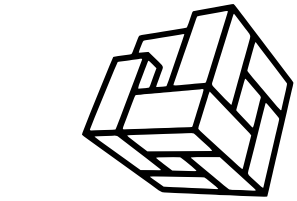
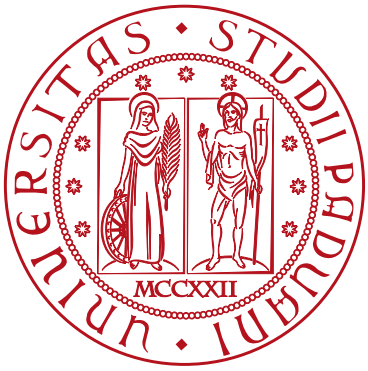


PEBKAC
Gruppo: 11
Email: pebkacswe@gmail.com
Docs: https://pebkac-swe-group-11.github.io
GitHub: https://github.com/PEBKAC-SWE-Group-11



Università degli Studi di Padova
Corso di Laurea: Informatica
Corso: Ingegneria del Software
Anno Accademico: 2024/2025

Norme di progetto

Informazioni sul documento:

Verificatori	Alessandro Benin Tommaso Zocche Derek Gusatto Davide Martinelli
Redattori	Derek Gusatto Matteo Piron Matteo Gerardin
Uso	Interno
Destinatari	Tullio Vardanega Riccardo Cardin Gruppo PEBKAC

Registro delle versioni

Versione	Data	Autore	Ruolo	Descrizione
2.0.0	2025-04-05	Derek Gusatto	Responsabile	Approvazione e rilascio
1.5.0	2025-04-05	Derek Gusatto	Verificatore	Verifica finale e completa
1.4.1	2025-04-05	Matteo Gerardin	Amministratore	Correzioni minori e aggiunta dei termini del glossario
1.4.0	2025-04-01	Derek Gusatto	Verificatore	Verificato paragrafo §3.2.4.2, dal paragrafo §4.3 al §4.3.5, dal paragrafo §4.6 al §4.8.3, dal paragrafo §5.1.4.4 al §5.4.3 e dal paragrafo §7 a §7.15
1.3.1	2025-04-01	Matteo Gerardin	Amministratore	Aggiunto paragrafo §3.2.4.2, dal paragrafo §4.3 al §4.3.5, dal paragrafo §4.6 al §4.8.3, dal paragrafo §5.1.4.4 al §5.4.3 e dal paragrafo §7 a §7.15
1.3.0	2025-03-31	Derek Gusatto	Verificatore	Verifica versione 1.2.1
1.2.1	2025-03-29	Matteo Gerardin	Amministratore	Aggiunta da paragrafo §3.2.4 a §3.2.7.4 e da paragrafo §4.3 a §4.4.3
1.2.0	2025-03-26	Derek Gusatto	Verificatore	Verifica versione 1.1.1
1.1.1	2025-03-12	Matteo Gerardin	Amministratore	Aggiunta da paragrafo §3.2.3.6 a §3.2.3.9
1.1.0	2025-03-09	Davide Martinelli	Verificatore	Verifica paragrafi modificati nella versione 1.0.1
1.0.1	2025-03-06	Matteo Gerardin	Amministratore	Modifica §4.1.4 Ciclo di vita dei documenti e aggiunta da §3.2.2.5 a §3.2.3.5.2
1.0.0	2025-01-26	Derek Gusatto	Responsabile	Approvazione e rilascio
0.4.0	2025-01-26	Davide Martinelli	Amministratore	Verifica intero documento
0.3.1	2025-01-23	Matteo Gerardin	Amministratore	Inserimento redattori e verificatori nella prima pagina

0.3.0	2025-01-15	Derek Gusatto	Verificatore	Verifica paragrafi aggiunti da V0.2.0
0.2.2	2025-01-11	Matteo Gerardin	Amministratore	Configuration Control (fino a §4.2.5.2)
0.2.1	2024-12-28	Matteo Gerardin	Amministratore	Inizio §4.2.2 Configuration Control (fino a §4.2.2.3.5)
0.2.0	2024-12-17	Tommaso Zocche	Verificatore	Verifica parziale
0.1.5	2024-12-17	Tommaso Zocche	Verificatore	Correzioni “strutturali” e typo
0.1.4	2024-12-5	Matteo Piron	Amministratore	§6 Metriche di qualità
0.1.3	2024-11-25	Derek Gusatto	Amministratore	§4.2 Configuration Management (struttura e scopo)
Versione	**Data**	**Autore**	**Ruolo**	**Descrizione**
0.1.2	2024-11-22	Derek Gusatto	Amministratore	§3.2.2 Analisi dei requisiti
0.1.1	2024-11-20	Derek Gusatto	Amministratore	§3.1 Fornitura
0.1.0	2024-11-19	Alessandro Benin	Verificatore	Verifica parziale
0.0.3	2024-11-17	Derek Gusatto	Amministratore	§5.1 Gestione organizzativa
0.0.2	2024-11-16	Derek Gusatto	Amministratore	§4.1 Documentazione
0.0.1	2024-11-14	Derek Gusatto	Amministratore	§1 Introduzione, §2 Standard ISO/IEC 12207:1195

Indice

1 Introduzione **10**
1.1 Scopo del documento . . . 10
1.2 Scopo del prodotto . . . 10
1.3 Glossario . . . 10
1.4 Riferimenti . . . 10
1.4.1 Riferimenti normativi . . . 10
1.4.2 Riferimenti informativi . . . 10

2 Standard ISO/IEC 12207:1995 **12**
2.1 Processi del ciclo di vita del software$_G$ . . . 12
2.1.1 Processi primari . . . 12
2.1.2 Processi di supporto . . . 12
2.1.3 Processi organizzativi . . . 13
2.1.4 Ruoli . . . 13

3 Processi Primari **14**
3.1 Fornitura . . . 14
3.1.1 Scopo . . . 14
3.1.2 Implementazione . . . 14
3.1.3 Gestione . . . 14
3.1.4 Documentazione fornita . . . 14
3.1.4.1 Piano di Progetto . . . 15
3.1.4.2 Analisi dei requisiti . . . 15
3.1.4.3 Piano di Qualifica . . . 16
3.1.4.4 Glossario . . . 16
3.1.5 Strumenti . . . 16
3.2 Sviluppo . . . 17
3.2.1 Scopo . . . 17
3.2.2 Analisi dei Requisiti . . . 17
3.2.2.1 Scopo . . . 17
3.2.2.2 Implementazione . . . 17
3.2.2.3 Casi d'uso . . . 18
3.2.2.3.1 Notazione . . . 18
3.2.2.3.2 Diagrammi UML$_G$ . . . 18
3.2.2.4 Requisiti . . . 19
3.2.2.4.1 Notazione . . . 19
3.2.2.4.2 Suddivisione . . . 19
3.2.2.5 Strumenti . . . 20
3.2.3 Progettazione . . . 20
3.2.3.1 Scopo . . . 20
3.2.3.2 Documentazione . . . 20
3.2.3.3 Principi di progettazione dei componenti . . . 21
3.2.3.4 Principi di progettazione di dettaglio . . . 21
3.2.3.5 Convenzioni nella progettazione di dettaglio . . . 21
3.2.3.5.1 Convenzioni di denominazione . . . 21
3.2.3.5.2 Convenzioni di rappresentazione . . . 22
3.2.3.6 Proprietà desiderate dell'architettura . . . 22

3.2.3.7 Fasi della progettazione . . . 23
3.2.3.7.1 Progettazione logica . . . 23
3.2.3.7.2 Progettazione di dettaglio . . . 23
3.2.3.8 Diagrammi delle classi . . . 23
3.2.3.9 Strumenti . . . 23
3.2.4 Codifica . . . 23
3.2.4.1 Scopo . . . 23
3.2.4.2 Nomenclatura di file e cartelle . . . 24
3.2.4.3 Norme di codifica in Python . . . 24
3.2.4.3.1 Organizzazione dei sorgenti . . . 24
3.2.4.3.2 Organizzazione delle classi . . . 24
3.2.4.3.3 Buone pratiche di codifica . . . 24
3.2.4.3.4 Regole di sintassi . . . 25
3.2.4.4 Norme di codifica in Angular . . . 26
3.2.4.4.1 Tipologie dei sorgenti . . . 26
3.2.4.4.2 Organizzazione dei sorgenti: Componenti . . . 27
3.2.4.4.3 Organizzazione dei sorgenti: Moduli . . . 27
3.2.4.4.4 Organizzazione dei sorgenti: Servizi . . . 27
3.2.4.4.5 Organizzazione dei sorgenti: Direttive . . . 28
3.2.4.4.6 Organizzazione dei sorgenti: Pipe . . . 28
3.2.4.4.7 Organizzazione dei sorgenti: Template . . . 28
3.2.4.4.8 Organizzazione dei sorgenti: Stili . . . 29
3.2.4.4.9 Organizzazione dei sorgenti: Main . . . 29
3.2.4.4.10 Buone pratiche di codifica . . . 29
3.2.4.4.11 Regole di sintassi . . . 30
3.2.4.5 Norme di codifica in SQL . . . 31
3.2.4.6 Strumenti . . . 31
3.2.5 Configurazione dell'ambiente di esecuzione . . . 32
3.2.5.1 Scopo . . . 32
3.2.5.2 Norme di codifica in Docker . . . 32
3.2.5.3 Strumenti . . . 33
3.2.6 Testing del codice . . . 34
3.2.6.1 Scopo . . . 34
3.2.6.2 Tipologie di test . . . 34
3.2.6.2.1 Test di unità . . . 34
3.2.6.2.2 Test di integrazione . . . 35
3.2.6.2.3 Test di sistema . . . 36
3.2.6.3 Notazione dei test . . . 36
3.2.6.4 Strumenti . . . 36
3.2.7 Integrazione software . . . 36
3.2.7.1 Scopo . . . 36
3.2.7.2 Branching . . . 37
3.2.7.3 Pull request . . . 37
3.2.7.4 Strumenti . . . 38

4 Processi di Supporto **39**
4.1 Documentazione . . . 39
4.1.1 Scopo . . . 39
4.1.2 Documenti . . . 39
4.1.3 Progettazione e sviluppo . . . 39
4.1.3.1 Template . . . 40
4.1.3.1.1 Parametri . . . 40
4.1.3.2 Struttura del documento . . . 40
4.1.3.3 Verbali . . . 40
4.1.3.4 Nomenclatura . . . 41
4.1.3.4.1 Verbali . . . 41
4.1.3.5 Versionamento . . . 41
4.1.3.6 Convenzioni stilistiche . . . 41
4.1.4 Ciclo di vita dei documenti . . . 42
4.1.5 Strumenti . . . 44
4.2 Configuration Management . . . 44
4.2.1 Scopo . . . 44
4.2.2 Configuration control . . . 44
4.2.2.1 Descrizione . . . 44
4.2.2.2 Scopo . . . 44
4.2.2.3 ITS_G . . . 45
4.2.2.3.1 Ticket . . . 45
4.2.2.3.2 Epic . . . 46
4.2.2.3.3 Versioni . . . 46
4.2.2.3.4 Backlog e Sprint . . . 46
4.2.2.3.5 Timeline . . . 47
4.2.2.4 Pull Request . . . 47
4.2.3 Configuration status accounting . . . 48
4.2.3.1 Scopo . . . 48
4.2.3.2 Version control . . . 48
4.2.4 Configuration evaluation . . . 48
4.2.4.1 Scopo . . . 48
4.2.4.2 Tracciamento dei requisiti . . . 48
4.2.5 Release Management . . . 49
4.2.5.1 Scopo . . . 49
4.2.5.2 Automazione compilazione documenti . . . 49
4.3 Accertamento di qualità . . . 49
4.3.1 Scopo . . . 49
4.3.2 Ciclo di Deming . . . 49
4.3.3 Metriche . . . 50
4.3.4 Obiettivi di qualità . . . 50
4.3.5 Strumenti . . . 51
4.4 Verifica . . . 51
4.4.1 Scopo . . . 51
4.4.2 Descrizione . . . 51
4.4.3 Analisi statica . . . 51
4.4.3.1 Walkthrough . . . 51
4.4.3.2 Ispezione . . . 52

4.4.4 Analisi dinamica . . . 52
4.5 Validazione . . . 53
4.5.1 Scopo . . . 53
4.5.2 Esecuzione del processo . . . 53
4.5.3 Test di accettazione . . . 53
4.6 Revisione congiunta con il proponente . . . 53
4.6.1 Scopo . . . 53
4.6.2 Implementazione . . . 54
4.6.3 Strumenti . . . 54
4.7 Verifiche ispettive interne . . . 54
4.7.1 Scopo . . . 54
4.7.2 Implementazione . . . 55
4.7.3 Strumenti . . . 55
4.8 Risoluzione dei problemi . . . 55
4.8.1 Scopo . . . 55
4.8.2 Implementazione . . . 55
4.8.3 Strumenti . . . 56

5 Processi Organizzativi **57**
5.1 Gestione organizzativa . . . 57
5.1.1 Scopo . . . 57
5.1.2 Ruoli . . . 57
5.1.2.1 Responsabile . . . 57
5.1.2.2 Amministratore . . . 57
5.1.2.3 Analista . . . 57
5.1.2.4 Progettista . . . 58
5.1.2.5 Programmatore . . . 58
5.1.2.6 Verificatore . . . 58
5.1.3 Attività . . . 58
5.1.3.1 Pianificazione . . . 58
5.1.3.1.1 Strumenti . . . 58
5.1.3.2 Esecuzione . . . 58
5.1.3.3 Revisione . . . 59
5.1.3.3.1 Strumenti . . . 59
5.1.3.4 Chiusura . . . 59
5.1.3.5 Tracciamento orario . . . 59
5.1.3.5.1 Strumenti . . . 59
5.1.4 Comunicazione . . . 59
5.1.4.1 Comunicazioni interne . . . 59
5.1.4.1.1 Comunicazioni sincrone . . . 59
5.1.4.1.2 Comunicazioni asincrone . . . 60
5.1.4.1.3 Strumenti . . . 60
5.1.4.2 Comunicazioni esterne . . . 60
5.1.4.2.1 Comunicazioni sincrone . . . 60
5.1.4.2.2 Comunicazioni asincrone . . . 60
5.1.4.2.3 Strumenti . . . 61
5.1.4.3 Norme comportamentali . . . 61
5.1.4.4 Moderazione . . . 61

5.1.4.5 Gestione dei rischi . . . 61
5.1.4.5.1 Nomenclatura . . . 62
5.1.4.5.2 Descrizione . . . 62
5.2 Gestione dell'Infrastruttura . . . 62
5.2.1 Scopo . . . 62
5.2.2 Implementazione . . . 62
5.2.3 Manutenzione . . . 63
5.2.4 Strumenti . . . 64
5.3 Miglioramento . . . 64
5.3.1 Scopo . . . 64
5.3.2 Determinazione del processo . . . 64
5.3.3 Valutazione del processo . . . 64
5.3.4 Miglioramento del processo . . . 65
5.4 Formazione . . . 65
5.4.1 Scopo . . . 65
5.4.2 Pianificazione . . . 65
5.4.3 Raccolta del materiale . . . 65

6 Metriche di qualità **66**
6.1 Metriche di qualità del processo . . . 66
6.1.1 Fornitura . . . 66
6.1.2 Sviluppo . . . 67
6.1.3 Documentazione . . . 67
6.1.4 Gestione delle qualità . . . 67
6.2 Metriche per la qualità del prodotto . . . 67
6.2.1 Funzionalità . . . 67
6.2.2 Affidabilità . . . 68
6.2.3 Manutenibilità . . . 68
6.2.4 Efficienza . . . 68

7 Strumenti **69**
7.1 Draw.io . . . 69
7.2 Discord . . . 69
7.3 Docker . . . 69
7.4 GitHub . . . 69
7.5 Google Calendar . . . 69
7.6 Google Gmail . . . 70
7.7 Google Sheets . . . 70
7.8 Jira . . . 70
7.9 LaTeX . . . 70
7.10 Microsoft PowerPoint . . . 70
7.11 Microsoft Teams . . . 70
7.12 Slack . . . 71
7.13 StarUML . . . 71
7.14 Visual Studio Code . . . 71
7.15 Whatsapp . . . 71

Elenco delle tabelle

1 Documenti del ciclo di vita del prodotto *software*$_{\mathrm{G}}$. 39

1 Introduzione

1.1 Scopo del documento

Il presente documento ha l'obiettivo di definire le *best practices*$_{\text{G}}$ e il *way of working*$_{\text{G}}$ che i componenti del team *PEBKAC* hanno l'obbligo di rispettare per l'intero svolgimento del progetto. L'intento è di garantire un metodo di lavoro omogeneo, verificabile e migliorabile nel tempo. La creazione delle norme è progressiva e incrementale nel tempo per consentire al team di apportare aggiornamenti continui in risposta alle esigenze e alle problematiche incorse durante lo sviluppo dell'intero progetto.

1.2 Scopo del prodotto

Il progetto "Vimar GENIALE" mira a sviluppare un'applicazione intelligente che supporti installatori elettrici nell'uso di dispositivi *Vimar*$_{\text{G}}$, facilitando l'accesso alle informazioni tecniche sui prodotti, rispondendo a domande poste in linguaggio naturale. La tecnologia alla base prevede l'uso di modelli di *LLM*$_{\text{G}}$ e di tecniche *RAG*$_{\text{G}}$, con una struttura di gestione basata su *container*$_{\text{G}}$ e integrata in un ambiente *cloud*$_{\text{G}}$. Il *sistema*$_{\text{G}}$ include tre componenti principali: una *applicativo web responsive*$_{\text{G}}$, un *applicativo server*$_{\text{G}}$ e un'*infrastruttura cloud-ready*$_{\text{G}}$.

1.3 Glossario

Per evitare ambiguità relative al linguaggio utilizzato nei documenti, viene fornito il Glossario V1.0.0, nel quale si possono trovare tutte le definizioni di termini che hanno un significato specifico che vuole essere disambiguato. Tali termini sono marcati con una G a pedice.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- Regolamento del progetto didattico
  https://www.math.unipd.it/ tullio/IS-1/2024/Dispense/PD1.pdf
  (Ultimo accesso 2024-11-14)
- ISO/IEC 12207:1995 Information technology - software life cycle processes
  https://www.math.unipd.it/ tullio/IS-1/2010/Approfondimenti/A03.pdf
  (Ultimo accesso 2024-11-14)

1.4.2 Riferimenti informativi

- Capitolato C2
  https://www.math.unipd.it/ tullio/IS-1/2024/Dispense/PD1.pdf
  (Ultimo accesso 2024-11-14)
- Capitolato C2 - slides
  https://www.math.unipd.it/ tullio/IS-1/2024/Dispense/PD1.pdf
  (Ultimo accesso 2024-11-14)

- Documentazione$_{G}$ GitHub$_{G}$
  https://docs.github.com/en
  (Ultimo accesso 2024-11-14)

2 Standard ISO/IEC 12207:1995

Il gruppo ha deciso di applicare nelle proprie modalità di lavoro e quindi nel presente documento di adottare lo *standard ISO/IEC 12207:1995 Information technology - Software life cycle processes*. In questa sezione del documento si possono trovare i criteri di applicazione e i processi definiti nell'ambito di questo standard.

