisione in cui tutti i membri del *team* con il cliente visionano i risultati dello *sprint*.
4. *Sprint retrospective*: evento immediatamente successivo alla *Sprint review* in cui i membri del *team* ragionano su cosa ha funzionato e cosa migliorare nei prossimi *sprint*.

1.3. Strumenti e tecnologie

1.3.1. Strumenti organizzativi

Jira

Il tracciamento e monitoraggio delle attività viene gestito tramite Jira, un ITS_G (*Issue tracking system*), ovvero uno strumento che permette di tracciare e defi-

nire in modo preciso le attività da svolgere, chiamate nel gergo *issue*. Un'attività in Jira prende il nome di *working item*, ad esso possono essere associate molte proprietà come il tipo, descrizione, tempo di inizio, tempo di fine, assegnatario e molto altro.

<input type="checkbox"/>	Type	Key	Summary	Status	Comments	Assignee	Due date	Labels	Created	Updated	Reporter	+
<input type="checkbox"/>	▼ ↕	WB-1	Frontend	TO DO	🗨️ Add comment				📅 Jul 10, 2025	📅 Jul 10, 2025	👤 User	
<input type="checkbox"/>	▼ ↕	WB-2	Admin Page	TO DO	🗨️ Add comment				📅 Jul 10, 2025	📅 Jul 10, 2025	👤 User	
<input type="checkbox"/>	🔗	WB-4	User list	TO DO	🗨️ Add comment				📅 Jul 10, 2025	📅 Jul 10, 2025	👤 User	
<input type="checkbox"/>	↕	WB-3	Home	TO DO	🗨️ Add comment				📅 Jul 10, 2025	📅 Jul 10, 2025	👤 User	

Figura 1.4: Lista dei *working item* in formato tabellare in Jira.

Fonte: Autore

Figura 1.4 mostra una vista in formato tabellare di alcuni *working item*. Precisamente sono presenti *epic* (WB-1), *user story* (WB-2 e WB-3) e *subtask* (WB-4), che seguono una struttura ad albero.

Epic, *user story* e *subtask* sono le tipologie di *working item* principali, ovviamente possono essere create altre tipologie qualora fosse necessario. Tipicamente gli *epic* rappresentano *feature* grandi e descritte ad alto livello. Ogni *epic* viene suddivisa in più *user story*, delle *feature* più concrete e realizzabili nella durata di uno *sprint*. Se necessario le *user story* possono essere suddivise a loro volta in *subtask* più precise.

GitHub

Il codice sorgente, la documentazione e qualsiasi artefatto riguardante il prodotto viene gestito su GitHub, una piattaforma che permette di condividere e versionare il codice. L'intera piattaforma si basa su Git, un VCS_G (*Version Control System*), ovvero un *software* che ha lo scopo di tenere traccia di tutti i cambiamenti che avvengono nei *file* di progetto. Il codice di un progetto si raggruppa in *repository*, delle collezioni indipendenti.

Slack

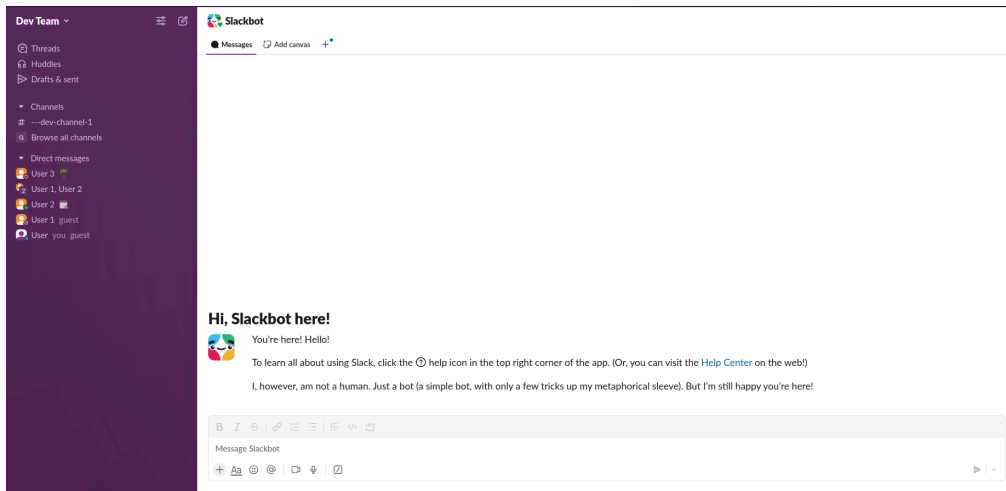


Figura 1.5: Interfaccia di Slack.

Fonte: Autore

La comunicazione asincrona all'interno dei *team* è gestita tramite Slack, uno strumento di messaggistica istantanea. Figura 1.5 mostra l'interfaccia che è molto simile alle classiche applicazioni di messaggistica, con un pannello laterale contenente la lista delle chat individuali e quelle di gruppo (chiamate anche canali). Inoltre possiede un *marketplace* delle estensioni, permettendo l'integrazione con molte altre piattaforme e servizi.

Microsoft Teams

Per le conversazioni sincrone e i *meeting* attraverso videoconferenza si utilizza Microsoft Teams. Essendo un prodotto fornito da Microsoft è già integrato con la suite *office*. Inoltre presenta altri vantaggi come l'organizzazione delle conversazioni e dei documenti per progetti e supporta fino a 1000 partecipanti in videoconferenza.

1.3.2. Tecnologie per lo sviluppo

Per lo sviluppo *software* vengono utilizzate varie tecnologie legate soprattutto all'ecosistema Node.js.

TypeScript

Il linguaggio di programmazione principalmente utilizzato è TypeScript, un *superset* di JavaScript. I vantaggi che si ha nell'usarlo riguardano soprattutto la presenza dei tipi a tempo di compilazione, che permettono di trovare dei *bug* e inconsistenze prima che il codice venga effettivamente eseguito. Inoltre aggiunge altri costrutti come tipi generici, classi, interfacce, ereditarietà e moduli.

```
1 function sum(a: number, b: number): number {  
2     return a + b;  
3 }
```

typescript

Codice 1.1: Esempio di codice TypeScript.

Codice 1.1 presenta un esempio di codice TypeScript in cui è presente una funzione che somma due numeri e restituisce il risultato. I tipi per ogni parametro e quello di ritorno sono indicati dopo il simbolo `:`, a differenza di altri linguaggi.

React

React è una libreria JavaScript utilizzata per lo sviluppo di *single page application*, ovvero una applicazione *web* che carica una sola pagina HTML e aggiorna dinamicamente il contenuto della pagina in risposta alle interazioni dell'utente, senza dover ricaricare l'intera pagina. Esso facilita lo sviluppo di componenti riutilizzabili per gestire al meglio l'interfaccia utente. Inoltre è presente un vasto ecosistema di librerie ed ha una grande *community* di sviluppatori.

NestJS

NestJS è un *framework* per lo sviluppo di applicazioni lato *backend* che permette di creare applicazioni scalabili e manutenibili attraverso un'architettura modulare. Ha a disposizione molti strumenti come la Dependency injection, che semplifica la creazione delle istanze delle classi, vari *middleware* che permettono di gestire le richieste in modo flessibile e integrazioni con le tecnologie più disparate.

MongoDB

MongoDB è un *database* non relazionale orientato ai documenti. Non organizza i dati in tabelle ma in documenti con un formato chiamato BSON (*binary JSON*). I vantaggi principali sono la flessibilità riguardo la struttura dei dati, infatti essa può cambiare senza creare nessun problema e permette di scalare orizzontalmente tramite *sharding*, andando a distribuire i dati su più nodi.

Docker

Docker è un *software* che utilizza la virtualizzazione a livello del sistema operativo per eseguire le applicazioni in ambienti isolati chiamati *container*. Questi *container* sono leggeri e condividono il *kernel* del sistema operativo, il che consente di eseguire più istanze di applicazioni in modo efficiente e con un utilizzo ottimale delle risorse. Una caratteristica di Docker è l'uso di immagini, ovvero dei file *template* facili da distribuire con cui si possono istanziare i

container. Infatti esistono vari *repository* di immagini Docker in cui è possibile reperire una vasta quantità di *software* sotto forma di immagine.

```
1 FROM ubuntu:20.04
2 WORKDIR /app
3 COPY . .
4 RUN apt-get update && apt-get install -y python3
5 CMD ["python3", "app.py"]
```

Codice 1.2: Esempio di Dockerfile.

Per creare le immagini si usano dei Dockerfile utilizzando un metalinguaggio. Codice 1.2 presenta un esempio di Dockerfile che prendendo come immagine di partenza Ubuntu esegue lo *script* Python `app.py`.

1.4. Innovazione

Essere al passo con le nuove tecnologie e l'innovazione è un elemento cruciale per le aziende, infatti Var Group dimostra molto interesse e impegno nello stare al passo con le ultime novità.

C'è molto interesse nel campo dell'intelligenza artificiale generativa, visti i progressi avvenuti negli ultimi anni con gli LLM_G. Più nello specifico si studiano e utilizzano tecnologie come RAG e agenti AI, cercando di sperimentare con progetti in collaborazione con le università ed altre aziende. Inoltre l'azienda organizza e partecipa ad eventi di frontiera per discutere in dettaglio di questi argomenti con altri esperti del settore.

Agenti AI

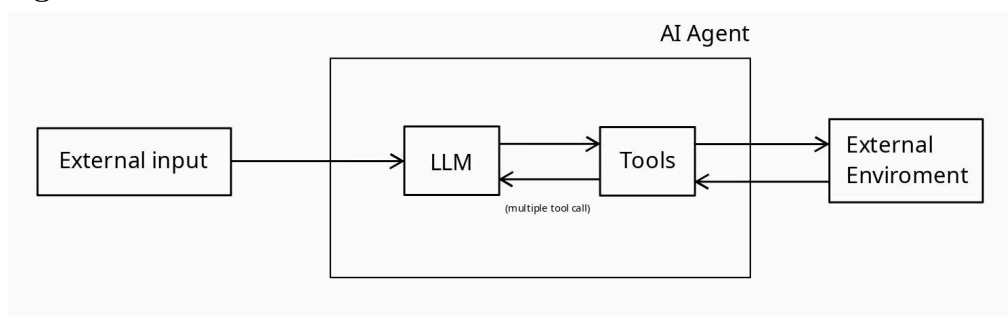


Figura 1.6: Architettura di un agente AI.

Fonte: Autore

Riguardo gli agenti AI c'è stato molto interesse ancor prima che diventassero popolari. Infatti l'azienda ha avuto modo di collaborare con l'università proprio

per proporre a gruppi di studenti dei progetti innovativi, con lo scopo di scoprire quanto utili ed affidabili potrebbero essere gli agenti in certi contesti.

Figura 1.6 rappresenta cosa è concretamente un agente AI e come funziona. Di fatto si tratta di un LLM_G che prende degli *input* esterni (sia da un utente umano che non) e li processa avendo a disposizione dei *tool* che può chiamare per accedere a risorse esterne ed effettuare richieste. Ultimamente sta diventando popolare l'uso di questi agenti nello sviluppo *software*, integrandoli negli strumenti di sviluppo.

RAG

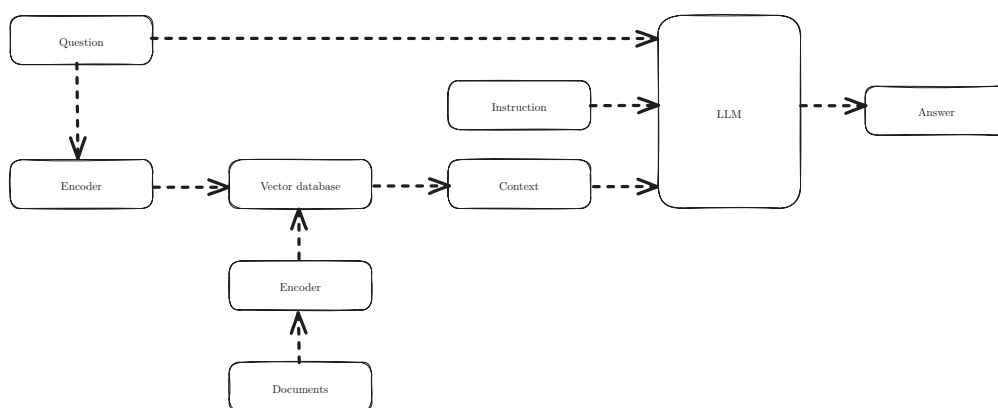


Figura 1.7: Schema LLM + RAG.