2.1 Processi del ciclo di vita del software$_G$

Questo documento è usato per normare il *way of working*$_G$ del gruppo, in particolare l'organizzazione dei processi del ciclo di vita del *software*$_G$ secondo lo *standard ISO/IEC 12207:1995 Information technology - Software life cycle processes*, questi sono organizzati in una organizzazione gerarchica in cui ogni processo è costituito da un insieme di attività, le quali possono prevedere delle procedure e un elenco di strumenti usati per lo svolgimento.

2.1.1 Processi primari

Lo standard adottato presenta cinque processi primari (*Acquisizione, Fornitura, Sviluppo, Operazione, Manutenzione*), ma all'interno del contesto del progetto universitario in atto, il gruppo non ritiene i processi di *Operazione* e *Manutenzione* pertinenti, mentre il processo di *Acquisizione* è di competenza del *committente*$_G$, pertanto, il gruppo decide di escluderli dalla presentazione nel documento. I processi primari presentati nel presente documento sono:

- **Fornitura**: definisce le attività del fornitore, l'organizzazione che fornisce il prodotto *software*$_G$ all'acquirente;
- **Sviluppo**: definisce le attività dello sviluppatore, l'organizzazione che definisce e sviluppa il prodotto *software*$_G$.

2.1.2 Processi di supporto

I processi di supporto presentati nel presente documento sono:

- **Documentazione**$_G$: definisce le attività per la registrazione delle informazioni prodotte da un processo del ciclo di vita;
- **Configuration Management**$_G$: definisce le attività di gestione della configurazione;
- **Accertamento di qualità**$_G$: definisce le attività per assicurare in modo oggettivo che i prodotti e i processi *software*$_G$ siano conformi ai requisiti$_G$ specificati;
- **Verifica**$_G$: definisce le attività per verificare il prodotto *software*$_G$;
- **Validazione**$_G$: definisce le attività per validare il prodotto *software*$_G$;
- **Risoluzione dei problemi**: definisce un processo per analizzare e risolvere i problemi di qualsiasi natura o origine, sorti durante l'esecuzione di processi.

2.1.3 Processi organizzativi

I processi organizzativi presentati nel presente documento sono:

- **Gestione organizzativa**: definisce le attività dell'acquirente, l'organizzazione che acquisisce un prodotto *software*$_{\mathrm{G}}$;
- **Infrastruttura**: definisce le attività del fornitore, l'organizzazione che fornisce il prodotto *software*$_{\mathrm{G}}$ all'acquirente;
- **Miglioramento**: definisce le attività dello sviluppatore, l'organizzazione;
- **Formazione**: definisce le attività atte a provvedere una adeguata formazione del gruppo.

2.1.4 Ruoli

I ruoli definiti all'interno di questo progetto didattico universitario sono:

- **Docente del corso**: *committente*$_{\mathrm{G}}$;
- **Azienda proponente**: cliente e mentore;
- **Gruppo di lavoro**: fornitore.

3 Processi Primari

3.1 Fornitura

3.1.1 Scopo

La fornitura è il processo che descrive le attività svolte dal fornitore, coinvolge pianificazione, acquisizione e gestione delle *risorse*$_{\text{G}}$ necessarie. Il processo determina le procedure e le *risorse*$_{\text{G}}$ necessarie per gestire e garantire il progetto. L'obiettivo di questo processo è garantire l'*efficienza*$_{\text{G}}$ e la conformità ai *requisiti*$_{\text{G}}$ del progetto per raggiungere gli obiettivi stabiliti dal proponente.

3.1.2 Implementazione

Il processo di fornitura è composto delle seguenti fasi:

1. **Risposta alla richiesta**: il fornitore, dopo aver analizzato i *requisiti*$_{\text{G}}$ di una richiesta del proponente (il *Capitolato*$_{\text{G}}$) prepara in risposta una proposta;
2. **Negoziazione**: il fornitore negozia e stipula un contratto con il proponente;
3. **Pianificazione**: il fornitore rivede i *requisiti*$_{\text{G}}$ e valuta le opzioni per lo sviluppo del prodotto *software*$_{\text{G}}$ in base ad un'analisi dei rischi associati alle varie opzioni per definire la struttura di un piano di gestione del progetto al fine di garantire la qualità del prodotto finale;
4. **Esecuzione e controllo**: il fornitore esegue il piano di gestione del progetto, monitorando il progresso e la qualità del prodotto per tutto il ciclo di vita del prodotto;
5. **Revisione**: il fornitore coordina le comunicazioni con il proponente e partecipa a riunioni e revisioni. Il fornitore *verifica*$_{\text{G}}$ e convalida il processo per dimostrare che i prodotti e i processi soddisfano i *requisiti*$_{\text{G}}$;
6. **Consegna**: il fornitore consegna il prodotto finale, fornendo assistenza al proponente a supporto del prodotto consegnato.

3.1.3 Gestione

Al fine di identificare e comprendere i bisogni del Proponente, per poter individuare i *requisiti*$_{\text{G}}$ e i vincoli del progetto, deve essere mantenuta costante comunicazione con il Proponente, mediante riunioni SAL periodiche calendarizzate, in presenza o su *Microsoft Teams*$_{\text{G}}$ e con scambio di messaggi su *Microsoft Teams*$_{\text{G}}$ e mail qualora fosse necessario. Il dialogo continuo permette anche una valutazione costante dell'operato del fornitore, in modo da apportare correzioni, integrazioni e miglioramenti in modo tempestivo, incrementale e costruttivo.

3.1.4 Documentazione fornita

Sono di seguito elencati i documenti che PEBKAC si impegna a consegnare ai *Committenti*$_{\text{G}}$ e al Proponente:

3.1.4.1 Piano di Progetto

Il Piano di Progetto V1.0.0, redatto dal *Responsabile*$_{\mathrm{G}}$ con l'aiuto degli Amministratori, offre una guida per la pianificazione l'esecuzione e il controllo del progetto e viene utilizzato come punto di partenza principale per il monitoraggio del progresso del progetto, la gestione dei rischi e la comunicazione tra proponente e fornitore. Il Piano di Progetto comprende:

- Calendario di Progetto;
- Stima dei costi di realizzazione;
- Rischi e relativa mitigazione;
- Pianificazione e *modello di sviluppo*$_{\mathrm{G}}$;
- Preventivo e *consuntivo*$_{\mathrm{G}}$;
- Retrospettiva.

3.1.4.2 Analisi dei requisiti

L'Analisi dei Requisiti V1.0.0, redatto degli *Analisti*$_{\mathrm{G}}$, è un documento fondamentale che ha l'obiettivo principale di definire nel dettaglio le funzionalità che il prodotto deve necessariamente avere per soddisfare a pieno le richieste del Proponente. Il documento di Analisi dei Requisiti è formato da una serie di definizioni essenziali:

- **Attori$_{\mathbf{G}}$**: vengono definite entità e persone che interagiscono col *sistema*$_{\mathrm{G}}$;
- **Casi d'uso**: vengono descritti narrativamente degli scenari specifici che descrivono come gli attori interagiscono col *sistema*$_{\mathrm{G}}$. Lo scopo dei casi d'uso è offrire una visione semplice e chiara delle azioni eseguibili all'interno del sisitema e delle interazioni degli utenti con lo stesso. Per ciascun caso d'uso viene fornito un elenco delle azioni dell'*attore*$_{\mathrm{G}}$ per attivare il caso d'uso, facilitando la comprensione dei requisiti corrispondenti;
- **Requisiti**: vengono individuati i requisiti obbligatori, desiderabili e opzionali e la loro categorizzazione in:
  - **Requisiti$_{\mathbf{G}}$ funzionali**: specificano le operazioni che il *sistema*$_{\mathrm{G}}$ deve essere in grado di eseguire;
  - **Requisiti$_{\mathbf{G}}$ di qualità**: definiscono gli standard e gli attributi che il *software*$_{\mathrm{G}}$ deve possedere per garantire prestazioni, affidabilità, sicurezza e usabilità ottimali;
  - **Requisiti$_{\mathbf{G}}$ di vincolo**: definiscono vincoli e limitazioni che il *sistema*$_{\mathrm{G}}$ deve rispettare. Possono includere restrizioni tecnologiche, normative o di *risorse*$_{\mathrm{G}}$.

3.1.4.3 Piano di Qualifica

Il Piano di Qualifica V1.0.0, redatto dall'*amministratore*$_{\text{G}}$, descrive gli approcci e le strategie che il gruppo ha adottato per garantire la qualità del prodotto. Lo scopo di questo documento è quello di definire le modalità di *verifica*$_{\text{G}}$ e *validazione*$_{\text{G}}$, oltre che gli standard e le procedure di qualità che il gruppo ha deciso di adottare per il ciclo di vita del progetto.
Si compone delle sezioni riguardanti:

- **Qualità di processo**: vengono definiti standard e procedure adottate per garantire la qualità durante tutto lo sviluppo del progetto. Vengono incluse anche informazioni sulle attività di gestione della qualità, i metodi utilizzati e le misurazioni dei processi stessi;
- **Qualità di prodotto**: vengono definiti standard, specifiche e caratteristiche che il prodotto deve soddisfare per essere considerato di qualità. Vengono incluse anche metriche e criteri di valutazione utilizzati per misurare la qualità del prodotto;
- **Specifiche dei test**: vengono definite specifiche dettagliate dei *test*$_{\text{G}}$ che verranno condotti durante lo sviluppo del progetto;
- **Cruscotto delle metriche**: viene fatto un resoconto delle attività di valutazione effettuate durante il progetto per tracciare l'andamento dello stesso rispetto a obiettivi e aspettative e per identificare eventuali azioni correttive necessarie.

3.1.4.4 Glossario

Il Glossario V1.0.0 serve come un catalogo completo dei termini tecnici impiegati all'interno del progetto, fornendo definizioni chiare e precise. L'obiettivo di questo documento previene fraintendimenti a favore di una comprensione condivisa della terminologia specifica, migliorando la coerenza e la qualità della *documentazione*$_{\text{G}}$ prodotta dal gruppo.

3.1.5 Strumenti

Gli strumenti utilizzati per il processo di fornitura sono:

- Google Calendar;
- *Google Sheets*$_{\text{G}}$;
- Microsoft PowerPoint;
- *Microsoft Teams*$_{\text{G}}$.

3.2 Sviluppo

3.2.1 Scopo

Il processo di sviluppo rappresenta la serie di attività svolte dal team PEBKAC al fine di implementare il prodotto *software*$_{\text{G}}$, rispettando le scadenze e i *requisiti*$_{\text{G}}$ concordati col Proponente. Il processo è suddiviso nelle seguenti attività:

- Analisi dei requisiti,
- Progettazione;
- Codifica;
- Testing;
- Integrazione *software*$_{\text{G}}$.

3.2.2 Analisi dei Requisiti

3.2.2.1 Scopo

Lo scopo dell'analisi dei requisiti è comprendere e definire in modo chiaro e completo le necessità e le aspettative del Proponente e degli utenti relativamente al prodotto *software*$_{\text{G}}$.

3.2.2.2 Implementazione

L'analisi dei requisiti, raccolta nel documento Analisi dei Requisiti V1.0.0, viene svolta secondo le seguenti fasi:

1. Studio del *capitolato*$_{\text{G}}$ e delle esigenze del Proponente;
2. Individuazione dei casi d'uso e dei *requisiti*$_{\text{G}}$;
3. Confronto con il Proponente su quanto prodotto;
4. Divisione dei *requisiti*$_{\text{G}}$ nelle categorie individuate e applicazione dei quanto emerso nella discussione col Proponente.

L'attività di analisi può essere svolta in modo incrementale, quindi le sue fasi possono essere svolte più volte durante lo sviluppo del progetto.

L'Analisi dei Requisiti V1.0.0 contiene:

- **Introduzione**: descrive lo scopo del documento, del prodotto e i riferimenti utilizzati;
- **Descrizione**: esplicita le funzionalità attese del prodotto;
- **Attori$_{\text{G}}$**: descrive gli utilizzatori del prodotto;
- **Casi d'uso**: individua le possibili interazioni tra gli *attori*$_{\text{G}}$ e il *sistema*$_{\text{G}}$;
- **Requisiti$_{\text{G}}$**: elenca le caratteristiche da soddisfare;

3.2.2.3 Casi d'uso

I casi d'uso sono strutturati nel seguente modo:

- **Attore**$_{\text{G}}$: l'*attore*$_{\text{G}}$ che intende compiere lo scopo rappresentato dal caso d'uso;
- **Precondizioni**: stato in cui il *sistema*$_{\text{G}}$ si deve trovare prima dell'avvio della funzionalità rappresentata dal caso d'uso;
- **Postcondizioni**: stato in cui il *sistema*$_{\text{G}}$ si troverà dopo che l'utente avrà terminato lo scopo rappresentato dal caso d'uso;
- **Scenario principale**: descrizione della funzionalità rappresentata dal caso d'uso;
- **Scenari secondari** (se necessario);
- **Estensioni** (se presenti);
- **Specializzazioni** (se presenti).

3.2.2.3.1 Notazione

i casi d'uso seguono la seguente notazione: **UC[Codice] - [Titolo]** in cui:

- **UC** sta per *Use Case*$_{\text{G}}$;
- **[Codice]** è l'identificativo univoco del caso d'uso. Si tratta di un numero intero progressivo assegnato in base all'ordine di descrizione, se il caso d'uso non ha padre, altrimenti se si tratta di un sottocaso d'uso si segue la notazione **[Codice_padre]-[Numero_figlio]**, ricorsivamente senza porre limite alla profondità della gerarchia;
- **[Titolo]** è il titolo del caso d'uso.

3.2.2.3.2 Diagrammi UML$_{\text{G}}$

Un *diagramma dei casi d'uso*$_{\text{G}}$ è uno strumento di modellazione che rappresenta visivamente le funzionalità di un *sistema*$_{\text{G}}$ e le modalità con cui gli utenti interagiscono con esso. È particolarmente utile nella progettazione di sistemi poiché offre una rappresentazione intuitiva delle dinamiche operative e delle interazioni tra *attori*$_{\text{G}}$ e *sistema*$_{\text{G}}$, senza entrare nei dettagli implementativi. I componenti principali di un *diagramma dei casi d'uso*$_{\text{G}}$ sono:

1. **Attori**$_{\text{G}}$: gli *attori*$_{\text{G}}$ rappresentano entità esterne (umane o meno) che interagiscono con il *sistema*$_{\text{G}}$ e sono raffigurati con un'icona stilizzata e un'etichetta identificativa. Possono essere generalizzati: un *attore*$_{\text{G}}$ generico può avere *attori*$_{\text{G}}$ più specifici che ne ereditano le funzionalità e aggiungono comportamenti contestuali;
2. **Casi d'uso**: un caso d'uso descrive un'operazione che un utente può compiere attraverso il *sistema*$_{\text{G}}$. Ogni caso d'uso ha un'identificazione univoca e una breve descrizione della funzione. Può includere sequenze di azioni che illustrano le possibili interazioni con il *sistema*$_{\text{G}}$ ed è collegato agli *attori*$_{\text{G}}$ autorizzati tramite linee continue.

Nei diagrammi in questione poi possono comparire delle relazioni:

1. **Generalizzazioni**: le generalizzazioni possono riguardare sia gli *attori*$_{\text{G}}$ che i casi d'uso. Gli *attori*$_{\text{G}}$ o i casi figli ereditano le funzionalità dei genitori, aggiungendo aspetti specifici. La relazione è rappresentata con una freccia continua e un triangolo vuoto bianco;

2. **Inclusioni**: si verificano quando un caso d'uso ne richiama un altro in modo obbligatorio. Questo favorisce la riduzione della duplicazione e il riutilizzo delle strutture. La relazione è indicata con una freccia tratteggiata e l'etichetta "include";

3. **Estensioni**: rappresentano relazioni condizionali in cui un caso d'uso aggiuntivo viene eseguito solo in circostanze particolari, interrompendo temporaneamente il flusso principale. La relazione è raffigurata con una freccia tratteggiata e l'etichetta "extend".

3.2.2.4 Requisiti

3.2.2.4.1 Notazione

Ogni *requisito*$_{\text{G}}$ analizzato sarà identificato univocamente da una sigla del tipo **R[Tipo].[Importanza].[Codice]** nella quale:

- [**R**] sta per *Requisito*$_{\text{G}}$;
- [**Tipo**] può essere:
  - **F** per Funzionale;
  - **Q** per Qualità;
  - **V** per Vincolo.
- [**importanza**] classifica i *requisiti*$_{\text{G}}$ in:
  - **O** per Obbligatorio;
  - **D** per Desiderabile;
  - **P** per Opzionale.
- [**Codice**] identifica univocamente i *requisiti*$_{\text{G}}$ per ogni tipologia. È un numero intero progressivo univoco assegnato in ordine di importanza se il *requisito*$_{\text{G}}$ non ha padre, se invece si tratta di un sotto-*requisito*$_{\text{G}}$ segue il formato [**Codice_padre**].[**Numero_figlio**] e trattandosi di una struttura ricorsiva non c'è limite alla profondità della gerarchia.

3.2.2.4.2 Suddivisione

1. **Requisiti$_{\text{G}}$ Funzionali**: descrivono le funzionalità del *sistema*$_{\text{G}}$, le azioni che il *sistema*$_{\text{G}}$ può compiere e le informazioni che il *sistema*$_{\text{G}}$ può fornire. Seguendo la notazione sopra riportata, si possono partizionare in:
   - RF.O - *Requisito*$_{\text{G}}$ Funzionale Obbligatorio;
   - RF.D - *Requisito*$_{\text{G}}$ Funzionale Desiderabile;
   - RF.P - *Requisito*$_{\text{G}}$ Funzionale Opzionale;

2. **Requisiti$_G$ di Qualità**: descrivono come un *sistema*$_G$ deve essere, o come il *sistema*$_G$ deve essere visualizzato, per soddisfare le esigenze dell'utente. Seguendo la notazione sopra riportata, si possono partizionare in:
    - RQ.O - *Requisito*$_G$ di Qualità Obbligatorio;
    - RQ.D - *Requisito*$_G$ di Qualità Desiderabile;
    - RQ.P - *Requisito*$_G$ di Qualità Opzionale;
3. **Requisiti$_G$ Funzionali**: descrivono i limiti e le restrizioni normative/legislative che un *sistema*$_G$ deve rispettare per soddisfare le esigenze dell'utente. Seguendo la notazione sopra riportata, si possono partizionare in:
    - RV.O - *Requisito*$_G$ di Vincolo Obbligatorio;
    - RV.D - *Requisito*$_G$ di Vincolo Desiderabile;
    - RV.P - *Requisito*$_G$ di Vincolo Opzionale;

3.2.2.5 Strumenti

Gli strumenti utilizzati per il processo di sviluppo dell'Analisi dei Requisiti sono:

- Draw.io;
- StarUML.

3.2.3 Progettazione

3.2.3.1 Scopo

Lo scopo della progettazione è definire l'architettura del prodotto *software*$_G$ e fornire i componenti e le interazioni del *sistema*$_G$ per garantire che funzioni in modo efficiente ed efficace rispetto ai requisiti funzionali e non funzionali determinati durante l'attività di analisi dei requisiti.

3.2.3.2 Documentazione

La progettazione porterà alla redazione del documento di Specifica Tecnica. Questo documento ha principalmente lo scopo di descrivere l'architettura del prodotto *software*$_G$ e di mettere a disposizione una linea guida per garantire che il *sistema*$_G$ venga implementato secondo i requisiti del progetto. Gli argomenti trattati in questo documento sono:

- **Tecnologie**: espone un'analisi delle tecnologie e dei linguaggi di programmazione utilizzati, delle librerie$_G$ e dei framework$_G$ necessari, oltre che delle infrastrutture realizzate, riportando in particolare vantaggi e svantaggi di ognuna;
- **Architettura di sistema**: descrive la struttura generale del software, la suddivisione in moduli o livelli e le interazioni tra i componenti;
- **Architettura delle componenti**: fornisce i dettagli dei singoli moduli o componenti del sistema, descrivendone responsabilità, interfacce, flussi di dati, modalità di interazione ed esplicitando eventuali sottocomponenti;
- **Progettazione di dettaglio**: definisce nel dettaglio gli algoritmi, le strutture dati, i flussi operativi e l'implementazione dei singoli componenti.

3.2.3.3 Principi di progettazione dei componenti

Per ogni componente devono essere specificati i seguenti punti:

1. Una breve descrizione delle funzionalità del componente;
2. Caratteristiche del componente:
   - **Route API**: il percorso dell'endpoint esposto dal componente per interagire con il sistema;
   - **Metodo**: il tipo di richiesta HTTP supportata;
   - **Lista parametri HTTP**: l'elenco dei parametri richiesti o opzionali, inclusi nome, tipo e scopo di ciascun parametro.
3. Una lista che comprenda i possibili risultati dell'esecuzione del componente, accompagnati da:
   - Il codice che porta al verificarsi di quel particolare esito;
   - Una descrizione di quel particolare esito;
   - Il messaggio che viene visualizzato nel caso in cui si verifichi quel particolare esito.
4. Un *sistema*$_{\text{G}}$ che consenta di effettuare il tracciamento dei requisiti soddisfatti;
5. Una dichiarazione esaustiva di tutte le sottocomponenti, corredate dal loro scopo.

3.2.3.4 Principi di progettazione di dettaglio

La sezione dedicata alla progettazione di dettaglio del documento di Specifica Tecnica definisce nel dettaglio gli algoritmi, le strutture dati, i flussi operativi e l'implementazione dei singoli componenti di sistema.
Per ogni componente è necessario realizzare un diagramma delle classi e per ogni classe è necessario elencare le proprietà, che comprendono:

- **Attributi**: un elenco degli attributi della classe;
- **Implementazione**: un campo che specifica se la classe implementa un'interfaccia;
- **Estensione**: un campo che specifica se la classe estende un'altra classe;
- **Metodi**: un elenco dei metodi della classe, dove per ogni metodo vengono specificati la firma che lo identifica e una descrizione;
- **Valori**: un campo che caratterizza le enumerazioni.

3.2.3.5 Convenzioni nella progettazione di dettaglio

3.2.3.5.1 Convenzioni di denominazione

- **Nomi delle classi**: utilizzano il PascalCase (es. NomeClasse);
- **Nomi degli attributi**: utilizzano il camelCase (es. nomeAttributo);
- **Nomi dei metodi**: utilizzano il camelCase con verbi all'infinito (es. nomeMetodo());
- **Nomi delle costanti**: utilizzano lo SCREAMING_SNAKE_CASE (es. NOME_COSTANTE).

3.2.3.5.2 Convenzioni di rappresentazione

Le classi sono rappresentate con tre sezioni:

- Nome della classe;
- Attributi, rappresentati come segue:
  `visibilità nomeAttributo: tipo`
- Metodi, rappresentati con la loro firma come segue:
  `visibilità nomeMetodo(parametro0: tipo0, parametro1: tipo1): tipo`

3.2.3.6 Proprietà desiderate dell'architettura

Durante l'attività progettazione, i progettisti devono tenere conto e fare in modo che l'architettura abbia le seguenti pproprietà:

- **Affidabilità**: capacità di funzionare correttamente e senza interruzioni per lunghi periodi, gestendo errori e guasti;
- **Basso accoppiamento**: i moduli o i componenti del $sistema_{\text{G}}$ sono indipendenti e interagiscono tra loro attraverso interfacce ben definite, riducendo le dipendenze e facilitando modifiche e manutenzione;
- **Coesione**: grado in cui i componenti di un modulo sono strettamente correlati e focalizzati su un unico scopo;
- **Comprensibilità**: la chiarezza e la facilità con cui si possono comprendere la struttura, il funzionamento e i componenti del sistema;
- **Disponibilità**: capacità del $sistema_{\text{G}}$ di essere operativo e accessibile per gli utenti in modo continuo, minimizzando i tempi di inattività e garantendo un rapido ripristino in caso di guasto;
- **Efficienza**: ottimizzazione delle $risorse_{\text{G}}$ per garantire prestazioni elevate e ridurre i costi operativi senza compromettere la qualità del sistema;
- **Flessibilità**: capacità di adattarsi facilmente a nuovi requisiti, modifiche o estensioni senza richiedere una revisione radicale del sistema;
- **Incapsulazione**: nascondere i dettagli interni di un modulo, esponendo solo le interfacce necessarie;
- **Modularità**: suddivisione del $sistema_{\text{G}}$ in componenti indipendenti e intercambiabili, facilitando manutenzione, scalabilità e riusabilità;
- **Riusabilità**: capacità di progettare componenti e moduli in modo che possano essere facilmente riutilizzati in diversi contesti o progetti, riducendo costi di sviluppo e tempi di implementazione;
- **Robustezza**: capacità di funzionare correttamente anche in condizioni avverse, gestendo errori, $input_{\text{G}}$ imprevisti e guasti senza compromettere stabilità e operatività del sistema;

- **Safety**: capacità di gestire guasti *hardware*$_{\text{G}}$ o *software*$_{\text{G}}$ senza compromettere la disponibilità e l'integrità del sistema;
- **Security**: capacità di proteggere il *sistema*$_{\text{G}}$ da accessi non autorizzati, vulnerabilità e attacchi informatici;
- **Semplicità**: riduzione della complessità non necessaria, privilegiando soluzioni chiare, comprensibili e facili da mantenere senza sacrificare funzionalità o scalabilità;
- **Sufficienza**: capacità di soddisfare i requisiti attuali senza essere eccessivamente complesso o sovradimensionato rispetto alle necessità effettive.

3.2.3.7 Fasi della progettazione

3.2.3.7.1 Progettazione logica

La fase di progettazione logica consiste nel definire un'architettura che illustra la struttura e il comportamento del *sistema*$_{\text{G}}$ a un livello astratto, senza preoccuparsi delle tecnologie specifiche. In questa fase bisogna assicurarsi che i componenti siano descritti dettagliatamente e che coprano tutti i requisiti funzionali e non, semplificando così la prossima fase della progettazione.

3.2.3.7.2 Progettazione di dettaglio

La fase di progettazione di dettaglio si concentra sulla definizione precisa di tutti gli aspetti tecnici del sistema, traducendo le specifiche logiche in soluzioni concrete e implementabili. In questa fase, vengono progettati i singoli componenti del sistema, insieme alle loro interfacce, strutture dati, classi e funzioni, e la comunicazione tra i moduli. L'intento è sviluppare una base solida e completa per l'implementazione del *software*$_{\text{G}}$, con chiarezza e precisione su tutte le decisioni progettuali.

3.2.3.8 Diagrammi delle classi

I diagrammi delle classi contenuti nel documento di Specifica Tecnica devono rispettare la notazione e le specifiche dalla versione 2.5 di *UML*$_{\text{G}}$.

3.2.3.9 Strumenti

Gli strumenti utilizzati per il processo di progettazione sono:

- StarUML.

3.2.4 Codifica

3.2.4.1 Scopo

Lo scopo del processo di codifica è trasformare la progettazione dettagliata in codice sorgente eseguibile, utilizzando un linguaggio di programmazione appropriato. Questa fase consiste nello sviluppo effettivo delle funzionalità del sistema, garantendo che il codice sia corretto, efficiente, manutenibile e conforme agli standard di qualità e sicurezza. Inoltre, la codifica deve seguire le specifiche definite nelle fasi precedenti, assicurando che il *software*$_{\text{G}}$ soddisfi i requisiti funzionali e non funzionali. Questa sezione ha lo scopo di normare la scrittura del codice al fine di:

- Rendere il codice più leggibile, uniforme e robusto;
- Semplificare e velocizzare il processo di verifica;
- Agevolare la manutenzione, il debugging e l'estensione del *software*$_{\text{G}}$;
- Migliorare la qualità complessiva del codice.

3.2.4.2 Nomenclatura di file e cartelle

Per garantire una denominazione coerente e leggibile, utilizzeremo la convenzione PascalCase per i nomi di file e cartelle. In PascalCase, ogni parola inizia con una lettera maiuscola e non ci sono spazi o caratteri speciali tra le parole.

3.2.4.3 Norme di codifica in Python

3.2.4.3.1 Organizzazione dei sorgenti

La struttura generale di un sorgente *Python*$_{\text{G}}$, di solito, include i seguenti elementi:

1. **Importazioni**: inclusione dei moduli necessari;
2. **Dichiarazioni di costanti globali**: variabili usate come configurazione;
3. **Definizione di funzioni e classi**: struttura principale del codice;
4. **Blocco principale**: punto di ingresso per l'esecuzione diretta dello script.

3.2.4.3.2 Organizzazione delle classi

La struttura generale di una classe *Python*$_{\text{G}}$ solitamente include:

1. **Lista di classi estese o interfacce implementate**: se la classe eredita da una o più classi, queste vengono elencate tra parentesi nella dichiarazione della classe;
2. **Metodo costruttore**: inizializza gli attributi dell'istanza;
3. **Attributi della classe e dell'istanza**: variabili che contengono lo stato della classe o dell'oggetto;
4. **Metodi della classe**: funzioni che definiscono il comportamento dell'oggetto.

3.2.4.3.3 Buone pratiche di codifica

- **Scrivere funzioni brevi e modulari**: ogni funzione dovrebbe fare una sola cosa e farla bene;
- **Usare tipi di dati appropriati**: scegliere i tipi di dati più efficienti per ogni operazione;
- **Evitare la duplicazione del codice**: centralizzare il codice ripetuto in funzioni o classi riutilizzabili;
- **Gestire correttamente le eccezioni**: utilizzare i blocchi try/except per intercettare e gestire gli errori in modo elegante, senza interrompere il flusso del programma;

- **Ottimizzare solo quando necessario**: scrivere codice semplice e comprensibile prima di preoccuparsi delle ottimizzazioni;
- **Non accedere direttamente agli attributi di una classe**: evitare di modificare o leggere direttamente gli attributi, ma utilizzare metodi getter/setter per l'accesso;
- **Usare self per riferirsi ai parametri interni**: utilizzare il riferimento `self` per accedere agli attributi e ai metodi interni della classe;
- **Usare la minima visibilità possibile per gli attributi**: limitare l'accesso agli attributi attraverso visibilità minima, preferendo attributi privati o protetti, ove possibile;
- **Importazioni a inizio script**: le importazioni di moduli devono essere fatte all'inizio del file o modulo;
- **Evitare importazioni generiche**: evitare l'uso di importazioni con asterisco;
- **Usare commenti TODO e FIXME**: usare `TODO` per indicare sezioni di codice da completare e `FIXME` per sezioni che necessitano di revisione o miglioramenti.

3.2.4.3.4 Regole di sintassi

- **Nomi autoesplicativi**: scegliere nomi descrittivi per variabili, funzioni e classi in modo che il loro scopo sia chiaro senza dover leggere il codice in dettaglio;
- **Lingua**: scrivere nomi di variabili, funzioni e classi in inglese, mentre i commenti e le docstring vanno scritte in italiano;
- **Usare con cautela le abbreviazioni**: evitare abbreviazioni eccessive nei nomi delle variabili e delle funzioni, a meno che non siano di uso comune, per migliorare la leggibilit;
- **Nomi di variabili e funzioni in camelCase**: utilizzare parole concatenate senza spazi, dove ogni parola successiva alla prima inizia con una lettera maiuscola, per migliorare la leggibilità (ad esempio `nomeVariabile`);
- **Nomi di classi in PascalCase**: iniziare ogni parola con una lettera maiuscola per distinguere facilmente le classi dagli altri elementi (ad esempio `NomeClasse`);
- **Costanti in SCREAMING_SNAKE_CASE**: dichiarare le costanti in maiuscolo con underscore per indicare che non dovrebbero essere modificate (ad esempio `NOME_COSTANTE`);
- **Uso coerente delle virgolette**: utilizzare le virgolette doppie per le stringhe (ad esempio `"test"`);
- **Indentazione**: usare un'indentazione di una tabulazione per migliorare la leggibilità del codice;
- **Spaziature attorno agli operatori**: inserire uno spazio prima e dopo gli operatori aritmetici e di assegnazione per aumentare la chiarezza (ad esempio `a = b + c`);

- **Evitare spaziature inutili**: non lasciare spazi prima delle parentesi aperte o dopo le parentesi chiuse (ad esempio `func(arg1, arg2)`);
- **Uso corretto delle parentesi**: usare le parentesi solo quando necessario, evitando di inserirle intorno a espressioni che non lo richiedono (ad esempio `if x == 5:`);
- **Uso appropriato dei commenti**: i commenti devono essere chiari, pertinenti e aggiornati, evitando spiegazioni ovvie. Usare `#` per commenti brevi e docstring per descrizioni più dettagliate;
- **Lunghezza delle righe**: limitare la lunghezza delle righe di codice a un massimo di 79 caratteri per migliorare la leggibilità su diversi schermi;
- **Evitare più istruzioni su una riga**: scrivere una sola istruzione per riga, evitando di separarle con punto e virgola per migliorare la leggibilità.
- **Condizioni e cicli su una sola riga**: evitare di scrivere condizioni o cicli complessi su una sola riga per non compromettere la leggibilità;
- **Linee vuote tra sezioni di codice**: usare una riga vuota tra funzioni e due tra classi per separare logicamente le sezioni del codice;

3.2.4.4 Norme di codifica in Angular

3.2.4.4.1 Tipologie dei sorgenti

In **Angular**$_{\text{G}}$, esistono diverse tipologie di sorgenti, ognuna con un *ruolo*$_{\text{G}}$ specifico per garantire modularità, scalabilità e manutenibilità:

- **Componenti** (`*.component.ts`). definiscono le unità dell'interfaccia utente con logica, template e stili;
- **Moduli** (`*.module.ts`): organizzano i componenti, direttive e servizi in unità indipendenti;
- **Servizi** (`*.service.ts`): gestiscono la logica di business e la comunicazione con il *backend*$_{\text{G}}$;
- **Direttive** (`*.directive.ts`): modificano il comportamento degli elementi HTML senza alterarne la struttura;
- **Pipe** (`*.pipe.ts`): trasformano i dati nei template per la formattazione o elaborazione;
- **Template** (`*.component.html`): contengono il codice HTML che definisce l'interfaccia utente;
- **Stili** (`*.component.scss` **o** `.css`): definiscono l'aspetto grafico dei componenti e dell'app;
- **File di configurazione** (`angular.json`, `package.json`, `tsconfig.json`): contengono impostazioni per compilazione, dipendenze e build;
- **Main** (`main.ts`): punto di ingresso dell'applicazione, che si occupa di avviare il modulo principale;

- **Environments (`environment.ts`)**: gestiscono configurazioni specifiche per diversi ambienti di esecuzione.

3.2.4.4.2 Organizzazione dei sorgenti: Componenti

La struttura generale di un sorgente `.component.ts` in *Angular*$_{\text{G}}$, di solito, include i seguenti elementi:

1. **Importazioni**: si importano i moduli e le dipendenze necessarie;
2. **Decoratore `@Component`**: contiene i metadati del componente;
3. **Definizione della classe**: la classe *Typescript*$_{\text{G}}$ esportata del componente definisce la logica dell'interfaccia utente:
   - **Dichiarazione delle proprietà**: variabili utilizzate nel template per gestire dati e stato del componente;
   - **Costruttore**: inizializza il componente e inietta eventuali dipendenze necessarie;
   - **Metodi**: funzionalità definite per gestire eventi, logica di business o manipolazione dei dati.

3.2.4.4.3 Organizzazione dei sorgenti: Moduli

La struttura generale di un sorgente `.module.ts` in *Angular*$_{\text{G}}$, di solito, include i seguenti elementi:

1. **Importazioni**: importa i moduli necessari;
2. **Decoratore `@NgModule`**: contiene i metadati del modulo:
   - **Dichiarazioni**: elenca i componenti, direttive e pipe appartenenti al modulo;
   - **Importazioni**: specifica altri moduli necessari per il funzionamento dell'app;
   - **Provider**: definisce i servizi disponibili all'interno del modulo;
   - **Bootstrap (solo per il modulo principale)**: indica il componente radice avviato dall'applicazione.
3. **Definizione della classe**: il modulo viene definito come una classe *Typescript*$_{\text{G}}$ esportata.

3.2.4.4.4 Organizzazione dei sorgenti: Servizi

La struttura generale di un sorgente `.service.ts` in *Angular*$_{\text{G}}$, di solito, include i seguenti elementi:

1. **Importazioni**: si importano i moduli e le dipendenze necessarie;
2. **Decoratore `@Injectable()`**: indica che la classe è un servizio e può essere iniettata in altri componenti o servizi;
3. **Definizione della classe**: la classe *Typescript*$_{\text{G}}$ esportata che implementa il servizio, contenente la logica centralizzata:

- **Dichiarazione delle proprietà**: variabili utilizzate nel template per gestire dati e stato del servizio;
- **Costruttore**: inizializza il servizio e inietta altri eventuali servizi necessari;
- **Metodi**: funzionalità fornite dal servizio.