Fonte: https://commons.wikimedia.org/wiki/File:RAG_schema.svg

RAG, acronimo di *Retrieval Augmented Generation*, è una tecnica utilizzata in combinazione con gli LLM_G per migliorare la qualità e la pertinenza delle risposte generate, integrando informazioni recuperate da fonti esterne come il *web* o *database*.

Figura 1.7 presenta il funzionamento di questa tecnologia. Sostanzialmente si usa *database* vettoriale a cui si possono fare delle *query* per recuperare le informazioni dai documenti. Dopo aver recuperato le informazioni il modello le utilizza per generare la risposta. Questa tecnica risulta utile per creare dei *chatbot* di assistenza o in generale per trovare informazioni in grandi quantità di dati.

Capitolo 2.

Progetto di *stage*

2.1. Visione aziendale

Var Group collabora attivamente con le università attraverso gli *stage* curriculari, realizzando progetti innovativi. I progetti non sono attività a sé stanti ma fanno parte di una strategia più grande, che comprende obiettivi a lungo termine.

Reclutamento

L'azienda partecipa ad eventi e progetti riguardanti corsi tenuti dall'università, ad esempio lo StageIT ed il progetto del corso di ingegneria del *software*. Attraverso lo StageIT si tengono dei colloqui formali in cui c'è un incontro in presenza fra studenti e aziende, nel quale si possono comprendere gli interessi da entrambe le parti e discutere dei progetti proposti. Dopo il colloquio formale, se c'è stato buon esito da entrambe le parti, ci si può accordare per svolgere effettivamente uno dei progetti proposti.

L'azienda recluta per i progetti ma ovviamente c'è anche un secondo fine, a fine *stage* se lo stagista dimostra di essere capace potrebbe entrare a far parte dell'azienda, evitando oltretutto i soliti colloqui che si farebbero di norma.

Strategia aziendale

I progetti degli *stage* sono una serie di incrementi che a lungo andare, nell'insieme, portano verso il risultato finale. È indispensabile utilizzare questa metodologia perché sarebbe sciocco pensare di poter ottenere un prodotto che possa essere utilizzato in produzione in soli due mesi di lavoro.

Tutti i progetti proposti cercano di andare a «toccare» gli ambiti più nuovi e di frontiera, ad esempio gli LLM_G nell'intelligenza artificiale come nel mio caso. In questo modo si acquisiscono le conoscenze pratiche ed i limiti che si hanno nel loro utilizzo. Dopo aver ottenuto le giuste conoscenze, si potrebbero utilizzare per i prodotti sul mercato senza incorrere in grossi rischi.

Attraverso questa strategia l'azienda è in grado di restare al passo con l'innovazione con uno sforzo minimo ed inoltre di conoscere gente capace che potrebbe assumere.

Visione dello sviluppo *software*

Il fatto che il progetto sia realizzato da studenti universitari è un dettaglio da non trascurare. Studenti e aziende possono avere visioni e idee diverse riguardo le pratiche dello sviluppo *software*. Questo contrasto di idee può portare lo studente a comprendere meglio il funzionamento di certe metodologie in azienda ma anche i membri dell'azienda a conoscere nuovi modi di realizzare il *software*.

2.2. Idea

2.2.1. Problema iniziale

L'idea di questo progetto è nata da un problema sostanziale che esiste da decenni e che sarà presente finchè si scriverà *software*, ovvero la produzione di documentazione.

Nella maggior parte dei contesti aziendali, soprattutto dal punto di vista delle figure che hanno ruoli gestionali, scrivere la documentazione è visto spesso come un intralcio che devia dal procedere con le nuove *feature* e correggere i *bug*. Dal punto di vista degli sviluppatori, per fortuna, si comprende l'importanza di documentare il *software*. Però un buon sviluppatore cerca sempre di ridurre lo sforzo e automatizzare il più possibile. Di conseguenza sarebbe utile avere un qualche strumento che renda la stesura della documentazione un processo automatico (o semiautomatico), pur non rinunciando alla qualità.

2.2.2. Strumenti esistenti

Nel corso degli anni gli sviluppatori si sono già confrontati con questo problema e sono state create delle soluzioni per ridurre il tempo da dedicare alla documentazione. Infatti esistono strumenti che permettono di agevolare il processo di documentazione, attraverso la scrittura di commenti speciali all'interno del codice.

```

1  /**
2   * This class provides some math methods.
3   */
4  public class Calculator {
5
6     /**
7     * Sum two integers.
8     *
9     * @param a the first number to be summed
10    * @param b the second number to be summed
11    * @return the sum of a and b
12    */
13    public int sum(int a, int b) {
14        return a + b;
15    }
16 }

```

Codice 2.3: Esempio di commenti in formato Javadoc.

Un esempio di questi strumenti è Javadoc, utilizzato per il linguaggio Java. Codice 2.3 mostra una classe Java che contiene un metodo per sommare due numeri interi. Sopra ad ogni classe, metodo e variabile si possono inserire dei commenti speciali che iniziano con `/**`. L'uso di commenti del genere è una pratica che si usa da prima che venissero inventate queste tecnologie, infatti essi possono essere visti come delle «interfacce» che descrivono in modo astratto la struttura sottostante.

Class Calculator
java.lang.Object[Ⓢ]
 Calculator

public class **Calculator**
 extends Object[Ⓢ]

This class provides some math methods.

Constructor Summary

Constructors	Description
Calculator()	

Method Summary

Modifier and Type	Method	Description
int	sum(int a, int b)	Sum two integers.

Figura 2.8: Documentazione generata da Javadoc.

Fonte: Autore

Eseguendo lo strumento verrà creata una pagina HTML contenente tutti i commenti speciali formattati con un certo stile, ottenendo così la documentazione.

Figura 2.8 mostra il documento generato da Javadoc dell'esempio mostrato in Codice 2.3

Strumenti del genere funzionano da anni e continueranno ad essere utilizzati per molti altri anni, visto che svolgono bene il loro lavoro e agevolano il processo di documentazione. Tuttavia richiede allo sviluppatore di dedicare del tempo a scrivere questi commenti e mantenerli aggiornati nel corso del tempo, quindi in alcuni casi rappresenterebbe lo stesso problema iniziale. Inoltre questa soluzione risulta impraticabile quando il *software* è già stato scritto senza nessun tipo di documentazione, ad esempio in prodotti *legacy*.

2.2.3. DocumentAI

Visti gli ultimi sviluppi dell'intelligenza artificiale, Var Group ha deciso di provare a creare uno strumento che potrebbe agevolare e automatizzare la creazione della documentazione, lasciando allo sviluppatore solo l'onere di leggerla e verificarla.

Lo strumento in questione si chiama DocumentAI ed è un'applicazione *web*. Esso permette di generare documentazione utilizzando solamente il codice sorgente di un progetto, attraverso l'uso degli LLMG.

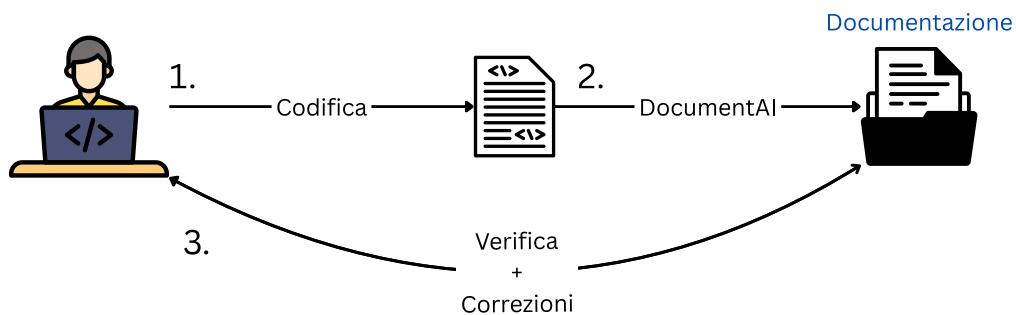


Figura 2.9: *Workflow* tramite DocumentAI.

Fonte: Autore

Figura 2.9 mostra come potrebbe essere integrato lo strumento durante la realizzazione di un prodotto.

Lo sviluppatore segue i seguenti passi ciclicamente:

1. Codifica della funzionalità.
2. Generazione della documentazione tramite lo strumento.
3. Verifica della documentazione generata, applicando correzioni se necessario.

In questo modo è molto più semplice documentare il *software*, però ci sono dei compromessi legati proprio all'uso degli LLM_G. Infatti essi riescono a comprendere il codice e spiegarlo, però non è detto che riescano a ricostruire l'intero processo cognitivo dello sviluppatore durante la codifica, rischiando di omettere dettagli importanti. Proprio per questo motivo è opportuno che la documentazione generata sia sempre verificata.

2.3. Aspettative

2.3.1. Obiettivi

L'azienda ha come obiettivo principale di ottenere un *software* funzionante, che possa davvero essere utile per generare e gestire la documentazione.

Obiettivi obbligatori

Riguardo la generazione della documentazione è essenziale che lo strumento sia capace di gestire almeno tre linguaggi di programmazione, che gestisca progetti con molti *file* e che abbia in *output* un formato standardizzato. Oltre alla parte tecnica della generazione si vuole ottenere anche un'interfaccia utente funzionale, che permetta in modo intuitivo di operare sui progetti e sulla documentazione garantendo la compatibilità su tutti i *browser* moderni.

Obiettivi opzionali

Sarebbe interessante che il *software* possa gestire una quantità più ampia di linguaggi e che permetta l'uso collaborativo attraverso dei *team*. Inoltre si potrebbe gestire un sistema di autenticazione per gli utenti e permettere la condivisione della documentazione. Infine si potrebbero ottimizzare le prestazioni nei grandi progetti.

Totale obiettivi

Obiettivo	Descrizione	Tipo
OB1	Supporto di tre linguaggi.	Obbligatorio
OB2	Gestione di molti <i>file</i> .	Obbligatorio
OB3	Formato standard per la documentazione.	Obbligatorio
OB4	Interfaccia utente funzionale.	Obbligatorio
OB5	Compatibilità sui maggiori <i>browser</i> .	Obbligatorio
OB6	Supportare più di tre linguaggi.	Opzionale
OB7	Utilizzo collaborativo tramite <i>team</i> .	Opzionale
OB8	Autenticazione utenti.	Opzionale
OB9	Condivisione della documentazione.	Opzionale
OB10	Ottimizzare prestazioni nei grandi progetti.	Opzionale

Tabella 2.1: Tabella degli obiettivi.

2.3.2. Vincoli

Comincio precisando che questo progetto è stato realizzato in coppia con un altro stagista, di conseguenza è stato indispensabile organizzare e suddividere bene il lavoro per poter ottenere i risultati attesi.

Vincoli tecnologici

Il funzionamento e la corretta riuscita del prodotto dipendono intrinsecamente dal modello di intelligenza artificiale e dalla sua configurazione, quindi ad esempio il *prompt* e la temperatura. Per avere dei risultati efficaci bisogna trovare i modelli più adatti ad estrarre dal codice le informazioni più rilevanti. Inoltre i modelli non conoscono tutti i linguaggi di programmazione allo stesso modo, perché i dati di *training* riguardo al codice non sono distribuiti in modo uniforme in base al linguaggio.

Vincoli temporali

Il progetto di *stage* ha la durata di 8 settimane lavorative, di conseguenza

bisogna accettare dei compromessi per poter ottenere un risultato utilizzabile entro questi tempi.

Pianificazione del lavoro

Uno dei documenti che bisogna presentare per poter svolgere lo *stage* è il piano di lavoro, che riassume gli obiettivi principali e quello che si andrà a svolgere settimana per settimana. In ogni settimana si dovrebbero conseguire degli obiettivi precisi, arrivando a completare tutti gli obiettivi (almeno obbligatori) alla fine dell'ultima settimana. La pianificazione del lavoro avviene all'inizio della settimana facendo riferimento al piano di lavoro e adattandolo in funzione dell'andamento degli obiettivi completati. Nel concreto scegliamo delle attività, precedentemente definite su Jira, facendo in modo che possano essere completate entro una settimana.

Allineamento

Per la buona riuscita del progetto abbiamo deciso di allinearci sui progressi ed eventuali dubbi con il *tutor* aziendale (ed altri membri dell'azienda) con incontri settimanali. Ad ogni incontro si controllano i progressi fatti riguardo il piano di lavoro e lo stato dei requisiti. Dopo l'allineamento si passa a discutere dei prossimi obiettivi, precisando le possibili scelte da adottare per verificare la loro validità.