3.2.4.4.5 Organizzazione dei sorgenti: Direttive

La struttura generale di un sorgente `.directive.ts` in *Angular*$_{\mathrm{G}}$, di solito, include i seguenti elementi:

1. **Importazioni**: si importano i moduli necessari;
2. **Decoratore** `@Directive()`: definisce il selettore della direttiva, che può essere applicato agli elementi HTML;
3. **Definizione della classe**: definisce la classe esportata che implementa la logica della direttiva:
   - **Dichiarazione delle proprietà**: variabili utilizzate nel template per gestire dati e stato del servizio;
   - **Costruttore**: inizializza la direttiva e, si possono iniettare `ElementRef` per accedere all'elemento DOM e `Renderer2` per manipolarla in modo sicuro;
   - **Gestione degli eventi (`@HostListener`)**: ascolta eventi per eseguire azioni sugli elementi HTML;
   - **Metodi**: funzionalità fornite dalla direttiva.

3.2.4.4.6 Organizzazione dei sorgenti: Pipe

La struttura generale di un sorgente `.pipe.ts` in *Angular*$_{\mathrm{G}}$, di solito, include i seguenti elementi:

1. **Importazioni**: si importano i moduli necessari;
2. **Decoratore** `@Pipe()`: definisce il nome della pipe, utilizzato nei template HTML;
3. **Definizione della classe**: la classe esportata implementa l'interfaccia `PipeTransform` e contiene tutta la logica per la trasformazione dei dati tramite l'implementazione del metodo `transform()`:
   - **Gestione delle trasformazioni**: nel metodo `transform()`, si applicano le trasformazioni complesse e condizionali sui dati;

3.2.4.4.7 Organizzazione dei sorgenti: Template

La struttura generale di un sorgente `.component.html` in *Angular*$_{\mathrm{G}}$, di solito, include i seguenti elementi:

1. **Markup HTML**: contiene gli elementi HTML che definiscono la struttura della vista;
2. **Binding dei dati**: utilizza la sintassi di binding di *Angular*$_{\mathrm{G}}$ per associare variabili e proprietà ai componenti del DOM;

3. **Direttive Angular**: direttive per manipolare il DOM in base a dati dinamici;
4. **Componenti figli**: componenti inclusi all'interno di un template;
5. **Event binding**: sintassi per collegare eventi alle azioni nel componente;
6. **Template reference variables**: variabili di riferimento per accedere agli elementi del DOM o ai componenti figli;
7. **Formattazione e styling**: può includere stili inline o riferimenti a fogli di stile esterni.

3.2.4.4.8 Organizzazione dei sorgenti: Stili

La struttura generale di un sorgente `.component.css o .scss` in *Angular*$_{\text{G}}$, di solito, include i seguenti elementi:

1. **Definizioni di variabili**: le variabili *CSS*$_{\text{G}}$ vengono usate per centralizzare valori comuni, come colori e dimensioni;
2. **Definizione degli stili locali**: contiene le regole *CSS*$_{\text{G}}$ per definire l'aspetto del componente;

3.2.4.4.9 Organizzazione dei sorgenti: Main

La struttura generale di un sorgente `main.ts` in *Angular*$_{\text{G}}$, di solito, include i seguenti elementi:

1. **Importazione di moduli e funzionalità**: importa il modulo `platformBrowserDynamic` per avviare l'applicazione nel browser e il modulo principale;
2. **Bootstrap del modulo principale**: avvia l'applicazione chiamando `platformBrowserDynamic().bootstrapModule(AppModule)`.

3.2.4.4.10 Buone pratiche di codifica

- **Seguire le convenzioni di naming**: usa nomi significativi e coerenti per componenti, servizi, moduli e variabili;
- **Usare la modularizzazione**: organizza il codice in moduli *Angular*$_{\text{G}}$ separati per gestire diverse funzionalità;
- **Organizzare il codice in cartelle tematiche**: organizza il codice in base alla funzionalità e non per tipo di file;
- **Mantenere i componenti piccoli e riutilizzabili**: ogni componente dovrebbe avere una sola responsabilità e dovrebbe essere riutilizzabile;
- **Separare la logica di presentazione dalla logica di business**: mantieni separata la logica di presentazione (componenti) da quella di business (servizi);
- **Evita logica complessa nei componenti**: sposta la logica complessa nei servizi, mantenendo i componenti leggeri e facilmente testabili;

- **Utilizzare i servizi per la gestione della logica e delle chiamate API**: centralizza la logica di business e la gestione delle chiamate alle *API*$_{\text{G}}$ nei servizi, e iniettili nei componenti tramite dependency injection;
- **Usare l'injection di dipendenze**: utilizza l'iniezione di dipendenze per gestire le dipendenze tra componenti e servizi;
- **Usare i Reactive Forms per una gestione avanzata dei moduli**: usa i Reactive Forms invece dei Template-driven Forms per una gestione avanzata della *validazione*$_{\text{G}}$ dei moduli;
- **Usare il binding unidirezionale quando possibile**: preferisci il binding unidirezionale piuttosto che il binding bidirezionale;
- **Usare le direttive in modo modulare**: le direttive devono essere piccole, modulari e riutilizzabili;
- **Usare i pipe per la trasformazione dei dati**: usa le pipe per trasformare i dati nella vista, mantenendo la logica di presentazione fuori dai componenti;
- **Gestire gli errori in modo appropriato**: utilizza il *sistema*$_{\text{G}}$ di gestione degli errori di *Angular*$_{\text{G}}$;
- **Usare il Lazy Loading per migliorare le performance**: carica solo i moduli necessari all'avvio dell'applicazione usando il Lazy Loading;
- **Ottimizzare le performance**: usa tecniche per migliorare le performance di rendering;
- **Usare il *CSS*$_{\text{G}}$ scoped**: utilizza il `ViewEncapsulation` di *Angular*$_{\text{G}}$ (scoped *CSS*$_{\text{G}}$) per evitare che gli stili di un componente influenzino altri componenti;
- **Sfruttare i moduli di terze parti con cautela**: usa librerie di terze parti per risolvere problemi comuni, ma evita di sovraccaricare l'applicazione con dipendenze non necessarie;
- **Usare commenti TODO e FIXME**: usare TODO per indicare sezioni di codice da completare e FIXME per sezioni che necessitano di revisione o miglioramenti;

3.2.4.4.11 Regole di sintassi

- **Nomi autoesplicativi**: scegliere nomi descrittivi per componenti, servizi, moduli e variabili in modo che il loro scopo sia chiaro senza dover leggere il codice in dettaglio;
- **Lingua**: scrivere nomi di variabili, funzioni e classi in inglese, mentre i commenti e la *documentazione*$_{\text{G}}$ possono essere in italiano;
- **Usare con cautela le abbreviazioni**: evitare abbreviazioni eccessive nei nomi delle variabili e delle funzioni, a meno che non siano di uso comune, per migliorare la leggibilità;

- **Nomi di componenti, servizi e moduli in PascalCase**: utilizzare parole concatenate dove ogni parola inizia con una lettera maiuscola, seguite dal suffisso appropriato (ad esempio `UserProfileComponent`, `AuthService`, `AppModule`);
- **Nomi di variabili e metodi in camelCase**: iniziare con una lettera minuscola e usare lettere maiuscole per le parole successive (ad esempio `getUserData`);
- **Costanti in SCREAMING_SNAKE_CASE**: dichiarare le costanti in maiuscolo con underscore per indicare che non dovrebbero essere modificate (ad esempio `API_BASE_URL`);
- **Uso coerente delle virgolette**: preferire le virgolette singole (`'...'`) per le stringhe in *Typescript*$_{\text{G}}$, a meno che non siano necessarie virgolette doppie (`"..."`) per nidificazioni;
- **Indentazione**: usare un'indentazione di una tabulazione per migliorare la leggibilità del codice;
- **Spaziature attorno agli operatori**: inserire uno spazio prima e dopo gli operatori per aumentare la chiarezza (ad esempio `a = b + c`);
- **Evitare spaziature inutili**: non lasciare spazi prima delle parentesi aperte o dopo le parentesi chiuse (ad esempio `myFunction(arg1, arg2)`);
- **Uso corretto delle parentesi**: usare le parentesi solo quando necessario e preferire il formato multilinea per le dichiarazioni lunghe;
- **Lunghezza delle righe**: limitare la lunghezza delle righe di codice a un massimo di 100 caratteri per migliorare la leggibilità su diversi schermi;
- **Evitare più istruzioni su una riga**: scrivere una sola istruzione per riga, evitando di separarle con punto e virgola per migliorare la leggibilità;
- **Linee vuote tra sezioni di codice**: usare una riga vuota tra funzioni e due tra classi per separare logicamente le sezioni del codice;
- **Organizzazione delle importazioni**: mantenere un ordine chiaro: prima i moduli di *Angular*$_{\text{G}}$, poi le librerie di terze parti, infine le importazioni locali.

3.2.4.5 Norme di codifica in SQL

Per le norme di codifica del linguaggio *SQL*$_{\text{G}}$ è stato seguito lo standard ISO/ANSI SQL:2019, adottando ove necessario le norme specifiche applicate da *PostgreSQL*$_{\text{G}}$.

3.2.4.6 Strumenti

Gli strumenti utilizzati per il processo di codifica sono:

- Visual Studio Code;
- GitHub.

3.2.5 Configurazione dell'ambiente di esecuzione

3.2.5.1 Scopo

Lo scopo del processo di configurazione dell'ambiente di esecuzione è garantire che l'ambiente in cui il *software*$_{\mathrm{G}}$ sarà eseguito sia correttamente predisposto, ottimizzato e sicuro per supportare l'esecuzione dell'applicazione. Questa fase consiste nell'installazione e configurazione di *software*$_{\mathrm{G}}$, strumenti e dipendenze necessarie, come *database*$_{\mathrm{G}}$, server web, linguaggi di programmazione e librerie. Inoltre, è fondamentale assicurarsi che l'ambiente di esecuzione sia conforme agli standard aziendali e alle specifiche del progetto, riducendo al minimo i problemi che potrebbero sorgere durante l'esecuzione del sistema. L'obiettivo principale di questa sezione è di:

- Creare un ambiente stabile e coerente per l'esecuzione dell'applicazione;
- Ottimizzare le *risorse*$_{\mathrm{G}}$ *hardware*$_{\mathrm{G}}$ e *software*$_{\mathrm{G}}$ per garantire le migliori performance;
- Garantire che tutte le dipendenze siano correttamente configurate e aggiornate;
- Minimizzare gli errori di esecuzione dovuti a configurazioni errate o incompatibilità;
- Facilitare il deployment e la gestione dell'applicazione in ambienti diversi.

3.2.5.2 Norme di codifica in Docker

La creazione dei file *Docker*$_{\mathrm{G}}$ è una parte fondamentale del processo di sviluppo del *software*$_{\mathrm{G}}$. Le *best practices*$_{\mathrm{G}}$ e le linee guida per la scrittura dei file *Docker*$_{\mathrm{G}}$ sono essenziali per assicurare la gestione, la distribuzione e l'efficienza dei *container*$_{\mathrm{G}}$. Di seguito sono riportate alcune linee guida principali:

- **Chiarezza e coerenza**:
  - Utilizzare nomi significativi e descrittivi per le immagini e i *container*$_{\mathrm{G}}$;
  - Mantenere una struttura uniforme e organizzata nei file *Docker*$_{\mathrm{G}}$ per garantire la coerenza.
- **Versionamento**:
  - Sempre specificare la versione dell'immagine di base per assicurare la riproducibilità dell'ambiente;
  - Evitare l'uso del tag "latest" per le immagini di produzione, per evitare incertezze su quale versione venga utilizzata.
- **Minimizzazione degli strati (Layering)**:
  - Ridurre al minimo il numero di istruzioni nel Dockerfile per contenere il numero di strati nell'immagine;
  - Raggruppare insieme le istruzioni che possono essere eseguite in una sola fase, per migliorare l'efficienza della cache di *Docker*$_{\mathrm{G}}$.
- **Sicurezza**:

  - Utilizzare immagini ufficiali o verificate per garantire la sicurezza e l'affidabilità;
  - Evitare di eseguire processi con privilegi elevati quando non strettamente necessario;
  - Parametrizzare le informazioni sensibili tramite l'uso di `ARG`.

- **Ottimizzazione delle risorse**:
  - Limitare l'uso delle $risorse_{\mathrm{G}}$ del $container_{\mathrm{G}}$ (come CPU e memoria) per migliorare l'efficienza;
  - Preferire immagini leggere e ottimizzate per l'ambiente di produzione.

- **Gestione delle variabili d'ambiente**:
  - Usare variabili d'ambiente per configurazioni dinamiche;
  - Fornire valori di default sensati per le variabili d'ambiente, per facilitare la configurazione.

- **Logging e monitoraggio**:
  - Configurare adeguatamente i $container_{\mathrm{G}}$ per la gestione dei log;
  - Integrare strumenti di monitoraggio, quando necessario, per tenere traccia delle performance.

- **Pulizia e riduzione delle dimensioni**:
  - Rimuovere pacchetti temporanei e file non necessari dopo l'installazione delle dipendenze per mantenere l'immagine leggera;
  - Utilizzare i multi-stage builds per ottimizzare le immagini e ridurre le loro dimensioni.

- **Testing**:
  - Implementare $test_{\mathrm{G}}$ automatizzati per verificare la corretta funzionalità del Dockerfile e dei $container_{\mathrm{G}}$.

3.2.5.3 Strumenti

Gli strumenti utilizzati per il processo di configurazione dell'ambiente di esecuzione sono:

- Visual Studio Code;
- Docker;
- GitHub.

3.2.6 Testing del codice

3.2.6.1 Scopo

Lo scopo del processo di testing del codice è verificare che il *software*$_{\text{G}}$ funzioni correttamente, rispetti i requisiti e sia privo di *bug*$_{\text{G}}$ o malfunzionamenti. Questa fase consiste nell'eseguire una serie di *test*$_{\text{G}}$ per identificare eventuali errori nel codice, determinare la qualità del *sistema*$_{\text{G}}$ e garantire che ogni componente funzioni come previsto. Inoltre, è fondamentale assicurarsi che le modifiche al codice non introducano regressioni o nuovi problemi. L'obiettivo principale di questa sezione è di:

- Verificare che il codice funzioni come previsto e soddisfi i requisiti funzionali e non funzionali;
- Identificare e correggere eventuali *bug*$_{\text{G}}$ o malfunzionamenti nel sistema;
- Assicurare che le modifiche non introducano regressioni o nuovi problemi nel sistema;
- Garantire che il *software*$_{\text{G}}$ sia di alta qualità, stabile e privo di errori;
- Facilitare la manutenzione e l'evoluzione del *software*$_{\text{G}}$, riducendo il rischio di problemi futuri.

3.2.6.2 Tipologie di test

I *test*$_G$ realizzabili possono essere suddivisi in tre categorie principali:

- ***Test*$_G$ di unità**: verificano il corretto funzionamento di una singola unità di codice indipendente (ad esempio una funzione), assicurandosi che produca i risultati attesi al variare dei possibili input$_{\text{G}}$, e vengono generalmente automatizzati per facilitare l'individuazione degli errori durante la fase di sviluppo;
- ***Test*$_G$ di integrazione**: verificano il corretto funzionamento delle interazioni tra diverse unità di codice o componenti di un sistema, assicurandosi che, una volta integrati, i vari moduli lavorino insieme senza problemi, rilevando eventuali errori nelle interfacce e nei flussi di dati tra di essi;
- ***Test*$_G$ di sistema**: verificano il comportamento complessivo di un'intera applicazione o sistema, testando tutte le sue componenti integrate per assicurarsi che soddisfi i *requisiti*$_G$ funzionali e non funzionali, assicurandosi di valutare il *sistema*$_{\text{G}}$ nel suo insieme simulando l'uso reale per identificare eventuali problemi di performance, sicurezza o compatibilità;

3.2.6.2.1 Test di unità

- **Redazione**: i verificatori sono responsabili della scrittura e manutenzione dei *test*$_{\text{G}}$ di unità per il codice. Essi si occupano della loro stesura in ogni fase dello sviluppo del codice per garantire che sia robusto, affidabile e facilmente manutenibile;
- **Descrizione**: i *test*$_{\text{G}}$ di unità, noti anche come unit testing, sono una pratica fondamentale nello sviluppo *software*$_{\text{G}}$ che consiste nel verificare il corretto funzionamento delle più piccole parti isolabili del codice, come singole funzioni, metodi o classi.

L'obiettivo principale è assicurarsi che ciascuna unità operi secondo le aspettative, facilitando l'identificazione precoce di errori e garantendo una maggiore qualità del $software_{\mathrm{G}}$. I $test_{\mathrm{G}}$ di unità presentano alcune caratteristiche fondamentali che ne garantiscono l'efficacia. Una di queste è l'isolamento, che assicura che ogni $test_{\mathrm{G}}$ venga eseguito separatamente, senza dipendenze da altre parti del sistema, in modo da attribuire i risultati esclusivamente all'unità in esame. Un'altra caratteristica essenziale è l'automatizzazione, che consente di eseguire i $test_{\mathrm{G}}$ in modo rapido e frequente, facilitando il processo di integrazione continua e riducendo il rischio di errori durante lo sviluppo;

- **Tipologie** : esistono due tipologie di $test_{\mathrm{G}}$ di unità:
  - **Test Funzionali (Black Box Testing)**: questi $test_{\mathrm{G}}$ si concentrano sulla $verifica_{\mathrm{G}}$ che una specifica unità di codice fornisca l'$output_{\mathrm{G}}$ corretto in risposta a determinati input, senza considerare l'implementazione interna;
  - **Test Strutturali (White Box Testing)**: questi $test_{\mathrm{G}}$ valutano l'implementazione interna dell'unità, analizzando il flusso del codice e la copertura delle istruzioni o dei rami.

3.2.6.2.2 Test di integrazione

- **Redazione**: i verificatori sono responsabili della scrittura e manutenzione dei $test_{\mathrm{G}}$ di integrazione per il codice. Essi si occupano della loro stesura dopo la stesura dei $test_{\mathrm{G}}$ di unità e prima della stesura dei $test_{\mathrm{G}}$ di sistema;

- **Descrizione**: i $test_{\mathrm{G}}$ di integrazione sono una fase cruciale nel ciclo di vita dello sviluppo $software_{\mathrm{G}}$, focalizzata sulla $verifica_{\mathrm{G}}$ dell'interazione tra diversi moduli o componenti di un sistema. Dopo aver eseguito i $test_{\mathrm{G}}$ di unità su singoli componenti, i $test_{\mathrm{G}}$ di integrazione combinano progressivamente questi moduli e ne valutano il funzionamento congiunto, assicurandosi che collaborino correttamente per soddisfare i requisiti funzionali e prestazionali dell'applicazione. L'obiettivo principale è identificare e risolvere eventuali problemi che possono sorgere quando le diverse parti del $software_{\mathrm{G}}$ interagiscono, garantendo così la coerenza e l'affidabilità dell'intero sistema;

- **Tipologie**: esistono due tipologie di $test_{\mathrm{G}}$ di integrazione:
  - **Test di integrazione Bottom-Up**: questa strategia prevede l'inizio dei $test_{\mathrm{G}}$ dai moduli di livello più basso nella gerarchia del $software_{\mathrm{G}}$. I singoli componenti vengono testati individualmente e, una volta verificata la loro correttezza, vengono integrati progressivamente con i moduli di livello superiore. Questo approccio è particolarmente utile quando lo sviluppo dei moduli di base è completato prima di quelli di livello superiore;
  - **Test di integrazione Top-Down**: in questo approccio, i $test_{\mathrm{G}}$ iniziano dai moduli di livello più alto e procedono verso il basso. I moduli superiori vengono testati utilizzando "stub" per simulare il comportamento dei moduli inferiori non ancora sviluppati o testati. Man mano che i moduli di livello inferiore diventano disponibili, vengono integrati e testati in combinazione con quelli superiori. Questo metodo è vantaggioso quando l'architettura del $sistema_{\mathrm{G}}$ è ben definita fin dall'inizio;

3.2.6.2.3 Test di sistema

- **Redazione**: i verificatori sono responsabili della scrittura e manutenzione dei $test_{\mathrm{G}}$ di $sistema_{\mathrm{G}}$ per il codice. Essi si occupano della loro stesura dopo la stesura dei $test_{\mathrm{G}}$ di unità e di sistema;
- **Descrizione**: lo scopo principale dei $test_{\mathrm{G}}$ di $sistema_{\mathrm{G}}$ è di fornire una valutazione completa e indipendente del $sistema_{\mathrm{G}}$ $software_{\mathrm{G}}$ completato rispetto ai requisiti $software_{\mathrm{G}}$ definiti, garantendo che il prodotto sia pronto per la fase di collaudo con il cliente. Essi sono funzionali (ovvero black-box), il che significa che si basano sulla specifica dei requisiti $software_{\mathrm{G}}$ e non richiedono conoscenza della logica interna del $software_{\mathrm{G}}$.

3.2.6.3 Notazione dei test

É stato decisa come notazione per identificare univocamente i $test_G$ la seguente:

T[Tipologia][Numero]

Tipologia indica la tipologia del $test_G$:

- **U**: di unità;
- **I**: di integrazione;
- **S**: di sistema;

Ogni $test_G$ si trova in uno **Stato**, che può essere:

- **V**: verificato. Questo stato indica che il $test_G$ ha fornito un esito positivo;
- **NV**: non verificato. Questo stato indica che il $test_G$ ha fornito un esito negativo;
- **NI**: non implementato. Questo stato indica che il $test_G$ non è ancora stato implementato, e quindi non fornisce nessun esito.

3.2.6.4 Strumenti

Gli strumenti utilizzati per il processo di testing del codice sono:

- Visual Studio Code;
- GitHub.

3.2.7 Integrazione software

3.2.7.1 Scopo

L'obiettivo dell'integrazione $software_{\mathrm{G}}$ è costruire e verificare il $sistema_{\mathrm{G}}$ in modo incrementale, aumentando progressivamente il valore funzionale attraverso l'integrazione di nuove componenti in insiemi già verificati. Questo approccio facilita l'individuazione delle cause di eventuali difetti, poiché è più probabile che derivino dall'ultima integrazione effettuata. Inoltre, assicura che ogni passaggio sia reversibile, permettendo di tornare a uno stato sicuro precedente in caso di problemi.

3.2.7.2 Branching

Il *repository*$_{\text{G}}$ è costituito da alcuni *branch*$_{\text{G}}$ principali:

- **main**: *branch*$_{\text{G}}$ che contiene la versione stabile e ufficiale del progetto, pronta per essere rilasciata o distribuita;
- **develop**: *branch*$_{\text{G}}$ di sviluppo principale, in cui vengono integrate le nuove funzionalità prima di essere rilasciate nel *branch*$_{\text{G}}$ main;
- **norme-di-progetto**: *branch*$_{\text{G}}$ dedicato alla gestione e all'aggiornamento delle Norme di Progetto;
- **piano-di-progetto**: *branch*$_{\text{G}}$ dedicato alla gestione e all'aggiornamento del Piano di Progetto;
- **analisi-dei-requisiti**: *branch*$_{\text{G}}$ dedicato alla gestione e all'aggiornamento dell'Analisi dei Requisiti;
- **piano-di-qualifica**: *branch*$_{\text{G}}$ dedicato alla gestione e all'aggiornamento del Piano di Qualifica;
- **specifica-tecnica**: *branch*$_{\text{G}}$ dedicato alla gestione e all'aggiornamento della Specifica Tecnica;
- **manuale-utente**: *branch*$_{\text{G}}$ dedicato alla gestione e all'aggiornamento del Manuale Utente;
- **glossario**: *branch*$_{\text{G}}$ dedicato alla gestione e all'aggiornamento del Glossario;
- **Lettera_di_Presentazione**: *branch*$_{\text{G}}$ dedicato alla gestione e all'aggiornamento della Lettera di Presentazione alle revisioni del progetto;
- **poc-develop**: *branch*$_{\text{G}}$ utilizzato per lo sviluppo della Proof of Concept (PoC_{G});
- **mvp-develop**: *branch*$_{\text{G}}$ utilizzato per lo sviluppo del Minimum Viable Product (MVP).

Ogni modifica o nuova funzionalità viene sviluppata in un *branch*$_{\text{G}}$ dedicato, generato a partire dal *branch*$_{\text{G}}$ in cui dovrà successivamente essere integrato.

3.2.7.3 Pull request

Nel momento in cui una modifica o una nuova funzionalità arrivano alla fine del loro sviluppo, per integrare ciò che è stato fatto in uno dei *branch*$_{\text{G}}$ principali, la persona che ha creato il *branch*$_{\text{G}}$ apre una *pull request*$_{\text{G}}$ e richiede una verifica.
Per assegnare dei titoli significativi alle *pull request*$_{\text{G}}$, abbiamo seguito la seguente denominazione:

- ***Titolo***: Indica una *pull request*$_{\text{G}}$ standard, in cui vengono introdotte nuove funzionalità, aggiornamenti o modifiche al codice e alla *documentazione*$_{\text{G}}$. Il titolo deve essere descrittivo e chiaro, in modo da rendere immediatamente comprensibile l'obiettivo della modifica;

- **HOTFIX-*Titolo***: utilizzato per *pull request*$_{\text{G}}$ relative a correzioni urgenti di *bug*$_{\text{G}}$ o problemi critici riscontrati nel codice già rilasciato.

I verificatori potranno approvare la *pull request*$_{\text{G}}$ solamente dopo aver effettuato tutti i *test*$_{\text{G}}$ di unità, di integrazione e di *sistema*$_{\text{G}}$ e aver ricevuto solamente esiti positivi.

3.2.7.4 Strumenti

Gli strumenti utilizzati per il processo di testing del codice sono:

- GitHub.

4 Processi di Supporto

4.1 Documentazione

4.1.1 Scopo

Il processo di *documentazione*$_{\mathrm{G}}$ procede sempre di pari passo con tutte le attività di sviluppo, con l'obiettivo di fornire tutte le informazioni necessarie, sotto forma di testo scritto facilmente consultabile, inerenti al prodotto e alle attività stesse. Oltre a svolgere un *ruolo*$_{\mathrm{G}}$ essenziale nella descrizione del prodotto per coloro che lo sviluppano, lo distribuiscono e lo utilizzano, la *documentazione*$_{\mathrm{G}}$ svolge un *ruolo*$_{\mathrm{G}}$ di storicizzazione e di supporto alla manutenzione.

4.1.2 Documenti

In questa sezione viene descritto il piano che identifica i documenti da produrre durante il ciclo di vita del prodotto *software*$_{\mathrm{G}}$. Tutti i documenti da redigere sono presentati nella tabella che segue, vengono esclusi i documenti presentati per la candidatura per il progetto didattico, quali *Lettera di presentazione*, *Preventivo dei costi e assunzione degli impegni* e *Analisi dei capitolati*.

Nome	**Scopo**	**Redattore**	**Destinatari**	**Consegne**
Analisi dei requisiti	Definizione dei requisiti utente	*Analista*$_{\mathrm{G}}$	Azienda proponente, Docenti	RTB_{G}, PB_{G}
Norme di progetto	Regolamento normativo del gruppo	*Amministratore*$_{\mathrm{G}}$, *Responsabile*$_{\mathrm{G}}$	Docenti	RTB_{G}, PB_{G}
Piano di Progetto	Definizione temporale scadenze e progressi	*Responsabile*$_{\mathrm{G}}$	Docenti	RTB_{G}, PB_{G}
Piano di qualifica	Definizione qualità e testing	*Amministratore*$_{\mathrm{G}}$	Docenti	RTB_{G}, PB_{G}
Verbali esterni	Tracciamento riunioni esterne	*Responsabile*$_{\mathrm{G}}$, *Amministratore*$_{\mathrm{G}}$	Azienda proponente, Docenti	Candidatura, RTB_{G}, PB_{G}
Verbali interni	Tracciamento riunioni interne	*Responsabile*$_{\mathrm{G}}$, *Amministratore*$_{\mathrm{G}}$	Docenti	Candidatura, RTB_{G}, PB_{G}

Tabella 1: Documenti del ciclo di vita del prodotto *software*$_{\mathrm{G}}$.

4.1.3 Progettazione e sviluppo

In questa sezione vengono presentati gli standard e le regole (nello specifico di stile) a cui i membri di PEBKAC si devono attenere per la stesura dei documenti relativi al progetto.

4.1.3.1 Template

Per la stesura dei documenti il gruppo ha creato un template in formato $LaTeX_{\text{G}}$. Il template fornisce una struttura e un formato predefinito per semplificare la creazione di documenti, al fine di garantire coerenza, efficienza e standardizzazione della presentazione. Il template è progettato per essere facile da usare, dovendo inserire solo con piccole modifiche per rispecchiare le specificità di ciascun tipo di documento.
In particolare nel template è definite la pagina di copertina con intestazione contenente logo informazioni del gruppo e dell'Università di Padova, titolo del documento, informazioni sul documento (uso, destinatari) e un breve abstract del contenuto, oltre che altre specifiche di stile come il titolo dell'indice in italiano e il numero di pagina come `X di Tot`, dove `X` è il numero della pagina e `Tot` è il numero totale di pagine.

4.1.3.1.1 Parametri

Nel principale file $LaTex_{\text{G}}$ del template sono definiti una serie di comandi personalizzati per l'inserimento automatico delle informazioni come titolo, data, uso, destinatari e abstract.
Sono inoltre già presenti ma commentate le voci necessarie solo per i verbali (vedi §4.1.3.3 Verbali)

4.1.3.2 Struttura del documento

Tutti i documenti prodotti da PEBKAC presentano la medesima struttura, alla quale ogni membro si deve attenere durante la procedura di stesura e modifica.

- **Pagina di copertina**: come nella sezione Template precedente;
- **Registro delle versioni**: questo registro è utilizzato per tenere traccia delle varie versioni per permettere di comprendere velocemente chi ha realizzato o modificato determinate sezioni della *documentazione*$_{\text{G}}$ e quando. Il registro presenta le versioni ordinate a partire dalla versione più recente;
- **Indice**: presente per facilitare la consultazione del documento, dotato di sezioni. Il suo scopo è di facilitare e agevolare l'accesso ad un determinato contenuto all'interno nel documento;
- **Contenuto**: il contenuto vero e proprio del documento.

4.1.3.3 Verbali

I verbali differiscono dalla struttura precedentemente esposta in quanto ad essi prevedono delle sezioni aggiuntive ed obbligatorie:

- **Pagina di copertina**: nel caso di un verbale tra le informazioni sul documento compaiono anche i nominativi con i rispettivi ruoli dei membri che hanno lavorato alla loro produzione;
- **informazioni generali**: la prima sezione di un verbale deve sempre essere quella nominata "Informazioni generali" che prevede, sotto forma di elenco puntato, le seguenti informazioni:
  - Tipo di riunione,

- Luogo in cui si è tenuta la riunione (anche se telematica),
- Data in cui si è tenuta la riunione,
- Ora di inizio della riunione,
- Ora di fine della riunione,
- Membri presenti ed eventuali altre persone alla riunione,
- Membri assenti dalla riunione;

- **Todo**: l'ultima sezione di un verbale deve sempre essere quella che elenca i *task*$_{\text{G}}$ emersi durante la riunione da aggiungere al *backlog*$_{\text{G}}$. Questi vengono presentati sotto forma di tabella a due colonne:
  - **Assegnatario**: il membro a cui quel *task*$_{\text{G}}$ è stato assegnato, nel caso in cui non ve ne sia uso ma il *task*$_{\text{G}}$ possa essere autoassegnato da uno dei membri si scriverà "autoassegnazione" in corsivo;
  - **Task$_{\text{G}}$ Todo**: denominazione del *task*$_{\text{G}}$.

4.1.3.4 Nomenclatura

La nomenclatura per i documenti si ottiene unendo il nome del file in *Snake_Case* quindi con le parole separate da un underscore (_) (`Nome_del_File`), un underscore (_) e la sua versione (`1.2.3`), ottenendo per esempio `Norme_di_Progetto_1.2.3.pdf`. Nel caso di documenti il cui nome contiene una data, essa si inserisce dopo il nome, ma prima della versione, sempre usando gli underscores come separatori, nella forma YYYY-MM-DD: YYYY rappresenta l'anno, MM il mese e DD il giorno, sempre scritto in due cifre.

4.1.3.4.1 Verbali

Per quanto riguarda i verbali, per facilitarne l'ordinamento) il loro nome è la data in cui la riunione di è tenuta nella forma YYYY-MM-DD: YYYY rappresenta l'anno, MM il mese e DD il giorno, sempre scritto in due cifre. Nel caso si tratti di un verbale esterno viene aggiunta una `E`, sempre separata da underscores tra la data e la versione.

4.1.3.5 Versionamento

La versione di un documento è del tipo [**x**].[**y**].[**z**]:

- **z**: è un numero intero che incrementato dal Redattore ad ogni modifica;
- **y**: è un numero intero incrementato dal *Verificatore*$_{\text{G}}$ ad ogni *verifica*$_{\text{G}}$;
- **x**: è un numero intero che viene incrementato dal *Responsabile*$_{\text{G}}$ dopo la sua approvazione (versione di produzione).

4.1.3.6 Convenzioni stilistiche

- **Date**: tutte le date nella *documentazione*$_{\text{G}}$ prevedono il seguente formato YYYY-MM-DD, dove DD indica il giorno a due cifre, MM il mese a due cifre e YYYY l'anno a 4 cifre;
- **Elenchi**: elenchi puntati o numerati, ogni punto inizia con la lettera maiuscola e termina con ";" ad eccezione dell'ultimo che termina con ".";

- **Menzioni**: ogni menzione ad una persona, interna o esterna, avviene nel formato Nome Cognome;
- **Riferimenti interni**: i riferimenti a sezioni interne allo stesso documento devono essere riportati seguendo la notazione `§1.2 Nome sezione`, dove `§1.2` è il numero della sezione. Inoltre questi riferimenti devono essere opportunamente collegati tramite link al paragrafo indicato, senza alterare lo stile del testo;
- **Riferimenti esterni**: i riferimenti a sezioni di documenti esterni devono essere riportati seguendo la notazione `Nome Documento (versione di riferimento), Nome sezione`;
- **Link URL**: possono essere estesi o avere una visualizzazione abbreviata, ma sempre visualizzati di colore blu;
- **Caratteri maiuscoli**: devono essere utilizzati per
  - Le iniziali dei nomi;
  - Le lettere che compongono degli acronimi e le iniziali delle rispettive definizioni;
  - Le iniziali dei ruoli svolti dai componenti del gruppo;
  - Le iniziali dei ruoli definiti all'interno del progetto didattico;
  - La prima lettera di ogni elenco puntato.
- **Grassetto**: devono essere visualizzati in grassetto
  - I titoli di sezioni/sottosezioni/paragrafi di un documento;
  - Le parole che meritano enfasi;
  - Le definizioni negli elenchi puntati.
- **Caption**: ogni immagine o tabella deve avere una caption, utile a fornire una breve descrizione o spiegazione del contenuto visivo.

4.1.4 Ciclo di vita dei documenti

Ogni documento segue le fasi del seguente *workflow*$_{\text{G}}$:

1. **Assegnazione**: il gruppo assegna un documento a uno o più redattori, affiancati da uno o più verificatori;
2. **Branch$_{\text{G}}$**: si crea un *branch*$_{\text{G}}$ per lo sviluppo del documento nell'apposita *repository*$_{\text{G}}$ Docs;
3. **Template**: si copia il Template all'interno della cartella appropriata;
4. **Feature branch$_{\text{G}}$**: si crea un *branch*$_{\text{G}}$ per la realizzazione di una specifica modifica da apportare al relativo documento nell'apposita *repository*$_{\text{G}}$ Docs;
5. **Stesura**: si redige una o più sezioni del documento. Qualora serva un elevato parallelismo di lavoro è possibile usare Google Drive per la prima stesura e successivamente caricare il documento all'interno del *branch*$_{\text{G}}$;

6. **Commit$_{\mathbf{G}}$**: si esegue la *commit*$_{\mathrm{G}}$ sul feature *branch*$_{\mathrm{G}}$ creato;

7. **Pull Request$_{\mathbf{G}}$ verso il *branch*$_{\mathbf{G}}$ del documento**: si apre una *pull request*$_{\mathrm{G}}$ dal feature *branch*$_{\mathrm{G}}$ appena creato verso il *branch*$_{\mathrm{G}}$ del documento;

8. **Verifica$_{\mathbf{G}}$ della feature**: se il *verificatore*$_{\mathrm{G}}$ richiede modifiche si ripetono, in ordine, il punto 5 e il punto 6;

9. **Chiusura feature branch$_{\mathbf{G}}$**: si elimina, quando la *pull request*$_{\mathrm{G}}$ viene chiusa o risolta, il feature *branch*$_{\mathrm{G}}$ creato.