Oltre gli allineamenti periodici, è possibile chiedere suggerimenti riguardo qualsiasi dubbio o problema in ogni momento, sia al *tutor* aziendale che ai membri dell'azienda. Potendo chiedere suggerimenti a più persone la risoluzione dei problemi diventa più efficace, visto che permette di ragionare su più punti di vista a volte differenti.

2.4. Motivazioni della scelta

Ho deciso di svolgere il mio *stage* presso Var Group per due motivi principali.

Capacità

Il primo motivo riguarda quello che vorrei ottenere alla fine dello *stage*. Vorrei prima di tutto apprendere nuove capacità riguardanti le tecnologie e le metodologie, visto che il progetto deve essere formativo. Oltre a questo voglio ottenere un'architettura modulare e ben strutturata, sia per la mia soddisfazione personale che per facilitare gli sviluppi futuri del *software* dopo la fine dello *stage*. Dati questi obiettivi ho scelto Var Group con la certezza che mi avrebbero

permesso molte libertà dal punto di vista realizzativo, data la mia conoscenza pregressa con loro.

Inoltre nella realizzazione del progetto ci sono vari modi per gestire il flusso di dati dei progetti e la loro documentazione, ciò vuol dire che è un progetto non banale e di conseguenza realizzandolo bene potrei apprendere molto.

Esperienza pregressa

Il secondo motivo è la conoscenza pregressa dell'azienda. Ho già collaborato con loro in passato per lo svolgimento del progetto del corso di ingegneria del *software*. Ero già stato negli uffici e avevo fatto conoscenza con alcuni dipendenti, restando molto colpito dall'ambiente aziendale.

Obiettivi personali

Riassumendo, i miei obiettivi personali per questo progetto sono:

1. Comprendere quanto più possibile come funziona di come funziona AWS_G ed i suoi servizi, in modo da poterli utilizzare al meglio per ottenere una buona architettura ed un buon prodotto.
2. Apprendere come vengono utilizzati certi strumenti in azienda, soprattutto la gestione dei *repository* su GitHub con le *monorepo*.
3. Capire più a fondo com'è strutturata e come funziona un'architettura a microservizi.

Capitolo 3.

Realizzazione

3.1. Analisi

3.1.1. Studio dei casi d'uso

Dopo aver studiato le varie tecnologie necessarie per lo sviluppo ed aver capito a fondo gli obiettivi del progetto, abbiamo iniziato a definire in dettaglio i requisiti attraverso il processo di analisi dei requisiti. Abbiamo svolto l'analisi attraverso l'uso di User Story_G, descrizioni ad alto livello di funzionalità dal punto di vista utente, inizialmente definite in modo veloce su dei documenti Google condivisi e successivamente in modo più preciso e formale su Jira.

Le User Story_G ottenute attraverso il processo di analisi dei requisiti, raggruppate per *epic* (indicato fra parentesi) sono riportate di seguito.

US1 - *Login* (Gestione utenti)

Come utente non autenticato, voglio poter fare il *login* per accedere alle funzionalità dell'applicativo.

Criteri di accettazione:

1. L'utente deve poter inserire le proprie credenziali (*email* e *password*) in un modulo di *login*.
2. Se l'utente inserisce credenziali corrette, il sistema reindirizza l'utente alla *Dashboard*.
3. Se l'utente inserisce credenziali errate, il sistema mostra un messaggio di errore.

US2 - *Registrazione* (Gestione utenti)

Come utente non autenticato, voglio potermi registrare per accedere alla piattaforma.

Criteri di accettazione:

1. L'utente deve poter compilare un modulo di registrazione con i campi richiesti (nome, cognome, *email*, *password*).

2. Il sistema effettua la verifica che l'utente non sia già registrato in piattaforma.
3. Dopo la verifica, il sistema invia una *email* all'utente che si sta registrando con un codice per poter attivare il proprio *account*.
4. Quando l'utente inserisce il codice ricevuto nella *webapp*, il suo *account* viene attivato ed ha la possibilità di accedere.

US3 - Procedura *password* dimenticata (Gestione utenti)

Come utente non autenticato che possiede un *account*, devo poter reimpostare la *password* così da poter accedere nuovamente alla piattaforma.

Criteri di accettazione:

1. Sulla schermata di *login* deve essere presente un *link* per avviare la procedura di recupero *password*.
2. Durante la procedura di recupero *password* l'utente inserisce una *email* che deve appartenere ad un *account* registrato.
3. Il sistema invia una *email* contenente un codice da inserire nella schermata di recupero *password* assieme al campo che conterrà quella nuova.

US4 - *Logout* (Gestione utenti)

Come utente autenticato voglio poter eseguire il *logout* così da poter terminare la mia sessione.

Criteri di accettazione:

1. L'utente autenticato ha a disposizione un pulsante per poter effettuare il *logout*.
2. L'utente autenticato che preme il pulsante di *logout* riceve un avviso di conferma.
3. Dopo che l'utente ha cliccato il pulsante di conferma *logout*, viene reindirizzato alla pagina principale e la sua sessione viene terminata.

US5 - Funzionalità amministratore (Gestione utenti)

Come utente amministratore devo poter accedere al pannello amministratore, così da poter gestire gli *account*.

Criteri di accettazione:

1. L'utente amministratore riesce ad accedere alla pagina di amministrazione utenti.
2. L'utente amministratore, dalla pagina amministratore può disattivare e riattivare l'*account* di un utente.

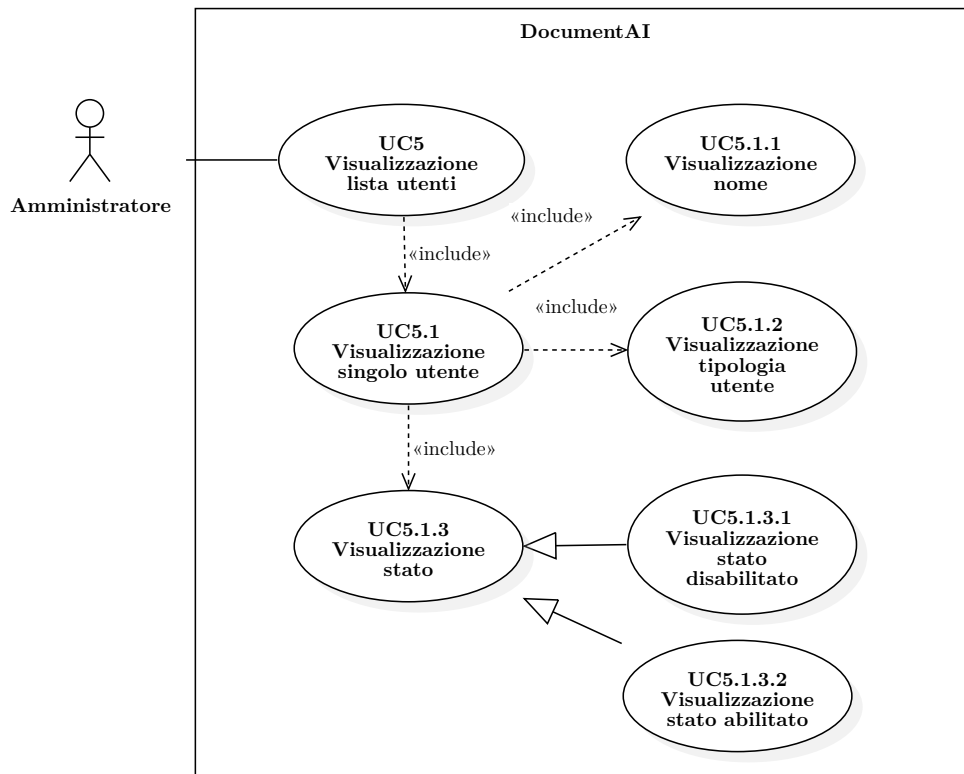


Figura 3.10: Funzionalità amministratore - caso d'uso.

Fonte: Autore.

US6 - Creazione progetto (Gestione progetti)

Come utente autenticato voglio poter creare un progetto *software* per poterlo far analizzare dal sistema.

Criteri di accettazione:

1. Poter creare un progetto.
2. Poter caricare i *file* del progetto.
3. Durante la fase di creazione l'utente può selezionare i linguaggi presenti all'interno del progetto *software*.
4. Quando l'utente crea un progetto, il sistema lo salva e lo associa al suo *account*.

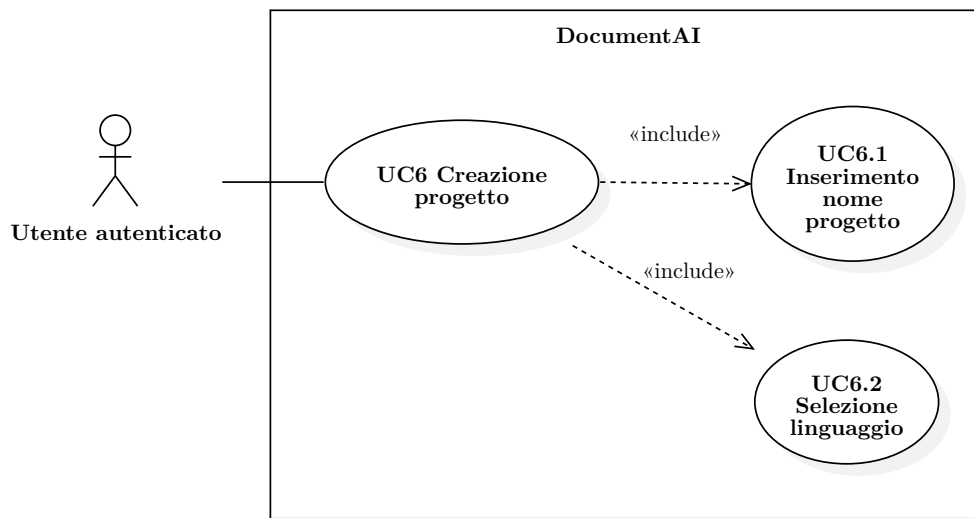


Figura 3.11: Creazione progetto - caso d'uso.

Fonte: Autore.

US7 - Modifica progetto (Gestione progetti)

Come utente autenticato voglio poter modificare i dati di un progetto per adattarlo ai cambiamenti che sono avvenuti.

Criteri di accettazione:

1. Poter modificare i dati di un progetto.
2. Poter ricaricare i *file* del progetto.
3. Quando l'utente modifica un progetto, il sistema salva tutte le informazioni.

US8 - Cancellazione progetto (Gestione progetti)

Come utente autenticato devo poter cancellare un progetto caricato così da poter gestire i miei *upload*.

Criteri di accettazione:

1. L'utente ha a disposizione un pulsante per cancellare il progetto.
2. Quando l'utente preme il pulsante, viene avvisato della cancellazione e gli viene chiesto di confermare l'azione.
3. Dopo aver confermato l'azione di cancellazione del progetto, l'utente viene avvisato sull'esito dell'operazione (successo/errore).

US9 - Lista progetti (Gestione progetti)

Come utente autenticato voglio poter vedere la lista dei miei progetti così da avere una panoramica completa di quelli disponibili.

Criteri di accettazione:

1. Nella pagina /dashboard è visibile una barra laterale che mostra la lista dei progetti caricati.
2. Per ogni progetto nella lista viene mostrato il nome, la data di creazione, il linguaggio e se sono stati caricati i *file*.

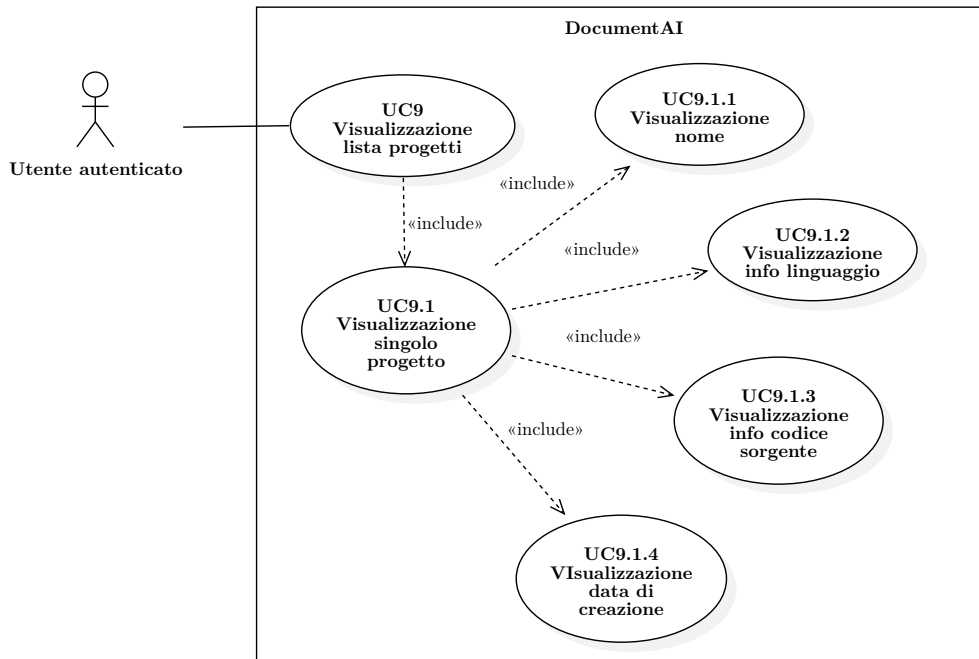


Figura 3.12: Lista progetti - caso d'uso.

Fonte: Autore.

US10 - *Download utility per caricare progetti* (Gestione progetti)

Come utente qualsiasi devo poter scaricare un programma che mi consenta di caricare il codice sorgente così da poter usufruire di una procedura automatizzata.

Criteri di accettazione:

1. L'utente nella pagina dedicata al manuale ha a disposizione un pulsante che gli permette di scaricare il programma.

US11 - *Generazione documentazione* (Documentazione)

Come utente autenticato voglio poter far generare la documentazione del progetto caricato per poterla consultare successivamente.

Criteri di accettazione:

1. Il sistema genera la documentazione utilizzando il modello scelto dall'utente tra quelli disponibili, analizzando ogni *file* sorgente.
2. Il sistema salva e associa al progetto la documentazione prodotta.

3. L'utente ha la possibilità di far rigenerare i documenti, mantenendo l'accesso alle versioni precedenti.

US12 - Consultazione documentazione (Documentazione)

Come utente autenticato voglio poter consultare i *file* contenenti la documentazione generata per poterla revisionare.

Criteri di accettazione:

1. L'utente può consultare il documento generato, raggruppato per progetto.
2. L'utente può accedere solamente ai progetti che ha creato.
3. Se l'utente ha generato più volte i documenti, deve essere in grado di consultare tutte le versioni.

US13 - Esportazione documentazione (Documentazione)

Come utente autenticato voglio poter esportare la documentazione generata in formato PDF in modo da consultarla localmente.

Criteri di accettazione:

1. L'utente può esportare un documento in formato PDF che contiene tutta la documentazione associata ad un singolo progetto nella versione che sta consultando in quel momento.

US14 - Personalizzazione *template* (Documentazione)

Come utente autenticato voglio poter personalizzare il *template* della documentazione generata così da poterla adattare alle mie esigenze.

Criteri di accettazione:

1. L'utente ha a disposizione una pagina in cui può modificare la struttura del documento.
2. L'utente che ha modificato il *template* deve poter vedere l'effetto delle sue modifiche nei nuovi documenti generati.

US15 - Condivisione documentazione generata (Documentazione)

Come utente autenticato che ha generato la documentazione di un progetto devo poterla condividere attraverso un *link* in modo da inviarla alle persone interessate.

Criteri di accettazione:

1. L'utente ha a disposizione un pulsante che gli permette di generare un *link* condivisibile.
2. Accedendo al *link* è possibile vedere solo la documentazione condivisa.

US16 - Manuale (Pagine secondarie)

Come utente voglio poter accedere alla pagina manuale così da studiare come funziona il servizio e come caricare i *file*.

Criteri di accettazione:

1. L'utente visualizza la pagina quando va alla *route* /manual.
2. Nella pagina manuale è presente una guida che spiega:
 1. come creare un nuovo progetto;
 2. come utilizzare l'*utility* a riga di comando per caricare i progetti;
 3. come generare la documentazione e utilizzare il servizio DocumentAI.

US17 - *Landing page* (Pagine secondarie)

Come utente voglio poter accedere alla pagina *Home* che mostra il nome del sito e descrive le funzionalità del servizio.

Criteri di accettazione:

1. La pagina è presente alla *route* /.
2. La pagina mostra il nome, il logo, un sottotitolo e dei pannelli che elencano cosa può fare il servizio, compreso di esempio.

3.1.2. Specifica dei requisiti

Tabella 3.2, Tabella 3.3 e Tabella 3.4 mostrano rispettivamente il resoconto dei requisiti funzionali, di qualità e di vincolo. Il codice identificativo indica la tipologia del requisito e la sua priorità.

Il primo carattere indica la tipologia:

- F (*functional*), funzionalità del sistema.
- Q (*qualitative*), proprietà del sistema che non sono direttamente legate alle funzionalità, ma che influenzano l'esperienza dell'utente.
- C (*constraint*), vincoli che il sistema deve rispettare.

Il secondo carattere indica la priorità:

- M (*mandatory*), necessari per qualcuno degli *stakeholder*.
- D (*desirable*), non necessari ma portano valore aggiunto.
- O (*optional*), non necessari o non ritenuti importanti al momento.

Requisito	Descrizione	Fonti
FMR1	L'utente non autenticato deve poter effettuare il <i>login</i> inserendo <i>email</i> e <i>password</i> .	US1
FMR2	L'utente non autenticato deve poter ricevere un messaggio d'errore nel caso in cui le credenziali inserite siano errate.	US1
FMR3	L'utente non autenticato deve poter inserire una <i>email</i> e una <i>password</i> per potersi registrare.	US2
FMR4	L'utente non autenticato che ha inserito <i>email</i> e <i>password</i> nel <i>form</i> di registrazione deve ricevere un codice via <i>email</i> per poter attivare l' <i>account</i> .	US2
FMR5	L'utente non autenticato che si sta registrando e ha inserito con successo il codice ricevuto via <i>email</i> deve poter accedere con le credenziali inserite.	US2
FMR6	L'utente che ha creato un <i>account</i> deve poter avviare la procedura di recupero <i>password</i> .	US3
FMR7	L'utente che ha creato un profilo deve poter reimpostare la propria <i>password</i> .	US3
FMR8	L'utente autenticato deve poter eseguire il <i>logout</i> .	US4
FMR9	L'utente amministratore deve poter accedere al pannello amministratore.	US5
FMR10	L'utente amministratore deve poter visualizzare una lista degli <i>account</i> registrati sulla piattaforma.	US5
FMR11	L'utente amministratore deve poter disattivare e riattivare gli <i>account</i> .	US5

Requisito	Descrizione	Fonti
FMR12	L'utente autenticato deve poter creare un progetto vuoto, scegliendo un linguaggio tra Python, TypeScript e Java.	US6
FMR13	L'utente autenticato deve poter caricare il codice sorgente di un progetto.	US6
FMR14	L'utente autenticato deve poter caricare più volte i sorgenti del progetto.	US7
FMR15	L'utente autenticato deve poter modificare il nome del progetto.	US7
FMR16	L'utente autenticato deve poter modificare il linguaggio del progetto.	US7
FMR17	L'utente autenticato deve poter cancellare un progetto.	US8
FMR18	L'utente autenticato deve poter visualizzare la lista di tutti i progetti che ha creato.	US9
FMR19	Tutti gli utenti devono poter scaricare il programma per caricare il codice sorgente.	US10
FMR20	L'utente autenticato deve poter far generare la documentazione.	US11
FMR21	L'utente autenticato deve poter far generare più volte la documentazione.	US11
FMR22	L'utente autenticato deve poter consultare la documentazione generata.	US12
FMR23	L'utente autenticato deve poter consultare tutte le versioni di documentazione che ha generato.	US11, US12

Requisito	Descrizione	Fonti
FMR24	L'utente autenticato deve poter esportare la documentazione generata in formato PDF.	US13
FOR1	L'utente autenticato deve poter personalizzare il <i>template</i> della documentazione generata.	US14
FOR2	L'utente autenticato deve poter condividere la documentazione generata attraverso un <i>link</i> .	US15
FMR25	L'utente deve poter consultare il manuale che spiega come usare l'applicativo.	US16
FMR26	L'utente deve poter consultare il manuale che spiega come utilizzare la <i>utility</i> a riga di comando per caricare i progetti.	US16
FMR27	L'utente deve poter accedere alla pagina principale del sito che ne illustra le funzionalità.	US17

Tabella 3.2: Requisiti funzionali.

Requisito	Descrizione	Fonti
QMR1	L'applicazione deve garantire la compatibilità <i>cross-browser</i> .	Piano di lavoro.
QMR2	Fornire il codice sorgente prodotto attraverso un sistema di versionamento.	Piano di lavoro.
QMR3	Documentare nella specifica tecnica le criticità e i limiti delle soluzioni individuate.	Piano di lavoro.
QMR4	Documentare nella specifica tecnica le classi attraverso diagrammi UML.	Piano di lavoro.

Requisito	Descrizione	Fonti
QMR5	Redigere il manuale che descrive la configurazione dell'ambiente di produzione.	Piano di lavoro.
QMR6	Le API _G devono essere documentate in formato Swagger.	Riunione col <i>tutor</i> .
QDR1	L'applicazione deve poter essere fruibile anche su dispositivi <i>mobile</i> .	Piano di lavoro.
QOR1	Il sistema quando avvengono nuove generazioni di un progetto aggiorna solo le parti modificate del codice sorgente.	Piano di lavoro.

Tabella 3.3: Requisiti di qualità.

Requisito	Descrizione	Fonti
CMR1	L'applicazione deve essere sviluppata utilizzando i <i>container</i> in modo da essere facilmente rilasciabile in <i>cloud</i> .	Riunione col <i>tutor</i> .
CMR2	Il <i>frontend</i> dell'applicazione deve essere scritto usando la libreria React con linguaggio TypeScript.	Riunione col <i>tutor</i> .
CMR3	I servizi del <i>backend</i> devono essere realizzati usando il <i>framework</i> NestJS con linguaggio TypeScript.	Riunione col <i>tutor</i> .
CMR4	Le parti del sistema devono comunicare tra di loro attraverso API _G che usano il protocollo HTTP contenenti oggetti in formato JSON.	Decisione interna.
CMR5	La documentazione deve essere salvata in formato Markdown.	Piano di lavoro.

Requisito	Descrizione	Fonti
CMR6	In ambiente di produzione è possibile utilizzare solo i servizi <i>cloud</i> forniti da AWS _G .	Piano di lavoro.
CMR7	Il database utilizzato deve essere MongoDB.	Riunione col <i>tutor</i> .

Tabella 3.4: Requisiti di vincolo.

3.1.3. Analisi dei modelli

Il componente alla base del progetto, che permette la generazione della documentazione, è un LLM_G. Di conseguenza abbiamo svolto un'analisi approfondita riguardo i modelli forniti dal servizio Bedrock di AWS_G. Per testare i modelli abbiamo utilizzato dei progetti *software* (di grandi e medie dimensioni) scritti nei linguaggi Java, TypeScript e C.

Dopo aver reperito i progetti da GitHub per sottoporli ai *test* abbiamo definito delle metriche di valutazione, ovvero:

1. Velocità di generazione. Ritenuta eccessiva sopra al minuto di tempo.
2. Qualità della documentazione prodotta. Valutata direttamente da noi secondo i nostri canoni di qualità, comprensibilità e completezza.
3. Capacità di rispettare il formato specificato nel *prompt*.
4. Costo della generazione.

Tabella 3.5 mostra i risultati ottenuti dalla nostra analisi. La colonna costo indica il costo in dollari per un milione di *token* in *output* sul servizio Bedrock. La colonna qualità può avere i seguenti valori (dal migliore al peggiore): «Ottima», «Buona», «Mediocre», «Pessima».

Abbiamo scelto il valore in funzione di queste specifiche:

- Ottima: la documentazione contiene tutte le classi, funzioni e strutture dei sorgenti, spiegandole in modo comprensibile senza escludere dettagli importanti. Quando necessario deve fornire degli esempi di codice se aiuta la comprensione.
- Buona: come la precedente ma senza fornire esempi.
- Mediocre: come la precedente ma tralasciando qualche dettaglio importante.
- Pessima: come la precedente ma incompleta (la documentazione non contiene tutte le classi, funzioni e strutture).

Modello	Velocità	Prompt	Costo	Qualità
Sonnet 3.5	~1m	Conforme	15\$	Ottima
Sonnet 3.7	~2m	Conforme	15\$	Ottima
Llama 3.1 1B	>5m	Conforme	10\$	Mediocre
Llama 3.2 3B	>5m	Conforme	12\$	Mediocre
Pixtral Large	>5m (a volte non termina)	Non conforme	8\$	Pessima
Nova Lite	20s	Conforme	0,10\$	Mediocre
Nova Pro	20s	Conforme	0,20\$	Buona

Tabella 3.5: Risultati analisi dei modelli.