10. **Pull Request$_{\mathbf{G}}$ verso il *branch*$_{\mathbf{G}}$ develop**: si apre una *pull request*$_{\mathrm{G}}$ dal *branch*$_{\mathrm{G}}$ del documento verso il *branch*$_{\mathrm{G}}$ develop: se il documento non è pronto per la *verifica*$_{\mathrm{G}}$, ma ha bisogno di ulteriori modifiche, si apre la *pull request*$_{\mathrm{G}}$ in modalità draft, per marcarla successivamente come "Ready to Review", altrimenti in modalità normale;

11. **Verifica$_{\mathbf{G}}$ del documento**: se il *verificatore*$_{\mathrm{G}}$ richiede modifiche si ripete, in ordine, dal punto 4 al punto 9;

12. **Chiusura branch$_{\mathbf{G}}$**: si elimina, quando la *pull request*$_{\mathrm{G}}$ viene chiusa o risolta, il *branch*$_{\mathrm{G}}$ del documento.

Per la versione finale di un documento spetta al *Responsabile*$_{\mathrm{G}}$ conferire l'approvazione definitiva, annotando opportunamente nel registro delle versioni la versione `x.0.0` e la sua approvazione finale.

Per i verbali è sufficiente solamente la creazione del *branch*$_{\mathrm{G}}$ del documento, in quanto costituisce esso stesso un feature *branch*$_{\mathrm{G}}$, dato che i verbali vengono redatti interamente in un'unica volta. In conclusione, per questi è possibile utilizzare un ciclo di vita semplificato, che è rppresentato dal seguente *workflow*$_{\mathrm{G}}$:

1. **Assegnazione**: il gruppo assegna un documento a uno o più redattori, affiancati da uno o più verificatori;

2. **Branch$_{\mathbf{G}}$**: si crea un *branch*$_{\mathrm{G}}$ per lo sviluppo del documento nell'apposita *repository*$_{\mathrm{G}}$ Docs;

3. **Template**: si copia il Template all'interno della cartella appropriata;

4. **Stesura**: si redige il documento o una sua sezione. Qualora serva un elevato parallelismo di lavoro è possibile usare Google Drive per la prima stesura e successivamente caricare il documento all'interno del *branch*$_{\mathrm{G}}$;

5. **Commit$_{\mathbf{G}}$**: si esegue la *commit*$_{\mathrm{G}}$ sul *branch*$_{\mathrm{G}}$ creato;

6. **Pull Request$_{\mathbf{G}}$**: si apre una *pull request*$_{\mathrm{G}}$ dal *branch*$_{\mathrm{G}}$ appena creato verso il *branch*$_{\mathrm{G}}$ develop: se il documento non è pronto per la *verifica*$_{\mathrm{G}}$, ma ha bisogno di ulteriori modifiche, si apre la *pull request*$_{\mathrm{G}}$ in modalità draft, per marcarla successivamente come "Ready to Review", altrimenti in modalità normale;

7. **Verifica$_{\mathbf{G}}$**: se il *verificatore*$_{\mathrm{G}}$ richiede modifiche si ripete, in ordine, dal punto 3 al punto 5;

8. **Chiusura branch$_{\mathrm{G}}$**: si elimina, quando la *pull request*$_{\mathrm{G}}$ viene chiusa o risolta, il *branch*$_{\mathrm{G}}$ creato.

4.1.5 Strumenti

Gli strumenti utilizzati per il processo di *documentazione*$_{\mathrm{G}}$ sono:

- *LaTeX*$_{\mathrm{G}}$
- Visual Studio Code;
- *GitHub*$_{\mathrm{G}}$.

4.2 Configuration Management

4.2.1 Scopo

In questa sezione vengono presentate le attività svolte da PEBKAC per il processo di *Configuration Management*$_{\mathrm{G}}$. Il processo in questione consiste nell'applicazione di procedure amministrative e tecniche per l'intero ciclo di vita del *software*$_{\mathrm{G}}$, al fine di:

- Identificare, definire e stabilire una base per gli elementi *software*$_{\mathrm{G}}$ di un *sistema*$_{\mathrm{G}}$;
- Controllare le modifiche e le release degli elementi;
- Registrare lo stato degli elementi e delle richieste di modifica;
- Garantire la completezza, la coerenza e la correttezza degli elementi.

4.2.2 Configuration control

4.2.2.1 Descrizione

Il processo di configuration control è finalizzato a garantire il controllo e la coerenza delle configurazioni del *sistema*$_{\mathrm{G}}$, assicurando che tutte le modifiche apportate a *software*$_{\mathrm{G}}$, artefatti e documenti siano tracciate, gestite e allineate agli obiettivi e ai *requisito*$_{\mathrm{G}}$ del progetto.

4.2.2.2 Scopo

Il configuration control mira al raggiungimento dei seguenti punti:

- **Gestire le modifiche**: assicurare un controllo e una gestione corretti e sistematici per qualsiasi modifica nel progetto o nel *sistema*$_{\mathrm{G}}$;
- **Documentare le richieste**: registrare tutte le richieste di modifica per mantenere una cronologia accurata e completa;
- **Valutare l'impatto**: analizzare tutte le conseguenze tecniche, economiche e operative di ogni modifica proposta;
- **Decidere sull'approvazione**: stabilire criteri chiari e inequivocabili per l'approvazione o il rifiuto delle modifiche con gli *stakeholder*$_{\mathrm{G}}$ rilevanti;

- **Assicurare la tracciabilità**: creare *audit trail*$_{\text{G}}$ dettagliati per tracciare le modifiche e garantire la conformità alle politiche del progetto;
- **Evitare conflitti**: prevenire modifiche non autorizzate o che possano entrare in conflitto con il *sistema*$_{\text{G}}$;
- **Mantenere la qualità**: garantire che le modifiche non compromettano l'integrità, la funzionalità o gli obiettivi generali del progetto.

4.2.2.3 ITS$_{\text{G}}$

Per conseguire l'obiettivo di assicurare la tracciabilità delle modifiche è necessario creare degli *audit trail*$_{\text{G}}$ dettagliati, ovvero dei registri che tracciano tutte le attività e le modifiche all'interno di un *sistema*$_{\text{G}}$. Per la creazione, la gestione ed il tracciamento di questi *audit trail*$_{\text{G}}$, PEBKAC utilizza l'*Issue Tracking System*$_{\text{G}}$ *Jira*$_{\text{G}}$, sviluppato da *Atlassian*$_{\text{G}}$.

4.2.2.3.1 Ticket

Un *ticket*$_{\text{G}}$ è una voce che rappresenta una singola attività, problema, richiesta o *task*$_{\text{G}}$ all'interno di un progetto.
Esistono varie tipologie di *ticket*$_{\text{G}}$:

- **Task$_{\text{G}}$**: questa tipologia di *ticket*$_{\text{G}}$ rappresenta una comune attività che deve essere completata all'interno del progetto;
- **Sub-task**: questa tipologia di *ticket*$_{\text{G}}$ rappresenta una parte di un *ticket*$_{\text{G}}$ più grande (come un *task*$_{\text{G}}$) che viene suddivisa in azioni più piccole e gestibili;
- **Story**: questa tipologia di *ticket*$_{\text{G}}$ rappresenta un *requisito*$_{\text{G}}$ ed è generalmente scritto in un formato che descrive il risultato atteso dal punto di vista dell'utente;
- ***bug*$_{\text{G}}$**: questa tipologia di *ticket*$_{\text{G}}$ rappresenta un errore o difetto nel *sistema*$_{\text{G}}$ che necessita di correzione;

Ogni *ticket*$_{\text{G}}$ è dotato di campi per riportare i dettagli relativi all'attività, al problema, alla richiesta o alla *task*$_{\text{G}}$ che rappresenta:

- **Summary**: un riassunto breve in una sola riga del *ticket*$_{\text{G}}$;
- **Key**: un identificatore unico per ogni *ticket*$_{\text{G}}$, nella forma si SW-Key;
- **Epic**: *epic*$_{\text{G}}$ a cui il *ticket*$_{\text{G}}$ è associato;
- **Links**: un elenco di link a *ticket*$_{\text{G}}$ correlati;
- **Assignee**: la persona o le persone a cui il *ticket*$_{\text{G}}$ è attualmente assegnato;
- **Description**: una descrizione dettagliata del *ticket*$_{\text{G}}$;
- **Due**: la data entro cui questo *ticket*$_{\text{G}}$ è programmato per essere completato;
- **Reporter**: la persona che ha inserito il *ticket*$_{\text{G}}$ nel *sistema*$_{\text{G}}$;
- **Links**: un elenco di link alle *commit*$_{\text{G}}$ e alle *pull request*$_{\text{G}}$ effettuati nella *repository*$_{\text{G}}$ di *GitHub*$_{\text{G}}$ correlate al ticket;

- **Status**: la fase in cui si trova attualmente il *ticket*$_{\text{G}}$ nel suo ciclo di vita, che può essere "To Do", "In Process", "Verify" ed infine "Approve & Release";
- **Sprint**: *sprint*$_{\text{G}}$ a cui il *ticket*$_{\text{G}}$ è associato;
- **Fix Version**: la versione del progetto in cui il *ticket*$_{\text{G}}$ è stato (o sarà) risolto;
- **Priority**: l'importanza del *ticket*$_{\text{G}}$ rispetto ad altri ticket.

4.2.2.3.2 Epic

Un'*epic*$_{\text{G}}$ è una raccolta di *ticket*$_{\text{G}}$ che rappresenta uno degli obiettivi più ampi e significativi verso cui è diretto l'intero progetto. Si tratta di un concetto che aiuta a gestire e strutturare il lavoro più complesso, suddividendolo in parti più piccole e gestibili. Le *epic*$_{\text{G}}$ sono utili per monitorare i progressi rispetto a funzionalità che richiedono tempo o che coinvolgono diverse aree del progetto. Le *epic*$_{\text{G}}$ sono particolarmente utili nei processi *Agile*$_{\text{G}}$, poiché offrono una visione a lungo termine del progetto, anche mentre si adatta e si pianifica in modo incrementale, fornendo strumenti di tracciamento, come una scorebord che presenta le percentuali di *ticket*$_{\text{G}}$ presenti in ogni stato, che monitorano lo stato generale di ogni *epic*$_{\text{G}}$, misurano i progressi e identificano eventuali ritardi. Inoltre, un'*epic*$_{\text{G}}$, oltre ai *ticket*$_{\text{G}}$ che comprende, possiede tutti i campi precedentemente elecati per i ticket.

4.2.2.3.3 Versioni

Le versioni sono la modalità di organizzazione, pianificazione e monitoraggio del lavoro in base alle specifiche *milestone*$_{\text{G}}$ di un progetto. Ogni versione ha a che fare con le funzionalità, rappresentate da *epic*$_{\text{G}}$ e relativi *ticket*$_{\text{G}}$ ad essa associati, da realizzare entro una scadenza. In pratica, permette di sapere chiaramente quali *requisito*$_{\text{G}}$ devono essere soddisfatti per la specifica *milestone*$_{\text{G}}$ che rappresenta. Ciò rende semplice la tracciabilità dei progressi di ciascuna versione, oltre a ritardi o modifiche, usando una *scoreboard*$_{\text{G}}$ che adotta la stessa logica di quella utilizzata dalle *epic*$_{\text{G}}$. In seguito al completamento di tutti i *ticket*$_{\text{G}}$ associati ad una determina versione, questa può essere rilasciata. Inoltre, una versione, oltre ai *ticket*$_{\text{G}}$ che comprende, possiede dei campi che specificano la data di inizio, la data di fine ed una breve descrizione.

4.2.2.3.4 Backlog e Sprint

In *Jira*$_{\text{G}}$ sono integrati diversi strumenti per lo sviluppo secondo il metodo *Agile*$_{\text{G}}$, tra i quali è importante evidenziare *backlog*$_{\text{G}}$ e *sprint*$_{\text{G}}$.
Il *backlog*$_{\text{G}}$ contiene una lista di *ticket*$_{\text{G}}$ da completare dal team, ed è ordinata in base alla priorità: i più importanti sono posti in cima, mentre i meno importanti sono disposti verso il fondo. La lista non è statica, ma è uno spazio dinamico all'interno del quale il team può aggiungere, eliminare, aggiornare e riorganizzare le priorità dei *ticket*$_{\text{G}}$ al suo interno. Il *backlog*$_{\text{G}}$ è inteso come punto di partenza per pianificare il lavoro: prima di ogni *sprint*$_{\text{G}}$, il team esamina il *backlog*$_{\text{G}}$ per selezionare il lavoro da svolgere durante quello *sprint*$_{\text{G}}$.
Uno *sprint*$_{\text{G}}$ è un periodo di tempo predefinito in cui vengono completati i *ticket*$_{\text{G}}$ selezionati dal *backlog*$_{\text{G}}$ prima del suo inizio. Ogni *sprint*$_{\text{G}}$ è dotato di una data di inizio, una data di fine e di uno stato, che può essere "In Corso" o "Terminato".

4.2.2.3.5 Timeline

La *timeline*$_{\text{G}}$ messa a diposizione in *Jira*$_{\text{G}}$ è uno strumento realizzato tramite un *diagramma di Gantt*$_{\text{G}}$ che aiuta a gestire le scadenze, le dipendenze e l'andamento del progetto, fornendo una panoramica d'insieme dello stato di avanzamento.
Essa mostra tutti i *ticket*$_{\text{G}}$ associati ad una *epic*$_{\text{G}}$ che sono stati inseriti al suo interno, evidenziandone le date di inizio e fine e le dipendenze con altri ticket. Ogni *ticket*$_{\text{G}}$ viene visualizzato come un blocco che si estende lungo la *timeline*$_{\text{G}}$ in base alla durata prevista. Le dipendenze tra i vari possono essere visualizzate tramite linee di collegamento, mostrando come il completamento di un'attività dipenda da un'altra. Inoltre, la *timeline*$_{\text{G}}$ mostra chiamente anche lo stato delle attività, ovvero quali attività sono in corso, quali attività sono state completate e quali attività sono in ritardo.
Un'altra informazione mostrata nella *timeline*$_{\text{G}}$ sono le versioni e, in particolare, quando sono state fissate le loro date di scadenza. Le versioni sono rappresentate graficamente come delle linee verticali posizionate proprio sulla data di scadenza corrispondente.
É inoltre possibile visualizzare gli *sprint*$_{\text{G}}$ definiti all'interno di *Jira*$_{\text{G}}$ nell'area superiore della *timeline*$_{\text{G}}$, permettendo così di determinare quali attività sono state svolte durante ciascuno *sprint*$_{\text{G}}$.
Infine, è utile notare che nella *timeline*$_{\text{G}}$ e possibile effettuare delle operazioni di filtraggio dei *ticket*$_{\text{G}}$ visualizzati, permettendo così di visualizzare anche l'organizzazione di specifici gruppi di attività.

4.2.2.4 Pull Request

Le *pull request*$_{\text{G}}$ sono un meccanismo per la gestione controllata delle modifiche nei rilasci del prodotto. Quando un membro del team propone modifiche al *repository*$_{\text{G}}$, esse vengono raggruppate in una *pull request*$_{\text{G}}$, che consente ai verificatori di analizzarle prima di integrarle nel *repository*$_{\text{G}}$.
Il processo può essere riassunto nei seguenti passaggi:

- **Creazione della *pull request*$_{\text{G}}$**: Non appena le modifiche vengono apportate in un *branch*$_{\text{G}}$ dedicato, il contributore apre una *pull request*$_{\text{G}}$ per proporre la loro integrazione nel *branch*$_{\text{G}}$ develop o in un altro *branch*$_{\text{G}}$ di destinazione;

- **Revisione**: I verificatori esaminano le modifiche proposte direttamente all'interno della *pull request*$_{\text{G}}$, visualizzando i dettagli di ciò che è stato modificato e avendo la possibilità di commentare su una singola riga o su più righe/sezioni. Se le modifiche richieste sono minime, il *Verificatore*$_{\text{G}}$ può correggerle direttamente eseguendo una *commit*$_{\text{G}}$ sul *branch*$_{\text{G}}$ in cui sono state suggerite le modifiche;

- **Feedback$_{\text{G}}$ e suggerimenti**: Se le modifiche sono significative, i verificatori solitamente forniscono suggerimenti per miglioramenti, evidenziano possibili errori ed elencano tutte le modifiche necessarie per allineare il lavoro agli standard e alle linee guida del progetto;

- **Applicazione delle correzioni**: Il contributore che ha creato la *pull request*$_{\text{G}}$ risponde al *feedback*$_{\text{G}}$ ricevuto ed apporta le modifiche necessarie. Successivamente, la *pull request*$_{\text{G}}$ viene aggiornata con le modifiche revisionate;

- **Decisione finale**: Quando i verificatori approvano la proposta, o quando tutte le questioni sollevate da questi ultimi sono state risolte, la *pull request*$_{\text{G}}$ può essere

considerata accettata e viene "unita" al *branch*$_{\text{G}}$ di destinazione, garantendo che solo contributi di alta qualità facciano parte del progetto.

4.2.3 Configuration status accounting

4.2.3.1 Scopo

Il *configuration status accounting*$_{\text{G}}$ è il processo di *documentazione*$_{\text{G}}$ e monitoraggio di verifiche e cambiamenti alle caratteristiche del prodotto *software*$_{\text{G}}$. Grazie ad esso si ha una visione complessiva della sua evoluzione e della sua aderenza ai *requisito*$_{\text{G}}$ e agli standard.

4.2.3.2 Version control

Il *sistema*$_{\text{G}}$ di version control adottato dal gruppo per tracciare l'evoluzione del *software*$_{\text{G}}$ segue la seguente convenzione di numerazione delle versioni: [x].[y].[z]([build]). Si tratta di un *sistema*$_{\text{G}}$ di versionamento a quattro componenti, ognuna delle quali ha un significato specifico:

- **x**: Rappresenta la versione principale del *software*$_{\text{G}}$, e il suo valore viene incrementato per ogni fase significativa di revisione o avanzamento del progetto;
- **y**: Rappresenta cambiamenti significativi o aggiunte di nuove funzionalità, e il suo valore viene incrementato ogni volta che vengono apportati cambiamenti considerati rilevanti per il prodotto;
- **z**: Rappresenta piccole modifiche o correzioni di *bug*$_{\text{G}}$, e viene incrementato quando vengono svolte azioni come l'aggiornamento della *documentazione*$_{\text{G}}$ o la correzione di errori minori;
- **build**: Indica il numero delle build eseguite per una determinata versione, e viene incrementato ogni volta che vengono apportate delle modifiche alla *documentazione*$_{\text{G}}$.

Il *sistema*$_{\text{G}}$ prevede che la numerazione inizi dalla versione "0.0.1(0)". Ogni volta che il valore di x, y o z aumenta, tutti i valori alla sua destra vengono resettati a "0".