3.2. Progettazione

Dopo il processo di analisi siamo passati alla progettazione delle varie componenti del sistema. Precisamente il sistema è composto dalle seguenti componenti architettoniche:

- **Backend:** applicazione lato *server* che gestisce le richieste esterne e le processa. Qui è contenuta la tutta la logica di *business* dell'applicativo.
- **Frontend:** interfaccia utente accessibile tramite *browser* che comunica attraverso richieste HTTP con la componente *backend*.
- **CLI:** piccolo *script* eseguibile tramite terminale che permette di caricare i *file* dei progetti comunicando direttamente con il *backend*.
- **Database:** componente utilizzato per la persistenza dei dati degli utenti, con relativi progetti e documentazione generata.

3.2.1. Backend

Il *backend* è composto da tre microservizi scritti in TypeScript utilizzando il *framework* NestJS. Abbiamo progettato il tutto attraverso diagrammi delle classi UML. I microservizi sono tre, ovvero *Projects*, *Documentation* e *Users*. Sono stati usati dei microservizi in modo da poter dividere ogni «dominio» in un'applicazione differente, ottenendo isolamento e scalabilità.

Tutti i microservizi sono accessibili solo da utenti che si sono autenticati. L'autenticazione utente viene gestita dal servizio Cognito di AWS_G, tramite l'interfaccia del *frontend*. Tutti gli *endpoint* accettano un JWT_G, ovvero un *token* che contiene info varie sull'utente ed è firmato attraverso un sistema di chiavi, per cui si può verificare la validità e autenticità.

Microservizio *Projects*

Questo microservizio si occupa della gestione dei progetti degli utenti. Codice 3.4 mostra com'è composta la struttura dati principale di cui si occupa, ovvero il progetto.

```
1 class Project { typescript
2   id: string;
3   name: string;
4   language: Language;
5   creationDate: Date;
6   sourceAvailable: boolean;
7 }
```

Codice 3.4: Classe *Project*.

Partendo dal progetto sono state associate ad esso le seguenti operazioni, ognuna associata ad un *endpoint* HTTP:

- (POST /project/v1) Creazione di un nuovo progetto.
- (PUT /project/v1/{projectId}) Modifica degli attributi di un progetto.
- (DELETE /project/v1/{projectId}) Cancellazione di un progetto e della documentazione ad esso associata.
- (GET /project/v1/{projectId}) Ottenere gli attributi di un singolo progetto.
- (GET /project/v1/) Ottenere la lista dei progetti.
- (GET /project/v1/{projectId}/uploadUrl) Ottenere il *link* del servizio S3 di AWS_G per poter caricare il codice sorgente di un progetto.
- (PUT /project/v1/{projectId}/uploadCallback) Segnalare che si è caricato il codice sorgente con successo.

Inoltre sono presenti delle comunicazioni interne attraverso Redis con il *pattern publisher - subscriber*. Questa interfaccia è esclusivamente ad uso interno per gli altri microservizi, infatti le comunicazioni avvengono in una rete interna senza utilizzo di *token* di autenticazione. I messaggi che accetta questo microservizio sono:

- (get_project_source) Restituisce il *link* del servizio S3 di AWS_G con cui è possibile ottenere il codice sorgente di un progetto.
- (get_project_info) Restituisce gli attributi di un progetto.

Microservizio *Documentation*

Questo microservizio si occupa della gestione e generazione della documentazione dei progetti di un utente. Codice 3.5 mostra la struttura dati principale, la documentazione.

```
1 class Documentation { typescript
2   readonly id: string;
3   readonly creationDate: Date;
4 }
```

Codice 3.5: Classe *Documentation*.

Supporta le seguenti operazioni sulla documentazione:

- (POST /documentation/v1/generate) Generare la documentazione di un dato progetto.
- (GET /documentation/v1/{projectId}) Ottenere la lista delle varie documentazioni generate nel tempo.
- (GET /documentation/v1/url/{documentationId}) Ottenere il *link* del servizio S3 di AWS_G in cui è possibile scaricare la documentazione in formato *markdown*.
- (GET /documentation/v1/export/{documentationId}) Ottenere la documentazione in formato PDF.

Anche questo microservizio accetta messaggi da altri microservizi tramite Redis:

- (delete_all_documentation) Cancella tutta la documentazione associata ad un progetto. Questo *endpoint* è ovviamente utilizzato da *Project* nel caso in cui un progetto venga cancellato.

Microservizio *Users*

Questo microservizio si occupa della gestione degli utenti. Codice 3.6 mostra la struttura dati principale, l'utente.

```
1 class User { typescript
2   readonly id: string;
3   readonly email: string;
4   readonly name: string;
5   readonly surname: string;
6   readonly isDisabled: boolean;
7   readonly isAdmin: boolean;
8 }
```

Codice 3.6: Classe *User*.

Supporta le seguenti operazioni:

- (PUT /users/v1/{userId}/disable) Disabilitare un utente. Dopo questa operazione non sarà più in grado di accedere all'applicativo.
- (PUT /users/v1/{userId}/enable) Abilitare un utente.
- (GET /users/v1) Ottenere la lista degli utenti.
- (POST /users/v1/registerCallback) Comunicare al *backend* che è stato registrato un nuovo utente attraverso il servizio Cognito di AWS_G.

Tutti gli *endpoint* sono accessibili solo agli utenti *admin*, escluso quello per la registrazione di un nuovo utente.

Architettura esagonale

Ogni microservizio è strutturato utilizzando lo stesso *pattern* architetturale, ovvero l'architettura esagonale. Questa architettura ha l'obiettivo di minimizzare le dipendenze fra le componenti, attraverso l'uso di interfacce (chiamate porte) e *adapters*. Tutto questo serve a rendere il *software* più manutenibile, scalabile e facilmente testabile.

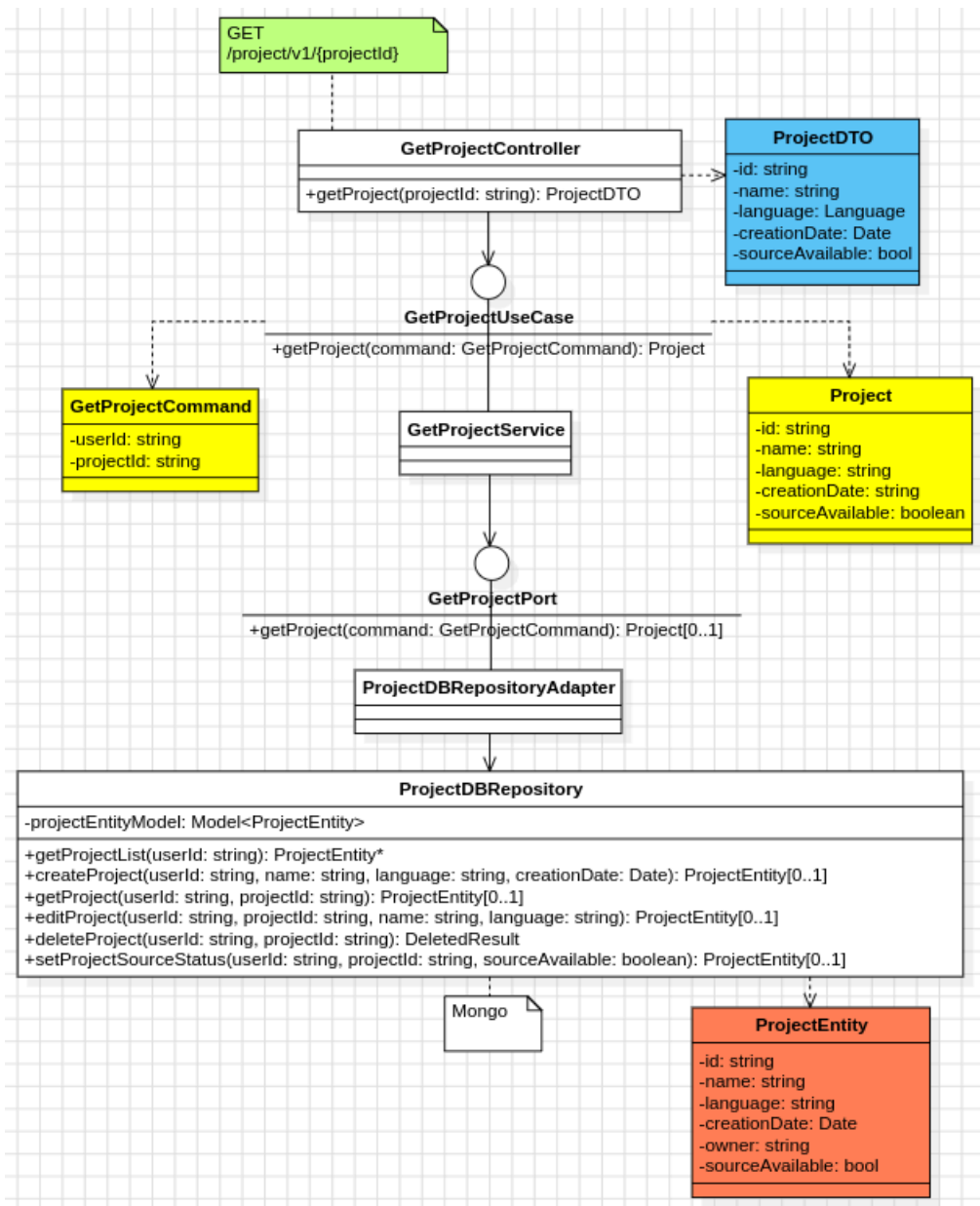


Figura 3.13: Esempio di *endpoint* in architettura esagonale.

Fonte: Specifica tecnica.

Figura 3.13 mostra uno degli *endpoint* del microservizio *Project*, precisamente quello che si occupa di ottenere le informazioni di un progetto dato il suo *id*. L'architettura suddivide le classi in tre *layer* logici:

1. *Application logic*.
2. *Business logic*.
3. *Persistence logic*.

Nell'esempio la classe `GetProjectController` e le classi in blu appartengono all'*application logic*, che si occupa di validare i dati che arrivano dall'utente attraverso richieste del *frontend*.

La classe `GetProjectService`, insieme alle due interfacce `GetProjectUseCase`, `GetProjectPort` e alle classi in giallo, appartengono la *business logic*. Ad essa arrivano i dati validati e con essi eseguono un'operazione particolare, in questo caso riprendere i dati del progetto (con tutti i controlli del caso).

Le classi `ProjectDBRepositoryAdapter` e `ProjectDBRepository`, insieme alle classi bordò appartengono alla *persistence logic*, che si occupa di comunicare con servizi esterni (in questo caso il *database*).

La peculiarità dell'architettura esagonale è proprio nell'uso di queste interfacce, che permettono di disaccoppiare la *business logic* dalle altre due. Le interfacce definiscono due tipologie di porte, di input e di output. Le porte di *input* (`GetProjectUseCase`) definiscono una funzionalità di *business*, mentre le porte di *output* (`GetProjectPort`) definiscono un servizio esterno, permettendo alla *business logic* di «comunicare» con la *persistence*.

Tutto quello che vive in *business logic* non deve essere usato nelle altre due, in modo da non creare dipendenze. Da questo si nota l'utilizzo del *design pattern Adapter*. Il *pattern* è implementato da `GetProjectController` per trasformare i dati di *application* in *business* (e viceversa) e da `ProjectDBRepositoryAdapter` per i dati di *persistence* in *business* (e viceversa).

3.2.2. Frontend

Abbiamo progettato il *frontend* utilizzando Figma, un *software* che permette di creare *design* di pagine *web*. Tutte le pagine sono implementate utilizzando la libreria React.

Dalla progettazione abbiamo ottenuto le seguenti pagine:

- *Home* - pagina di benvenuto.
- *Auth* - pagine che gestiscono le procedure di *login*, *logout* e registrazione.
- *Dashboard* - pagina principale con tutte le funzionalità per la gestione dei progetti e della documentazione.
- *Manuale* - pagina che mostra il manuale utente.
- *Admin* - pagina riservata per la gestione degli utenti.

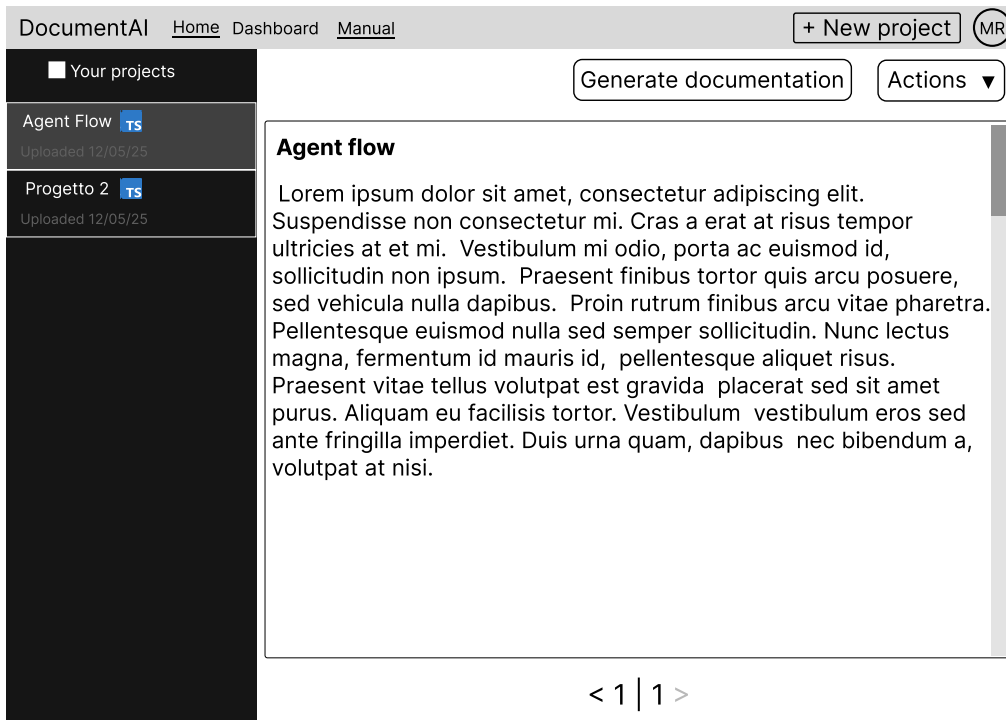


Figura 3.14: *Design* della pagina *Dashboard*.

Fonte: Autore.

Figura 3.14 mostra il *design* della pagina *Dashboard* su Figma. Nel centro dello schermo viene visualizzata la documentazione generata, mentre sul pannello laterale a sinistra si possono selezionare i progetti. Dai pulsanti in alto si possono effettuare tutte le operazioni, come generazione della documentazione, modifica, cancellazione di un progetto e così via. Questa è la pagina più complessa dell'applicazione, visto che gestisce tutte le interazioni con i progetti e la documentazione.

Infine passando all'implementazione sono avvenuti dei ritocchi finali che non hanno intaccato molto quanto progettato.

3.2.3. CLI

La terza componente è chiamata CLI (*Command line interface*) e si occupa di caricare i *file* sorgenti di un progetto verso lo *storage* S3 di AWS_G. La componente è scritta in Python con una singola dipendenza (`request` per poter eseguire richieste HTTP), in modo da essere facilmente utilizzabile su tutti i maggiori sistemi operativi.

Dal punto di vista utente la CLI è molto facile da utilizzare, visto che ne viene descritto l'uso nel *frontend*. Infatti lo *script* si può scaricare direttamente da esso e quando si decide di caricare i sorgenti di un progetto viene fornito in automatico il comando da inserire nel terminale. Figura 3.15 mostra il *design* Figma di questa funzionalità.

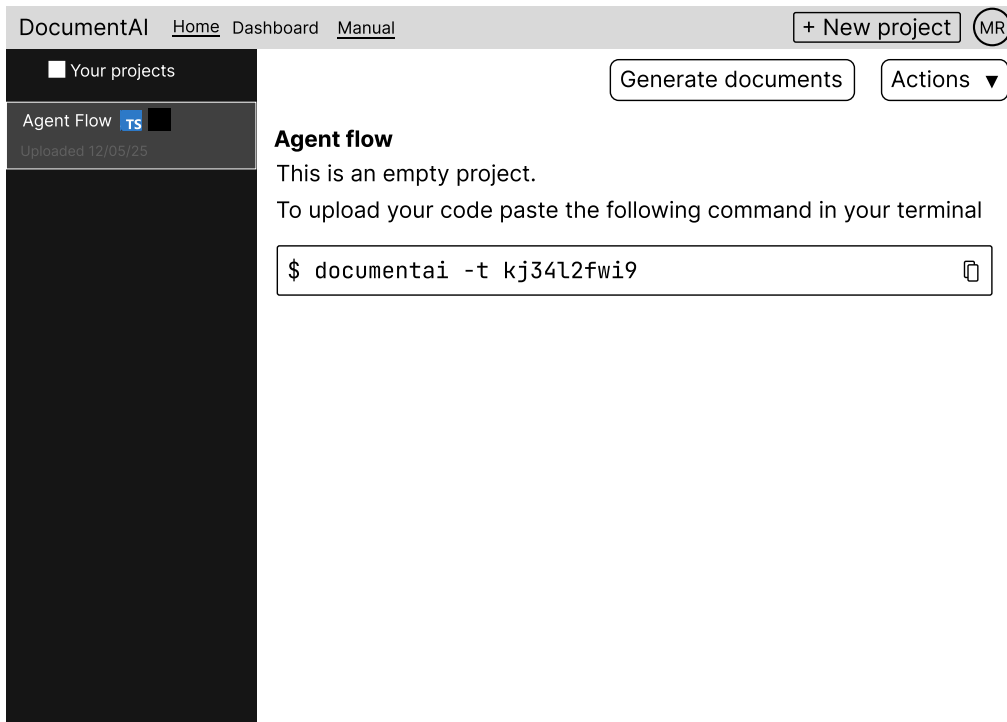


Figura 3.15: *Dashboard* - comando per il caricamento dei sorgenti.

Fonte: Autore.

3.2.4. *Database*

Per la persistenza dei dati abbiamo utilizzato MongoDB, un *database* non relazionale che organizza i dati in documenti simili a JSON.

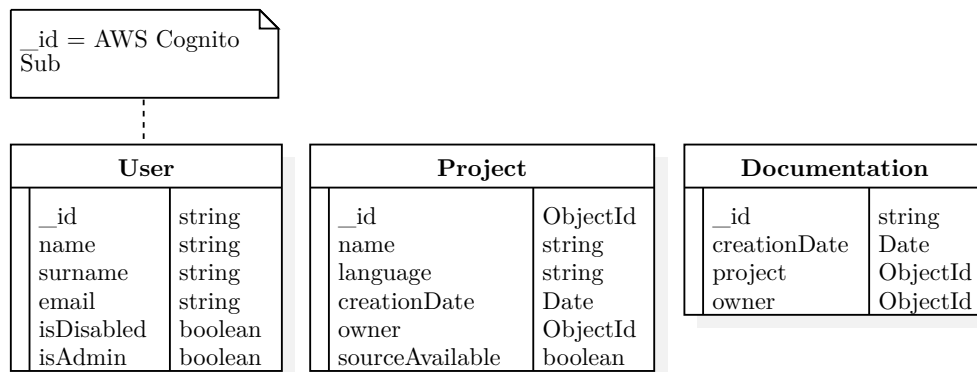


Figura 3.16: Diagramma UML del *database*.

Fonte: Specifica tecnica.

Figura 3.16 mostra le tre collezioni che abbiamo ottenuto dalla progettazione. Visto che il *backend* è composto da microservizi indipendenti (tralasciando qualche dipendenza sotto forma di messaggi interni) abbiamo cercato di rendere indipendenti anche le collezioni che accedono. Di fatto (come sarà mostrato nell'ambiente di produzione) si potrebbero creare tre *database* indipendenti, uno per ogni collezione, esclusivi per microservizio.

3.2.5. Configurazione dell'ambiente di produzione

Figura 3.17 mostra come sono configurate le componenti e come dialogano fra loro nell'ambiente di produzione. Ogni componente è all'interno di un riquadro, indicando che è un'entità indipendente. Un arco che collega due componenti indica che c'è un dialogo fra le due, attraverso richieste HTTP a specifici *endpoint*. Gli archi di colore giallo indicano che la comunicazione utilizza lo scambio di messaggi tramite Redis. Inoltre gli archi sono orientati, indicando qual è la componente che effettua la prima richiesta.

Partendo da sinistra sono presenti gli strumenti più vicini all'utente:

- CLI, che si occupa di caricare il codice sorgente comunicando con il microservizio *Projects*.
- *Frontend*, che comunica con tutti i microservizi e con AWS_G cognito per gestire l'autenticazione.

Continuando sono presenti i microservizi del *backend* con i servizi di AWS_G:

- *Projects*, che ha a disposizione il *database* e lo storage di S3 per salvare dati e *file* dei progetti.
- *Documentation*, che ha a disposizione il *database*, lo storage di S3 e può accedere agli LLM_G di Bedrock per generare la documentazione.

- *Users*, che ha a disposizione il *database* per salvare le informazioni degli utenti.

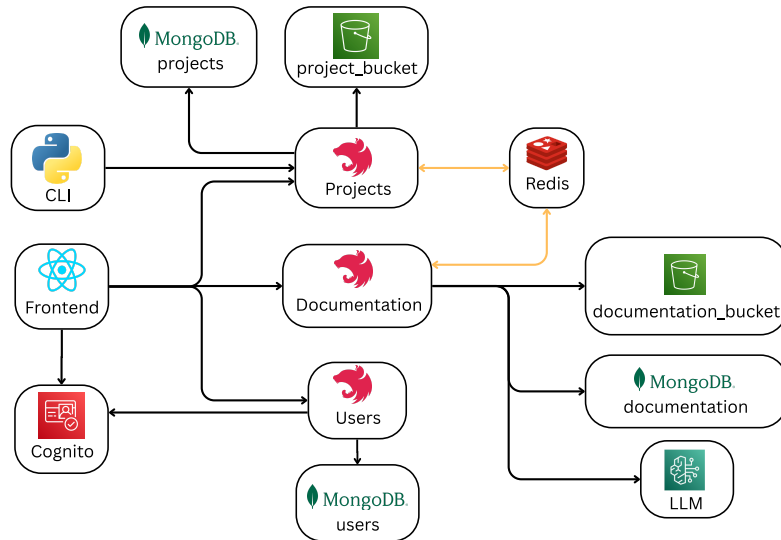


Figura 3.17: Configurazione ambiente di produzione.

Fonte: Specifica tecnica.

3.3. Implementazione

Passando all'implementazione abbiamo deciso di utilizzare una precisa gestione dei *branch* di Git. Esistono due *branch* principali, *main* e *dev*, nel primo si va in produzione e nel secondo si sviluppa attivamente. Quando si vuole aggiungere una funzionalità si crea un nuovo *branch* a partire da *dev* (chiamato *branch* di *feature*). Finita l'implementazione della *feature* si apre una *pull request* per fare il *merge* verso *dev*. Periodicamente si possono fare delle *pull request* da *dev* verso *main* per mandare il codice in produzione, ovviamente assicurandosi che il tutto sia corretto e conforme attraverso i *test*.

I nostri *step* per implementare una *feature* sono i seguenti:

1. Implementazione della funzionalità, seguendo la progettazione, lo stile, le convenzioni scelte e le regole definite per la gestione dei *branch*.
2. Implementazione dei *test* di unità, cercando di coprire tutti i casi particolari nel codice.
3. Processo di *code review* su GitHub.

Metriche

Tabella 3.6 mostra l'entità di codice prodotto per ogni componente architetturale.

Componente	Linee di codice	<i>Code coverage (unit test)</i>
<i>Frontend</i>	2517	97%
CLI	65	100%
<i>Projects</i>	3690	98%
<i>Documentation</i>	3597	98%
<i>Users</i>	1806	100%

Tabella 3.6: Metriche del prodotto.

Dettagli implementativi interessanti

Nel corso dell'implementazione è andato quasi tutto liscio tranne un piccolo problema riguardante le richieste verso gli LLM_G su Bedrock. Nello specifico c'è un limite massimo di richieste che possono avvenire in un dato istante e questo ha creato non pochi problemi inizialmente, visto che avrebbe reso impossibile generare documentazione per grandi quantità di *file* sorgenti. Inizialmente per generare la documentazione veniva preso un file sorgente per volta e inviato tramite una richiesta HTTP al modello su Bedrock, questa operazione veniva fatta in parallelo per ottimizzare i tempi di attesa.

```

1  const prompt = "<PROMPT>";
2  const aiModel = "<MODEL>";
3
4  const res: string[] = [];
5
6  res.push((await Promise.all(
7    project_files.map(async (file) => {
8      const [filePath, fileContent] = file;
9      const response = await this.llmInvokePort.llminvoke(
10         prompt,
11         filePath,
12         fileContent,
13         aiModel);
14      if (response === null)
15         throw new LLMGenError();
16
17      return response;
18    }))).join('\n\n'));

```

Codice 3.7: Generazione della documentazione in parallelo.