4.2.4 Configuration evaluation

4.2.4.1 Scopo

Il processo di configuration evaluation serve a garantire la correttezza, la coerenza e la conformità della configurazione del *sistema*$_{\text{G}}$, del *software*$_{\text{G}}$ e dell'infrastruttura rispetto ai *requisito*$_{\text{G}}$ definiti.

4.2.4.2 Tracciamento dei requisiti

Il team PEBKAC ha deciso di tracciare i *requisiti*$_{\text{G}}$ direttamente nel codice del prodotto *software*$_{\text{G}}$. Questo approccio offre un collegamento diretto e verificabile tra i *requisiti*$_{\text{G}}$ di progettazione e il codice che soddisfa tali *requisiti*$_{\text{G}}$. Il tracciamento viene effettuato includendo un commento specifico prima di ogni blocco di codice che implementa un determinato *requisito*$_{\text{G}}$. Il commento contiene l'ID univoco del *requisito*$_{\text{G}}$, in modo da facilitare l'associazione tra i *requisiti*$_{\text{G}}$ e le corrispondenti implementazioni nel codice.

4.2.5 Release Management

4.2.5.1 Scopo

Il processo di *release management*$_{\text{G}}$ serve a pianificare, coordinare e gestire il rilascio, per far sì che qualsiasi nuova versione di un prodotto venga distribuita dal *sistema*$_{\text{G}}$ in modo controllato.

4.2.5.2 Automazione compilazione documenti

Il gruppo si è dotato di una *GitHub Action*$_{\text{G}}$ che provvede all'*automazione*$_{\text{G}}$ della generazione e trasferimento di documenti *LaTeX*$_{\text{G}}$ in PDF.
Essa si attiva quando viene unita una *pull request*$_{\text{G}}$ nel *branch*$_{\text{G}}$ develop e funziona nel seguente modo:

1. Prepara l'ambiente ed installa gli strumenti necessari;
2. Esplora la directory contenente i documenti *LaTeX*$_{\text{G}}$;
3. Converte eventuali immagini in PDF e compila i file *LaTeX*$_{\text{G}}$;
4. Carica i PDF generati in una cartella organizzata;
5. Trasferisce la cartella sul *branch*$_{\text{G}}$ main;
6. Sposta i PDF all'interno della cartella corretta nel *branch*$_{\text{G}}$ main mediante un *commit*$_{\text{G}}$ automatico.

Il componente del gruppo che si è occupato della redazione di un documento dovrà semplicemente occuparsi di aprire una *pull request*$_{\text{G}}$ verso il *branch*$_{\text{G}}$ develop. Essa verrà verificata dal membro del gruppo che ricopre attualmente il *ruolo*$_{\text{G}}$ di *Verificatore*$_{\text{G}}$ e, in seguito, verrà approvata dal membro del gruppo che ricopre attualmente il *ruolo*$_{\text{G}}$ di *responsabile*$_{\text{G}}$, che, infine, chiuderà la *pull request*$_{\text{G}}$, eseguendo l'*automazione*$_{\text{G}}$.
Questa *automazione*$_{\text{G}}$ permette di eliminare il lavoro manuale ripetitivo e permette al team di concentrarsi sul contenuto invece che sulla gestione tecnica dei file.

4.3 Accertamento di qualità

4.3.1 Scopo

L'accertamento della qualità ha l'obiettivo di prevenire i difetti nel prodotto *software*$_{\text{G}}$, garantendo la conformità ai processi adottati. Piuttosto che limitarsi a individuare errori a posteriori, questo approccio assicura che ogni fase dello sviluppo venga eseguita correttamente, monitorando l'applicazione delle best practice in modo non intrusivo attraverso strumenti di controllo. Questo processo può fare uso anche dei risultati di altri processi di supporto, come ad esempio del processo di *verifica*$_{\text{G}}$ e del processo di *validazione*$_{\text{G}}$.

4.3.2 Ciclo di Deming

Il Ciclo di Deming, noto anche come Shewhart-Deming's Learning-and-Quality Cycle, è un modello a quattro fasi ideato intorno al 1950 per introdurre miglioramenti specifici nei processi. Fondamentale per il miglioramento continuo, il ciclo si articola in:

- **Pianificare (Plan)**: definizione delle attività, delle scadenze, delle responsabilità e delle *risorse*$_{\text{G}}$ necessarie per raggiungere obiettivi di miglioramento. Questa fase non riguarda la pianificazione di un progetto, ma l'organizzazione di azioni mirate all'ottimizzazione di un processo;
- **Eseguire (Do)**: implementazione delle attività pianificate, anche in modo esplorativo. Non si tratta dello sviluppo di un prodotto, ma dell'attuazione concreta delle azioni di miglioramento;
- **Valutare (Check)**: analisi dei risultati ottenuti per verificare se le azioni intraprese hanno prodotto gli effetti desiderati;
- **Agire (Act)**: consolidamento delle soluzioni efficaci, aggiornando il *way of working*$_{\text{G}}$, e individuazione di ulteriori opportunità di miglioramento.

Questo ciclo aiuta il gruppo a perfezionare continuamente la qualità del prodotto, garantendo che vengano raggiunti gli obiettivi di qualità che ci si era prefissati inizialmente e impedendo un peggioramento della stessa.

4.3.3 Metriche

Le metriche di qualità sono gli strumenti utilizzati per valutare la qualità di un prodotto *software*$_{\text{G}}$ o di un processo di sviluppo. Servono a misurare il grado di conformità agli standard, identificare aree di miglioramento e garantire che il *software*$_{\text{G}}$ soddisfi i requisiti attesi.
Si suddividono in diverse categorie, tra cui:

- **Metriche di processo**: valutano l'efficienza dello sviluppo;
- **Metriche di prodotto**: misurano le caratteristiche del *software*$_{\text{G}}$.

Le metriche di qualità utilizzate in questo progetto sono riportate alla sezione §6. Esse vengono rappresentate dal loro nome in lingua inglese, da un'abbreviazione, che genralmente consiste nella sequenza delle prime lettere delle parole del nome della metrica stessa, tutte in maiuscolo, e da una descrizione, che ne spiega brevemente il significato.

4.3.4 Obiettivi di qualità

Gli obiettivi di qualità sono riportati nel Piano di Qualifica e, come le metriche, sono suddivisi in obiettivi di qualità di processo e di prodotto. Essi sono strutturati nel seguente modo:

- **Metrica**: l'abbreviazione del nome della metrica;
- **Descrizione**: il nome in lingua inglese;
- **Valore accettabile**: valore per cui la metrica è da considerare soddisfatta, nonostante ci sia spazio per un miglioramento;
- **Valore ideale**: valore per cui la metrica viene considerata pienamente soddisfatta.

4.3.5 Strumenti

Gli strumenti utilizzati per il processo di *accertamento di qualità*$_{\text{G}}$ sono:

- ??;
- Jira.

4.4 Verifica

4.4.1 Scopo

Lo scopo del processo di *verifica*$_{\text{G}}$ è fornire evidenza oggettiva che le uscite di un particolare segmento dello sviluppo *software*$_{\text{G}}$ soddisfino tutti i requisiti specificati per esso. In altre parole, la *verifica*$_{\text{G}}$ si concentra sull'accertarsi che lo sviluppo stia costruendo il *sistema*$_{\text{G}}$ correttamente. La *verifica*$_{\text{G}}$ mira a ricercare la coerenza, la completezza e la correttezza di queste uscite. Essa fornisce inoltre supporto per la successiva conclusione che il *software*$_{\text{G}}$ sia validato.

4.4.2 Descrizione

Questo processo viene effettuato almeno una volta prima del rilascio ufficiale di un prodotto in uno dei *branch*$_{\text{G}}$ principali del *repository*$_{\text{G}}$. La *verifica*$_{\text{G}}$ è affidata ai verificatori, che per garantire un'analisi obiettiva ed efficace, non devono essere gli stessi che hanno partecipato allo sviluppo del prodotto in questione.

4.4.3 Analisi statica

L'analisi statica è una forma di *verifica*$_{\text{G}}$ del *software*$_{\text{G}}$ che non richiede l'esecuzione del codice. Invece, esamina il codice sorgente, il codice oggetto o la *documentazione*$_{\text{G}}$ per accertare la conformità a regole, l'assenza di difetti e la presenza di proprietà desiderate. Questa forma di *verifica*$_{\text{G}}$ viene utilizzata anche per i documenti testuali.
L'analisi statica include tecniche che si basano sulla revisione manuale dell'oggetto di verifica, che può essere codice sorgente, *documentazione*$_{\text{G}}$ o altri artefatti del processo di sviluppo. I due metodi di lettura utilizzati sono il walkthrough e l'inspection.
Nelle fasi iniziali è stato adottato l'approccio walkthrough. Tuttavia, man mano che acquisisce esperienza, il team potrà passare all'ispezione, rendendo il processo più rapido e ottimizzando l'uso delle *risorse*$_{\text{G}}$.

4.4.3.1 Walkthrough

Il walkthrough si basa su una lettura critica ad ampio spettro dell'oggetto in esame, che può essere codice sorgente, *documentazione*$_{\text{G}}$ di progetto, specifiche o altri artefatti del processo di sviluppo. Esso coinvolge tipicamente gruppi misti composti da autori e verificatori, con ruoli distinti tra i partecipanti, dato che questa eterogeneità di prospettive aiuta a identificare un'ampia gamma di problemi.
Il processo tipico di un walkthrough si articola in diverse fasi:

1. **Pianificazione**: gli autori e i verificatori si coordinano per organizzare la sessione di walkthrough;

2. **Lettura**: i verificatori esaminano l'oggetto in esame individualmente prima della sessione, familiarizzando con il materiale. Durante la sessione, uno dei partecipanti può "guidare" la lettura, illustrando passo dopo passo il codice o il documento;

3. **Discussione**: durante la sessione, i verificatori sollevano dubbi, pongono domande e identificano potenziali difetti o aree di incertezza. Autori e verificatori discutono i punti critici;

4. **Correzione dei difetti**: dopo la sessione, la responsabilità di correggere i difetti identificati ricade sugli autori;

5. **Documentazione**: ogni passo del walkthrough, incluse le attività svolte e i difetti identificati, viene documentato.

4.4.3.2 Ispezione

L'ispezione è caratterizzata da un esame focalizzato su presupposti, che utilizza liste di controllo predefinite che guidano l'esame verso aree specifiche dove è più probabile trovare difetti, rendendo così non necessaria la completa lettura del prodotto in questione.
Il processo tipico di un'ispezione si articola in diverse fasi:

1. **Pianificazione**: viene organizzata la sessione di ispezione e vengono identificati i partecipanti;

2. **Definizione lista di controllo**: viene creata o selezionata una lista di controllo specifica per l'oggetto di verifica. Questa lista elenca gli aspetti specifici da esaminare selettivamente;

3. **Lettura**: i verificatori esaminano l'oggetto di *verifica*$_{\text{G}}$ individualmente, guidati dalla lista di controllo, annotando i potenziali difetti riscontrati;

4. **Correzione dei difetti**: dopo la fase di ispezione, la responsabilità di correggere i difetti identificati ricade sugli autori dell'oggetto;

5. **Documentazione**: ogni passo dell'ispezione, inclusa la lista di controllo utilizzata e i difetti identificati, viene documentato.

4.4.4 Analisi dinamica

L'analisi dinamica è una forma di *verifica*$_{\text{G}}$ del *software*$_{\text{G}}$ che, a differenza dell'analisi statica, richiede l'esecuzione del codice per osservarne il comportamento ed evidenziare eventuali difetti. Questo processo prevede l'esecuzione di parti del *software*$_{\text{G}}$ o dell'intero *sistema*$_{\text{G}}$, permettendo di valutarne il funzionamento su un insieme finito di casi di prova. Ogni caso di prova specifica i valori di ingresso, lo stato iniziale previsto e l'effetto atteso, che funge da riferimento per determinare l'esito del test.
L'analisi dinamica si realizza principalmente attraverso l'esecuzione di test, spesso automatizzati, per verificare la correttezza del *software*$_{\text{G}}$ e identificare possibili errori. Questa tecnica è generalmente integrata con l'analisi statica, che esamina il codice senza eseguirlo, fornendo un quadro più completo della qualità del *software*$_{\text{G}}$. Esistono diverse tipologie di *test*$_{\text{G}}$ nell'analisi dinamica, ciascuna con obiettivi specifici: i *test*$_{\text{G}}$ di unità, i *test*$_{\text{G}}$ di integrazione e i *test*$_{\text{G}}$ di *sistema*$_{\text{G}}$. Un aspetto cruciale dell'analisi dinamica è la ripetibilità e l'*automazione*$_{\text{G}}$ dei test, che consentono di eseguire verifiche sistematiche,

ridurre i tempi di analisi e garantire coerenza nei risultati. L'*automazione*$_{\text{G}}$ permette inoltre di eseguire i *test*$_{\text{G}}$ in modo efficiente, ottimizzando il processo di *verifica*$_{\text{G}}$ del *software*$_{\text{G}}$.

4.5 Validazione

4.5.1 Scopo

Lo scopo del processo di *validazione*$_{\text{G}}$ è di confermare, attraverso esami e fornitura di evidenze oggettive, che il prodotto *software*$_{\text{G}}$ finale soddisfa i requisiti stabiliti e riportati nell'Analisi dei Requisiti.

4.5.2 Esecuzione del processo

Il processo di *validazione*$_{\text{G}}$ viene eseguito in presenza del proponente dopo aver avuto un esito positivo dai *test*$_{\text{G}}$ di unità, di integrazione e di *sistema*$_{\text{G}}$. Lo svolgimento di questo processo è suddiviso in due fasi:

1. **Verifica del tracciamento dei requisiti**: il team presenta all'azienda proponente l'analisi del tracciamento dei requisiti, dimostrando così che ogni *requisito*$_{\text{G}}$ è stato effettivamente integrato nel prodotto sottoposto a *validazione*$_{\text{G}}$;
2. **Fase di collaudo**: in collaborazione con l'azienda proponente, il team procede all'esecuzione del prodotto. Durante questa fase, vengono effettuati i *test*$_{\text{G}}$ di accettazione per verificare la conformità del *sistema*$_{\text{G}}$ rispetto alle specifiche richieste e, se tutti i *test*$_{\text{G}}$ di accettazione eseguiti forniscono un esito positivo, il proponente considererà il prodotto valido.

4.5.3 Test di accettazione

- **Redazione**: i *test*$_{\text{G}}$ di accettazione sono stabiliti in contemporanea alla redazione dell'Analisi dei Requisiti e vengono svolti alla presenza del *committente*$_{\text{G}}$;
- **Descrizione**: i *test*$_{\text{G}}$ di accettazione hanno come obiettivo principale di verificare che il prodotto soddisfi i requisiti utente, cioè le aspettative del *committente*$_{\text{G}}$ così come definite nel *capitolato*$_{\text{G}}$. Questa attività si svolge in modo formale alla presenza del *committente*$_{\text{G}}$, il quale supervisiona o esegue direttamente i casi di prova. Tali casi sono derivati dai requisiti utente e mirano a dimostrare la conformità del *software*$_{\text{G}}$ alle specifiche concordate. Il buon esito di questa fase è determinante per l'accettazione finale del prodotto e ne consente il rilascio ufficiale.

4.6 Revisione congiunta con il proponente

4.6.1 Scopo

Lo scopo del processo di revisione congiunta con il proponente è garantire che il prodotto *software*$_{\text{G}}$ in sviluppo soddisfi le aspettative del proponente, coinvolgendolo direttamente nella *verifica*$_{\text{G}}$ degli artefatti. Questo processo aiuta a individuare tempestivamente difetti, malintesi sui requisiti e possibili problematiche, permettendo di apportare modifiche in tempo utile. Inoltre, favorisce la comunicazione e la collaborazione tra team e proponente, riducendo il rischio di insoddisfazione e costose rilavorazioni nelle fasi finali del progetto.

4.6.2 Implementazione

Il processo di revisione congiunta con il proponente viene implementato attraverso una serie di fasi che permettono di verificare e migliorare il prodotto *software*$_{\text{G}}$ in modo collaborativo:

- **Pianificazione della revisione**: il primo passo della revisione è definire lo scopo e gli artefatti da esaminare. Si selezionano i partecipanti, tra cui membri del team di sviluppo, rappresentanti del proponente e *stakeholder*$_{\text{G}}$, e si organizzano i dettagli dell'incontro. Infine, i materiali vengono preparati e condivisi per l'analisi;
- **Preparazione**: per un'efficace revisione, i partecipanti ricevono in anticipo la *documentazione*$_{\text{G}}$ da analizzare. Vengono assegnati ruoli chiave: il moderatore guida la discussione, i revisori esaminano gli artefatti e un segretario registra le osservazioni. Si definiscono inoltre criteri di valutazione come coerenza, chiarezza e qualità del codice;
- **Esecuzione della revisione**: durante l'incontro, il team di sviluppo presenta gli artefatti al proponente, che fornisce *feedback*$_{\text{G}}$ su eventuali criticità. Tutti i commenti vengono raccolti e documentati per un'analisi successiva;
- **Analisi dei risultati e azioni correttive**: dopo la revisione, il team classifica i problemi per priorità e definisce le azioni correttive. Viene redatto un report con i risultati e le modifiche necessarie, che viene condiviso con il proponente per garantire trasparenza e allineamento alle aspettative;
- **Follow-up e validazione**: dopo le modifiche, si può fare un'ulteriore revisione per verificare la risoluzione dei problemi. L'obiettivo è ottenere la conferma del proponente e ridurre il rischio di rilavorazioni future;

4.6.3 Strumenti

Gli strumenti utilizzati per il processo di revisione congiunta con il proponente sono:

- Google Calendar;
- Microsoft Teams.

4.7 Verifiche ispettive interne

4.7.1 Scopo

Lo scopo del processo di verifiche ispettive interne è valutare se le attività e i risultati relativi alla qualità soddisfano le disposizioni pianificate e se queste sono attuate efficacemente. Le verifiche ispettive interne contribuiscono a garantire la qualità del *software*$_{\text{G}}$, valutare l'efficacia dei processi, identificare non conformità e fornire *feedback*$_{\text{G}}$ per il miglioramento continuo, supportando così la produzione di *software*$_{\text{G}}$ di alta qualità.