Codice 3.7 mostra l'esatta porzione di codice che andava a causare il problema. Precisamente si inseriscono dentro un *array* tutte le risposte del modello in parallelo, questo grazie all'uso del metodo statico `all` di `Promise`. Poi attraverso il metodo `map` viene inviata una richiesta (che avviene attraverso il metodo `llminvoke`) per ogni *file*.

Abbiamo risolto il problema limitando le richieste, inviandole a gruppi di n (in questo caso 5). Codice 3.8 mostra l'implementazione precisa.

```

1  const prompt = "<PROMPT>";
2  const aiModel = "<MODEL>";
3
4  const res: string[] = [];
5
6  const GROUP_SIZE = 5;
7  const projectFilesGroup = Array.from({ length:
8    Math.ceil(projectFiles.length / GROUP_SIZE) },
9    (_, index) =>
10     projectFiles.slice(index * GROUP_SIZE, (index + 1) * GROUP_SIZE));
11
12 for (const files of projectFilesGroup) {
13   res.push(await Promise.all(
14     files.map(async (file) => {
15       const [filePath, fileContent] = file;
16       const response = await this.llmInvokePort.llmInvoke(
17         prompt,
18         filePath,
19         fileContent,
20         aiModel);
21       if (response === null)
22         throw new LLMGenError();
23       return response;
24     }))).join('\n\n');
25 }

```

Codice 3.8: Generazione della documentazione in parallelo (gruppi di 5).

3.4. Verifica e collaudo

Verifica

Nel processo di verifica abbiamo adottato diverse tipologie di *test*, rispettivamente:

- *Test* di unità, che verificano il comportamento delle singole classi e metodi (nel caso della programmazione ad oggetti).
- *Test* di integrazione, che verificano la corretta comunicazione fra le diverse componenti.
- *Test* di sistema, che verificano interamente le funzionalità del prodotto dal punto di vista utente.

Come detto precedentemente, abbiamo cercato di ottenere una buona qualità del codice attraverso *test* di unità scritti al momento dell'implementazione e con *review* del codice verificando riga per riga che fosse tutto corretto e ben scritto.

La gran parte dei *test* implementati è di unità, con i quali abbiamo ottenuto una copertura del codice del 98%. Oltre a questi abbiamo anche implementato altri *test* di sistema e integrazione per le parti più critiche, ovvero:

- Generazione della documentazione.
- Esportazione della documentazione in PDF.
- Cancellazione di un progetto.

Abbiamo implementato la maggior parte dei *test* attraverso il *framework* Jest. Esso permette di definire delle *suite* di *test* e di controllare la copertura totale nei *file* sorgenti. Codice 3.9 mostra un esempio di un *test* di unità preso direttamente dal codice sorgente del microservizio *Users* del *backend*.

```
1 describe('getUserList', () => { typescript
2   it('should return a list of registered users', async () => {
3     getUserListPortMock.getUserList.mockResolvedValue(userListMock);
4     getUserPortMock.getUser.mockResolvedValue(userMock);
5     expect(await
6       getUserListService.getUserList('adminId')).toEqual(userListMock);
7   });
8   it('should throw a UserNotFoundError if the user does not exists', ()
9     => {
10    getUserPortMock.getUser.mockResolvedValue(null);
11    const result = getUserListService.getUserList(
12      'adminId',
13    );
14    void expect(result).rejects.toThrow(UserNotFoundError);
15  });
16 }
```

Codice 3.9: Esempio di *test* utilizzando Jest.

Resoconto copertura dei requisiti

Tabella 3.7 mostra il grado di copertura dei requisiti. Di fatto abbiamo soddisfatto pienamente tutti i requisiti obbligatori ma tralasciando completamente (i pochi) requisiti desiderabili e opzionali.

Tipo	Funzionali	Qualità	Vincolo
<i>Mandatory</i>	100%	100%	100%
<i>Desirable</i>	Non presenti	0%	Non presenti
<i>Optional</i>	0%	0%	Non presenti

Tabella 3.7: Riepilogo copertura dei requisiti.

Collaudo finale e presentazione

Al termine delle otto settimane di *stage* il nostro *tutor* aziendale ha organizzato un incontro con tutti i membri dell'azienda per il collaudo finale e la presentazione del prodotto, permettendoci di mostrare i risultati ottenuti e verificare la validità del nostro lavoro. Abbiamo diviso l'incontro in tre parti:

1. Idea del progetto e obiettivi.
2. Analisi e progettazione, ponendo enfasi sui modelli utilizzati e sull'architettura utilizzata.
3. Dimostrazione pratica, mostrando pregi e difetti della soluzione.

Abbiamo dedicato la maggior parte del tempo alla dimostrazione pratica, in cui abbiamo discusso sul come utilizzare al meglio il prodotto e delle scelte di *design* riguardo l'esperienza utente e *performance*. Successivamente abbiamo mostrato come eseguire il *software* in locale partendo da zero (con l'ausilio del file `README.md`, in cui abbiamo documentato tutto il processo in modo preciso) in modo da permettere a tutti di utilizzarlo per comprenderlo più a fondo. Alla fine della presentazione abbiamo dedicato qualche minuto finale per eventuali dubbi e confronti con i partecipanti.

L'incontro ha avuto un buon impatto ed i partecipanti hanno dimostrato interesse, soprattutto ponendo domande riguardo al processo con cui abbiamo implementato la generazione della documentazione e sulla scelta dei modelli.

3.5. Prodotto finale

Il prodotto finale si presenta abbastanza bene dal punto di vista estetico ed è abbastanza intuitivo anche per un utente novizio. Appena l'utente si collega tramite *browser* all'applicativo vedrà come prima cosa la pagina *Home*. Figura 3.18 mostra come si presenta esteticamente. In questa pagina viene presentato il *software* attraverso una semplice descrizione. In tutte le pagine è

sempre presente in alto una barra di navigazione in modo da potersi orientare facilmente.

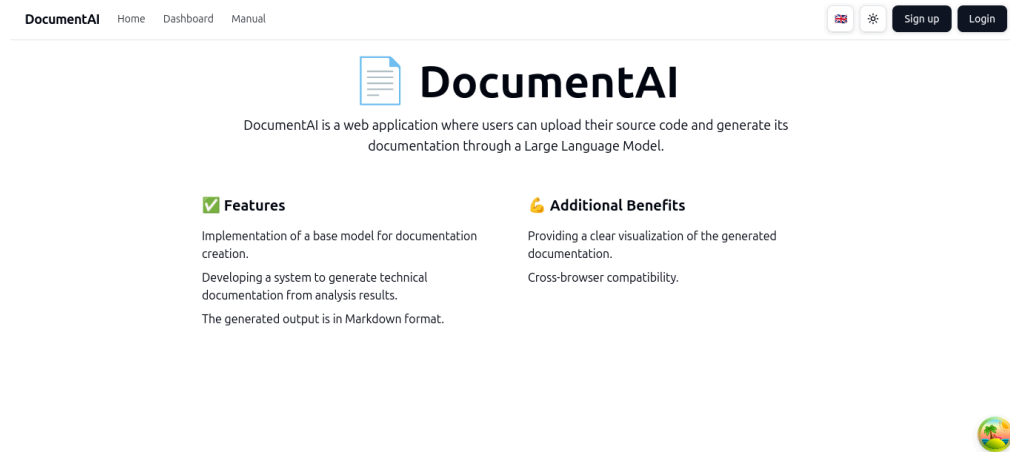


Figura 3.18: Pagina *Home* del prodotto.

Fonte: Autore.

Dopo aver visto la pagina *Home* è necessario autenticarsi per poter accedere alle funzionalità principali del *software*. Per questo scopo sono presenti le pagine di autenticazione, quindi *login* e *sign up*. Figura 3.19 mostra la pagina di *login*.

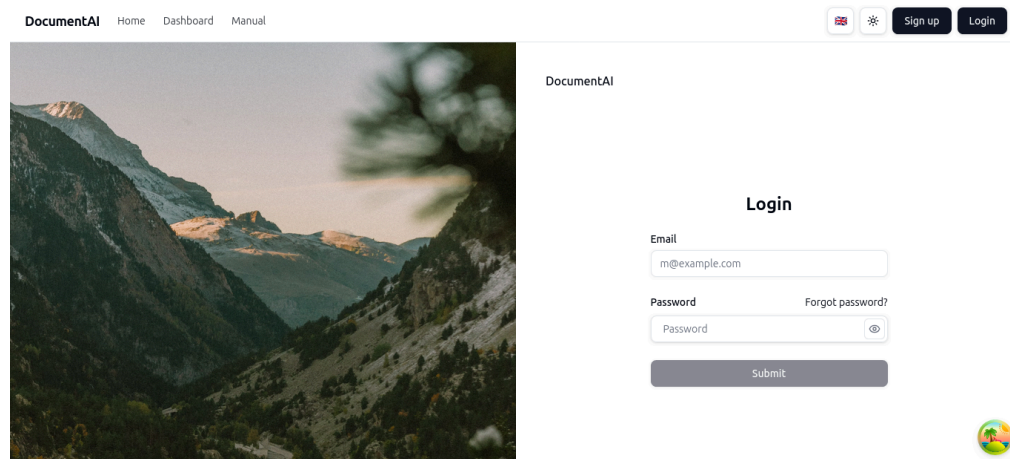


Figura 3.19: Pagina *Auth* del prodotto.

Fonte: Autore.

Dopo l'autenticazione si può utilizzare concretamente il software. Figura 3.20 mostra la pagina più importante, la pagina *Dashboard*, in cui avviene la maggior parte delle azioni sui progetti e sulla documentazione generata.

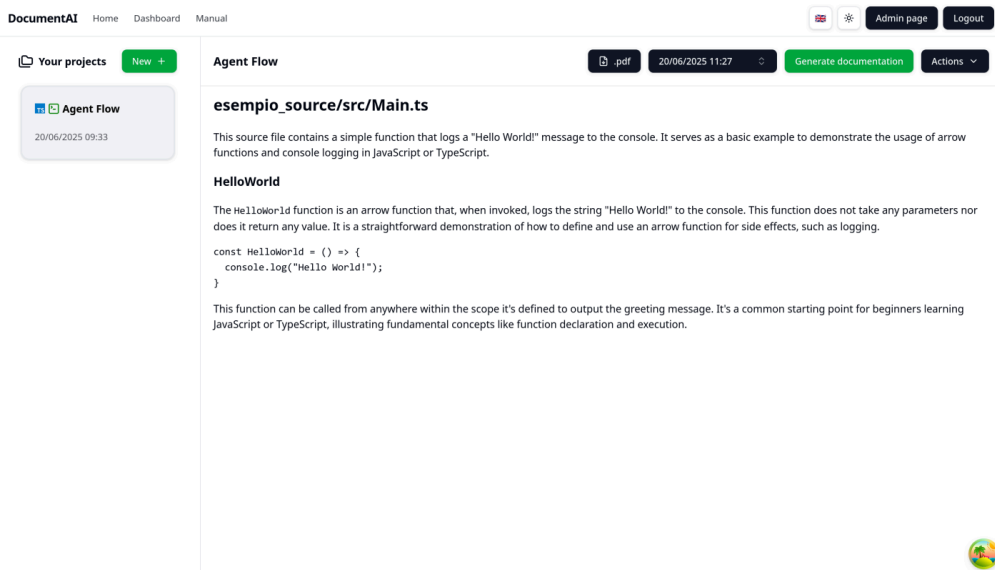


Figura 3.20: Pagina *Dashboard* del prodotto.

Fonte: Manuale utente.

Limiti del prodotto

Il prodotto funziona abbastanza bene nella generazione di documentazione di progetti sia di piccoli che grandi dimensioni (in tempi abbastanza brevi). La più grande limitazione del prodotto riguarda il modo in cui viene generata, ovvero utilizzando *file* per volta in modo indipendente. Questo permette di parallelizzare le richieste e quindi essere veloci nella generazione. Inoltre con software molto grandi è l'unico modo perchè il modello non ha un contesto di *token* abbastanza grande da poter contenere tutto il testo dei sorgenti. Questa scelta porta ad un problema non sempre trascurabile, ovvero che il modello potrebbe non cogliere dei dettagli importanti visto che non ha una vera visione di insieme.

Documentazione

Oltre al prodotto *software* abbiamo prodotto della documentazione a sostegno di esso. Precisamente i documenti sono:

- Specifica tecnica, in cui sono descritte le scelte implementative, *pattern* e tutto quello che serve per poter comprendere al meglio il funzionamento interno del *software*.
- Manuale utente, in cui viene presentato il *software* in maniera semplice ed intuitiva, in modo da poter essere usato al meglio.
- *API_G reference*, che riassume tutte le classi e i metodi all'interno del codice, generata attraverso strumenti automatici.

Capitolo 4.

Retrospettiva

4.1. Risultati raggiunti

4.1.1. Obiettivi aziendali

Tabella 4.8 mostra i risultati riguardo gli obiettivi aziendali presentati in Tabella 2.1. Abbiamo raggiunto tutti gli obiettivi obbligatori (OB1 - OB5). Inoltre abbiamo raggiunto tutti gli obiettivi opzionali tranne OB7 ed OB9.

Obiettivo	Descrizione	Esito
OB1	Supporto di tre linguaggi.	Supportati pienamente 3 linguaggi, ovvero TypeScript, Python e Java.
OB2	Gestione di molti <i>file</i> .	Supportata la capacità di gestire grandi quantità di <i>file</i> e di processarli in parallelo per ottenere una generazione veloce.
OB3	Formato standard per la documentazione.	Tutta la documentazione viene generata seguendo la sintassi Markdown.
OB4	Interfaccia utente funzionale.	Abbiamo progettato una interfaccia facile da comprendere e funzionale, attraverso l'uso di poche opzioni intuitive e aiutando l'utente attraverso il manuale.

Obiettivo	Descrizione	Esito
OB5	Compatibilità sui maggiori <i>browser</i> .	L'applicativo funziona correttamente sui maggiori <i>browser</i> , precisamente Mozilla Firefox e tutti i <i>browser</i> derivati da Chromium.
OB6	Supportare più di tre linguaggi.	Oltre ai 3 linguaggi supportati, l'applicativo è in grado di gestire la maggior parte dei linguaggi <i>mainstream</i> .
OB7	Utilizzo collaborativo tramite <i>team</i> .	Non realizzato. È possibile introdurre la funzionalità senza grossi problemi visto che in fase di progettazione si sono divisi appositamente progetti ed utenti.
OB8	Autenticazione utenti.	Per utilizzare l'applicativo è necessario autenticarsi. L'autenticazione è gestita utilizzando il servizio Cognito di AWS _G .
OB9	Condivisione della documentazione.	Non realizzato. Non è possibile condividere la documentazione tramite <i>link</i> però è comunque possibile esportarla in PDF e condividerla attraverso servizi esterni.
OB10	Ottimizzare prestazioni nei grandi progetti.	L'applicativo gestisce bene i grandi progetti in modo veloce grazie alla parallelizzazione in fase di generazione della documentazione. Tut-

Obiettivo	Descrizione	Esito
		tavia è sicuramente possibile ridurre i tempi con tecniche non implementate, come generazione in <i>background</i> e l'unione di <i>file</i> piccoli.

Tabella 4.8: Risultati obiettivi aziendali.

4.1.2. Obiettivi personali

Servizi di AWS_G

- Descrizione obiettivo: comprendere quanto più possibile di come funziona AWS_G ed i suoi servizi, in modo da poterli utilizzare al meglio per ottenere una buona architettura.
- Stato: **soddisfatto**.

Ho compreso abbastanza bene come orientarmi nell'ecosistema *cloud* di AWS_G utilizzando spesso la documentazione ufficiale. Ho imparato ad utilizzare molti servizi utili come Cognito per l'autenticazione, Bedrock per l'uso degli LLM_G ed S3 per la persistenza di dati.

Strumenti in azienda

- Descrizione obiettivo: apprendere come vengono utilizzati certi strumenti in azienda, soprattutto la gestione dei *repository* su GitHub con le *monorepo*.
- Stato: **soddisfatto**.

Ho imparato a gestire un *repository* di tipo *monorepo* attraverso lo strumento Nx, che ho usato durante tutto il percorso. Oltre alla gestione dei *repository* ho compreso meglio il funzionamento di Jira.

Architettura a microservizi

- Descrizione obiettivo: Capire più a fondo com'è strutturata e come funziona un'architettura a microservizi.
- Stato: **soddisfatto**.

Ho sicuramente incrementato le mie conoscenze riguardo la progettazione e realizzazione di architetture orientate ai microservizi. Questo grazie al fatto che abbiamo sviluppato interamente il *backend* cercando di ottenere microservizi semplici e indipendenti.

4.2. Competenze acquisite

4.2.1. Tecnologie

Ho imparato ad usare tutte le maggiori tecnologie dello *stack* aziendale. Le più importanti sono:

- TypeScript, linguaggio di programmazione con cui abbiamo realizzato il tutto.
- React, libreria utilizzata per il *frontend*.
- NestJS, *framework* utilizzato per il *backend*.
- MongoDB, *database* non relazionale e orientato ai documenti per gestire la persistenza di ogni microservizio.
- Docker, per creare le immagini che poi andranno a generare i *container* in produzione.
- Servizi di AWS_G, precisamente Cognito, Bedrock ed S3.

4.2.2. Metodologie

Rispetto al processo di analisi ho capito meglio come vanno strutturate le User Story_G in modo da poter ottenere un buon risultato con dei requisiti non banali. Ad esempio ho imparato che hanno una struttura precisa che segue la forma: «Come {utente tipo}, voglio {un'azione}, in modo da {ottenere un beneficio}». Oltre alla prima frase strutturata in questo modo è buona norma mettere anche una lista dei criteri di accettazione, che permettono di accertare senza ambiguità il completamento della User Story_G.

Riguardo la progettazione ho appreso concetti nuovi soprattutto per realizzare i microservizi. Ad esempio:

- Gestire le comunicazioni tra i vari microservizi attraverso *broker* e code di messaggi (nel nostro caso utilizzando Redis).
- Ottenere degli *endpoint* semplici e che siano pensati per essere integrati facilmente con altre componenti.

Infine abbiamo imparato ad organizzarci e suddividere le *task* da svolgere nel modo corretto e rispettando i tempi previsti.

4.3. Prerequisiti fondamentali

4.3.1. Competenze chiave

Per riuscire a completare questo progetto in modo efficace è stato fondamentale conoscere alcune competenze specifiche che elencherò in seguito.

Lavorare in *team*

Ho già realizzato progetti in passato in *team* e questo mi ha aiutato molto nello svolgere quest'ultimo. Ritengo che questo prerequisito sia il più importante, perché per realizzare grandi progetti e per lavorare in industria è indispensabile essere grado di discutere, prendere decisioni e capire i problemi insieme ad altre persone.

Autoapprendimento

All'inizio di questo progetto non conoscevo assolutamente nulla di certe tecnologie e metodologie. In questo campo le tecnologie cambiano spesso, quindi è indispensabile sapersi adattare e non c'è sempre qualche esperto che può dare una mano. Di conseguenza saper apprendere da soli a volte è l'unica strada percorribile. Inoltre affinando questa pratica si diventa più professionali e indipendenti.

Strumenti di versionamento e *tracking*

Visto che il prerequisito più importante è saper lavorare in *team*, a seguire servono anche gli strumenti che possano agevolare questo processo. Conoscere e saper utilizzare un minimo gli strumenti di versionamento (VCS_G) e di *tracking* (ITS_G) è indispensabile per poter gestire la realizzazione di un progetto e lavorarci assieme in *team*.

4.3.2. Influenza del corso di studio

Il corso di studio mi ha formato abbastanza rispetto ai prerequisiti che ho citato. In primis ho imparato a lavorare in gruppo ed a relazionarmi con i miei colleghi grazie ai progetti che si svolgono nei corsi. Riguardo l'autoapprendimento ho sempre cercato di studiare argomenti e tecnologie che ritengo interessanti ancora prima di iniziare questo percorso universitario, però devo dire che grazie alle basi fornite dal corso riesco a comprendere meglio certi concetti. Inoltre, pur avendo acquisito in passato delle basi riguardo strumenti di versionamento e *tracking*, ho comunque approfondito certe tematiche sia attraverso dei corsi specifici e sia attraverso progetti di gruppo.

Per concludere confermo che questo percorso mi ha aiutato a crescere e ritengo che l'influenza più importante è stata quella dei rapporti che si sono andati a creare.

Glossario

Agile:

Insieme di metodi di sviluppo del *software* creati a partire dai primi anni 2000. Si segue un approccio meno strutturato e focalizzato sull'obiettivo di consegnare al cliente, in tempi brevi e frequentemente. È fondato su una serie di principi definiti nel Manifesto Agile ovvero:

1. Gli individui e le interazioni più che i processi e gli strumenti.
2. Il *software* funzionante più che la documentazione esaustiva.
3. La collaborazione col cliente più che la negoziazione dei contratti.
4. Rispondere al cambiamento più che seguire un piano. [1] 2.

API – Application Programming Interface:

Interfaccia che espone un insieme di procedure che permettono la comunicazione e lo scambio di dati fra componenti differenti. 1., 27., 45.

AWS – Amazon Web Services:

Insieme di servizi *cloud* forniti da Amazon. Offre servizi come macchine virtuali, *container* e *storage*. 1., 16., 28., 30., 31., 32., 35., 37., 47., 48., 49.

Azure:

Insieme di servizi *cloud* forniti da Microsoft. Offre servizi tipici di un *cloud* provider, nel suo caso che si integrano bene con l'ambiente di sviluppo di Microsoft. 1.

Dependency injection:

Desing pattern architettuale utilizzato principalmente nella programmazione ad oggetti per gestire le dipendenze delle classi in modo flessibile. Ogni classe non deve preoccuparsi di creare gli oggetti da cui dipende, ma basta che siano esplicitate tramite costruttore o appositi *setter*. Poi il *dependency injector* si occuperà di risolvere le dipendenze e passarle alle classi. 6.

ITS – Issue Tracking System:

Strumento che ha come obiettivo la gestione delle *issue*, ovvero unità di lavoro che apportano miglioramenti ad un prodotto. Le *issue* possono riguardare *bug*, nuove *feature*, documentazione e molto altro. La gestione attraverso un ITS permette di ottimizzare e tracciare la loro gestione. 3., 50.

JWT – Json Web Token:

Standard *web* per lo scambio di dati definito dalla RFC 7519 proposto nel

2015. Un token JWT è composto da 3 sezioni: l'*header*, che contiene i dati relativi al tipo di algoritmo usato per la codifica, il *payload*, che contiene i dati in formato JSON codificati e una signature (firma) che garantisce la veridicità del *token*. [2] 30.

LLM – Large Language Model:

Modello del linguaggio allenato su enormi quantità di testo e utilizzato per compiti riguardanti il *natural language processing*. 7., 8., 9., 12., 13., 28., 37., 39., 48.

Serverless:

Modello di esecuzione in cloud in cui il *provider* del servizio alloca le risorse della macchina solo nel momento in cui arriva una richiesta. Oltre ad essere facilmente scalabile permette di ridurre i costi, visto che si paga solo il tempo di esecuzione effettiva. 1.

User Story:

Nello sviluppo *software* e nella gestione del prodotto, una *user story* è una descrizione informale, in linguaggio naturale delle funzionalità di un *software*. Sono scritte dalla prospettiva dell'utente finale e possono essere registrate su schede, note adesive o digitalmente in *software* di gestione specifici. [3] 17., 49.

VCS – Version Control System:

Strumento che permette di gestire i cambiamenti avvenuti nei *file* di progetto nel tempo. Ha una storia completa dei cambiamenti dei *file* e quindi permette di ripristinare una data versione immediatamente. Permette la collaborazione, solitamente attraverso l'utilizzo di *branch*, ovvero una serie di modifiche distaccate da quelle centrali in cui si vanno ad implementare certe funzionalità o correggere *bug*.

Esistono varie tipologie di VCS, ovvero locali, centralizzati e distribuiti. 4., 50.

Bibliografia

- [1] Wikipedia, «Agile software development». [Online]. Disponibile su: https://en.wikipedia.org/wiki/Agile_software_development
- [2] Wikipedia, «JSON Web Token». [Online]. Disponibile su: https://it.wikipedia.org/wiki/JSON_Web_Token
- [3] Wikipedia, «User Story». [Online]. Disponibile su: https://en.wikipedia.org/wiki/User_story