4.7.2 Implementazione

Per implementare il processo di verifiche ispettive interne, è necessario seguire questi passi:

- **Pianificazione**: definire l'ambito, gli obiettivi e gli artefatti da verificare, come *documentazione*$_{\text{G}}$, codice o processi. Stabilire un piano con tempistiche e *risorse*$_{\text{G}}$ necessarie;
- **Assegnazione dei ruoli**: designare i membri del team di verifica, che dovrebbero essere indipendenti dalle attività verificate. Ogni membro avrà un *ruolo*$_{\text{G}}$ specifico, come analizzare i processi, raccogliere dati e redigere il report;
- **Esecuzione della verifica**: condurre l'ispezione sistematica dei processi e dei risultati, confrontandoli con gli standard e le disposizioni pianificate. Raccogliere e documentare le non conformità o le aree di miglioramento;
- **Analisi dei risultati**: valutare i dati raccolti, classificando le non conformità e le deviazioni. Identificare le cause e le aree da migliorare;
- **Feedback$_{\text{G}}$ e azioni correttive**: redigere un report con i risultati e le raccomandazioni. Fornire un *feedback*$_{\text{G}}$ alle parti coinvolte, suggerendo azioni correttive e preventive per migliorare i processi;
- **Monitoraggio e follow-up**: verificare che le azioni correttive siano state implementate e che le problematiche siano state risolte, promuovendo il miglioramento continuo.

4.7.3 Strumenti

Gli strumenti utilizzati per il processo di verifiche ispettive interne sono:

- Discord;
- Jira.

4.8 Risoluzione dei problemi

4.8.1 Scopo

Lo scopo del processo di risoluzione dei problemi è identificare, analizzare e risolvere tempestivamente i problemi che emergono durante il ciclo di vita del *software*$_{\text{G}}$. Questo processo garantisce una gestione efficace ed efficiente dei problemi, dalla loro rilevazione fino alla completa risoluzione e verifica, contribuendo così alla qualità complessiva del prodotto *software*$_{\text{G}}$.

4.8.2 Implementazione

Il processo di risoluzione dei problemi viene implementato attraverso una serie di fasi strutturate, con l'obiettivo di identificare e risolvere tempestivamente le criticità che emergono durante lo sviluppo del *software*$_{\text{G}}$. L'implementazione di questo processo si articola nei seguenti passaggi:

- **Sviluppo di una strategia di gestione dei problemi**: definizione di un approccio strutturato per affrontare i problemi, stabilendo ruoli e responsabilità, flussi di comunicazione e strumenti da utilizzare;
- **Registrazione e classificazione dei problemi**: ogni problema rilevato viene registrato in uno storico, includendo dettagli come descrizione, data di rilevamento, origine e segnalatore. I problemi vengono poi classificati per gravità e priorità, permettendo una gestione efficace e un'analisi delle tendenze nel tempo;
- **Analisi dei problemi e individuazione delle soluzioni**: dopo la registrazione, il problema viene analizzato per identificarne le cause radice. Sulla base di questa analisi, vengono proposte e valutate soluzioni adeguate. Questa fase può coinvolgere l'analisi del codice, dei log di *sistema*$_{\text{G}}$, della *documentazione*$_{\text{G}}$ o il confronto con i membri del team;
- **Implementazione della soluzione**: dopo aver individuato la soluzione più adatta, questa viene applicata. L'intervento può riguardare modifiche al codice, configurazioni di *sistema*$_{\text{G}}$, aggiornamenti della *documentazione*$_{\text{G}}$ o altre azioni correttive;
- **Verifica dell'efficacia della correzione**: una volta implementata la soluzione, è essenziale verificare che il problema sia stato effettivamente risolto e che non siano stati introdotti nuovi difetti. Questa *verifica*$_{\text{G}}$ avviene attraverso test, revisioni o altri controlli di qualità;
- **Monitoraggio e gestione continua dei problemi**: il processo prevede un costante monitoraggio dello stato dei problemi aperti, assicurando che siano assegnati e gestiti fino alla loro completa risoluzione;

4.8.3 Strumenti

Gli strumenti utilizzati per il processo di risoluzione dei problemi sono:

- Jira.

5 Processi Organizzativi

5.1 Gestione organizzativa

5.1.1 Scopo

Lo scopo di questo processo è esporre le modalità e gli strumenti di coordinamento usati dal gruppo per la comunicazione, interna ed esterna, e normare l'assegnazione di *ruoli*$_{\mathrm{G}}$ e compiti, oltre che la gestione dei rischi.

5.1.2 Ruoli

Per ottimizzare la gestione delle attività e dei compiti da svolgere vengono definiti sei *ruoli*$_{\mathrm{G}}$ distinti, ciascuno con mansioni e responsabilità specifiche. Ogni componente del gruppo dovrà assumere ciascun *ruolo*$_{\mathrm{G}}$ per un numero di ore significativo.

5.1.2.1 Responsabile

Il *responsabile*$_{\mathrm{G}}$ è il punto di riferimento per tutto il gruppo e anche per le comunicazioni con il *committente*$_{\mathrm{G}}$ e con l'azienda proponente. Inoltre il *responsabile*$_{\mathrm{G}}$ è la figura che ha il compito di coordinare le azioni dei membri del gruppo, perciò deve avere competenze tecniche in ogni ambito del progetto. Le responsabilità di questo *ruolo*$_{\mathrm{G}}$ sono:

- Coordinamento tra gruppo ed enti esterni;
- Gestione delle comunicazioni interne;
- Pianificazione di progetto,
- Gestione dei *task*$_{\mathrm{G}}$ e delle *risorse*$_{\mathrm{G}}$;
- Gestione dell'avanzamento del progetto.

5.1.2.2 Amministratore

L'*amministratore*$_{\mathrm{G}}$ è la figura che definisce, gestisce e mantiene l'ambiente e l'infrastruttura necessari per lo sviluppo del progetto facendo in modo che siano affidabili e sicuri. Si occupa della gestione della configurazione, del versionamento, delle varie *automazioni*$_{\mathrm{G}}$ e della *documentazione*$_{\mathrm{G}}$. Si occupa di:

- Selezionare e abilitare *risorse*$_{\mathrm{G}}$ informatiche a supporto del *way of working*$_{\mathrm{G}}$;
- Gestire errori e malfunzionamenti nei meccanismi nell'infrastruttura.

5.1.2.3 Analista

La funzione dell'*analista*$_{\mathrm{G}}$ è quella di analizzare il problema per definire i *requisiti*$_{\mathrm{G}}$ del prodotto, per questo deve avere buona conoscenza del dominio del problema. L'*analista*$_{\mathrm{G}}$ raccoglie le sue produzioni nel documento Analisi dei Requisiti. Si tratta di un *ruolo*$_{\mathrm{G}}$ fondamentale all'inizio del progetto, ma la cui utilità cala nelle seguenti fasi del progetto.

5.1.2.4 Progettista

Al progettista spettano le scelte realizzative e le specifiche architetturali del prodotto. Deve avere buone competenze tecniche e tecnologiche. Durante il processo di sviluppo la sua utilità è massima, ma tende a calare dalla fase di manutenzione in poi.

5.1.2.5 Programmatore

Quello del programmatore è un *ruolo*$_{\text{G}}$ chiave nella fase di sviluppo. In particolare si occupa di :

- Codificare ciò che è stato definito dai progettisti;
- Implementare i test;
- Redigere il Manuale utente.

5.1.2.6 Verificatore

Ha il compito di verificare il lavoro degli altri e per questo deve avere competenze tecniche ed essere presente per l'intera durata del progetto. Questa figura deve controllare che tutto ciò che viene prodotto sia conforme alle norme e alle aspettative di qualità del gruppo.

5.1.3 Attività

Ogni membro del gruppo può proporre attività da svolgere, ma è compito del *responsabile*$_{\text{G}}$ stabilire la fattibilità rispetto alle *risorse*$_{\text{G}}$ usufruibili.

5.1.3.1 Pianificazione

Le attività definite devono poi essere pianificate in termini di tempo e *risorse*$_{\text{G}}$ dal *responsabile*$_{\text{G}}$, stabilendo quindi:

- La tempistica prevista per il completamento dell'attività;
- Il membro che dovrà eseguire l'attività, in base a *ruolo*$_{\text{G}}$ e *risorse*$_{\text{G}}$ disponibili;
- Il *verificatore*$_{\text{G}}$;
- Il rischio associato.

5.1.3.1.1 Strumenti

Gli strumenti utilizzati per la pianificazione sono:

- Jira.

5.1.3.2 Esecuzione

L'esecuzione delle attività avviene obbligatoriamente per mano dell'assegnatario, definito dal *responsabile*$_{\text{G}}$. L'esecuzione deve essere obbligatoriamente conforme alla *documentazione*$_{\text{G}}$ associata precedentemente redatta. L'esecutore dovrà proporre la sua soluzione con una *pull request*$_{\text{G}}$.

5.1.3.3 Revisione

La revisione dell'attività è effettuata dal *verificatore*$_{\text{G}}$ prima dell'effettivo inserimento delle modifiche nel *repository*$_{\text{G}}$ *GitHub*$_{\text{G}}$: la *pull request*$_{\text{G}}$ aperta dell'esecutore viene accettata o rifiutata, riportando le parti non valide ed eventuali accorgimenti possibili, a seconda dell'esito della *verifica*$_{\text{G}}$.

5.1.3.3.1 Strumenti

Gli strumenti utilizzati per la revisione sono:

- *GitHub*$_{\text{G}}$.

5.1.3.4 Chiusura

Solo nel caso dell'esito positivo della *verifica*$_{\text{G}}$, con l'accettazione della *pull request*$_{\text{G}}$, viene chiuso il *branch*$_{\text{G}}$ di cui è stato effettuato il *merge*$_{\text{G}}$ e l'attività viene segnata come completata.

5.1.3.5 Tracciamento orario

Il gruppo utilizza *Google Sheets*$_{\text{G}}$ per avere un foglio di calcolo condiviso in cui tenere conto del tempo speso per svolgere le attività. Ogni membro è tenuto a registrare, alla fine di ogni sessione lavorativa, il numero di ore effettive di lavoro e il *ruolo*$_{\text{G}}$ ricoperto.

5.1.3.5.1 Strumenti

Gli strumenti utilizzati per il tracciamento orario sono:

- *Google Sheets*$_{\text{G}}$.

5.1.4 Comunicazione

5.1.4.1 Comunicazioni interne

5.1.4.1.1 Comunicazioni sincrone

Le riunioni interne si svolgeranno sulla piattaforma *Slack*$_{\text{G}}$ oppure in presenza, dovranno in ogni caso decise e organizzate alcuni giorni prima per consentire la presenza di tutti i membri. In ogni riunione il gruppo predilige un approccio libero alla discussione, incentrato sulla crescita e allo scambio di opinioni.
La gestione delle riunioni interne viene affidata al *responsabile*$_{\text{G}}$ che, coadiuvato dall'*Amministratore*$_{\text{G}}$, ha il compito di:

1. Fissare data, ora e luogo della riunione;
2. Stabilire un ordine del giorno per le riunioni;
3. Fare da moderatore durante la discussione, per garantire a tutti l'opportunità di esprimersi;
4. Se necessario, comunicare con l'esterno in base alle decisioni prese dal gruppo.

5.1.4.1.2 Comunicazioni asincrone

Per le comunicazioni asincrone il gruppo utilizzerà:

- *Slack*$_{\text{G}}$: in un area di lavoro sono stati creati:
  - **Canali**: uno principale con tutti i membri e altri, alla necessità, in cui i componenti possono organizzarsi e lavorare su attività collaborative;
  - **Canvas**: uno per ogni canale in cui sia necessario, per fissare messaggi importanti, documenti e *risorse*$_{\text{G}}$ che richiedono facile accesso.
- Whatsapp: solo per comunicazioni immediate e poco formali.

5.1.4.1.3 Strumenti

Gli strumenti utilizzati per le comunicazioni interne sono:

- *Slack*$_{\text{G}}$;
- Whatsapp.

5.1.4.2 Comunicazioni esterne

5.1.4.2.1 Comunicazioni sincrone

Per quanto riguarda gli incontri con l'azienda proponente, cruciali per discutere in modo semplice e immediata di argomenti anche complessi, potranno essere in presenza (presso la sede R&D di Vimar S.p.A.) oppure da remoto sulla piattaforma *Microsoft Teams*$_{\text{G}}$. Per quanto riguarda le riunioni con l'azienda proponente, anche chiamate SAL - Stato di Avanzamento Lavori.

- Il gruppo ha concordato con l'azienda un calendario di incontri bisettimanali della durata di 60 minuti fino alla prima revisione, poi settimanali della durata di 30 minuti;
- Il gruppo si impegna a presenziare in maniera assidua agli incontri, segnalando per tempo eventuali assenze o modifiche a quanto precedentemente concordato;
- Il gruppo si impegna a redigere un verbale per ogni incontro per documentarne il contenuto e farlo approvare all'azienda.

5.1.4.2.2 Comunicazioni asincrone

Per le comunicazioni asincrone con il proponente o altri soggetti esterni vengono utilizzati:

- **Microsoft Teams**$_{\text{G}}$: per domande corte o piccoli chiarimenti si potrà usare la chat condivisa creata dall'azienda;
- **Posta elettronica**: per comunicare con soggetti esterni e con l'azienda proponente per domande articolate o modifiche agli appuntamenti fissati si utilizzerà la mail del gruppo: pebkacswe@gmail.com.

5.1.4.2.3 Strumenti

Gli strumenti utilizzati per le comunicazioni esterne sono:

- *Microsoft Teams*$_{\text{G}}$;
- Google Gmail.

5.1.4.3 Norme comportamentali

I membri del gruppo, per garantire il rispetto delle norme e degli altri membri del gruppo, sono obbligati a:

- Essere sempre puntuali o almeno comunicare tempestivamente al *responsabile*$_{\text{G}}$ eventuali problemi;
- Partecipare attivamente alla discussione;
- Mantenere un atteggiamento rispettoso, disciplinato, aperto alle discussione e disponibile.

Inoltre, per le riunioni SAL con l'azienda proponente i membri hanno concordato di rispettare le seguenti regole:

- Tenere i telefoni spenti o silenziati (a meno di particolati motivi);
- Non utilizzare PC o Tablet, fatta eccezione per il *responsabile*$_{\text{G}}$ o chi deve raccogliere degli appunti.

5.1.4.4 Moderazione

La gestione della comunicazione del gruppo di lavoro ha un *ruolo*$_{\text{G}}$ fondamentale nel garantire interazioni efficaci, costruttive e orientate agli obiettivi. Il *responsabile*$_{\text{G}}$, che assume il *ruolo*$_{\text{G}}$ di moderatore, si occupa di:

- Facilitare le discussioni;
- Assicurare che tutti i membri abbiano l'opportunità di esprimersi;
- Mantenere il focus sugli argomenti rilevanti;
- Aiutare a gestire il tempo degli incontri;
- Sintetizzare i punti chiave;
- Promuovere una comunicazione chiara e collaborativa.

5.1.4.5 Gestione dei rischi

La gestione dei rischi in un progetto *software*$_{\text{G}}$ è tipicamente responsabilità del *responsabile*$_{\text{G}}$ e dell'*amministratore*$_{\text{G}}$. I risultati delle attività identificazione, analisi, pianificazione e controllo dei rischi sono inseriti all'interno del Piano di Progetto.

5.1.4.5.1 Nomenclatura

Per documentare i rischi viene utilizzata la notazione seguente:

R[**Tipologia**][**Numero**]

Tipologia indica la tipologia del rischio:

- **T**: tecnologio;
- **O**: organizzativo;
- **G**: interno al gruppo;

Numero è un numero univoco progressivo correlato alla tipologia;

5.1.4.5.2 Descrizione

La definizione dei rischi presenta le seguenti informazioni:

- **Id. Rischio**: codice identificativo assegnato al rischio;
- **Rischio**: nome completo del rischio;
- **Descrizione**: spiegazione dettagliata del rischio;
- **Pericolosità**: livello di gravità dell'impatto che il rischio potrebbe avere sul progetto;
- **Occorrenza**: probabilità che il rischio si verifichi;
- **Piano di intervento**: strategie e azioni previste per prevenire, mitigare o gestire il rischio nel caso in cui si manifesti.

5.2 Gestione dell'Infrastruttura

5.2.1 Scopo

Lo scopo del processo di gestione delle infrastrutture è garantire la disponibilità, la stabilità e l'efficienza degli strumenti informatici a supporto dei processi di progetto.

5.2.2 Implementazione

L'implementazione dell'infrastruttura è il processo di creazione e configurazione dell'ambiente *hardware*$_{\text{G}}$ e *software*$_{\text{G}}$ necessario per supportare lo sviluppo, il *test*$_{\text{G}}$ e l'operatività di un *sistema*$_{\text{G}}$ *software*$_{\text{G}}$. Questa fase è essenziale per fornire al team di sviluppo le *risorse*$_{\text{G}}$ adeguate per lavorare in modo efficiente e senza ostacoli tecnici. Il *ruolo*$_{\text{G}}$ centrale in questa attività è ricoperto dall'*amministratore*$_{\text{G}}$.
Le attività principali dell'implementazione dell'infrastruttura includono:

1. **Selezione e messa in opera delle *risorse*$_{\text{G}}$ informatiche**: il primo passo consiste nell'identificazione dei requisiti infrastrutturali del progetto, considerando il *way of working*$_{\text{G}}$ adottato dal team. L'*amministratore*$_{\text{G}}$ sceglie le componenti *hardware*$_{\text{G}}$ e *software*$_{\text{G}}$, configurando la rete e assicurando la loro corretta installazione e operatività;

2. **Definizione e controllo dell'ambiente**: vengono stabilite configurazioni standard per garantire coerenza, efficienza e sicurezza. Inoltre, il monitoraggio continuo dell'infrastruttura assicura il rispetto delle configurazioni e previene eventuali malfunzionamenti;

3. **Analisi degli aspetti critici:** ogni scelta viene valutata considerando diversi fattori chiave:

   - **Funzionalità**: le modifiche devono portare un miglioramento, introducendo nuove capacità o affinando quelle esistenti. È essenziale prevenire malfunzionamenti o comportamenti indesiderati;
   - **Performance**: ogni intervento deve mantenere o migliorare le prestazioni dell'infrastruttura senza compromettere l'efficienza;
   - **Sicurezza**: nessuna modifica deve ridurre il livello di protezione dei dati e delle $risorse_{\mathrm{G}}$;
   - **Disponibilità**: la configurazione deve essere applicabile su tutti i dispositivi, evitando discrepanze tra le infrastrutture;
   - **Vincoli di spazio**: l'infrastruttura deve rispettare le limitazioni di archiviazione dei dispositivi utilizzati;
   - **Compatibilità hardware$_{\mathrm{G}}$**: le scelte devono essere compatibili con le $risorse_{\mathrm{G}}$ fisiche disponibili;
   - **Costi**: viene data priorità a soluzioni open-source o gratuite rispetto a quelle a pagamento, ove possibile;
   - **Tempistiche**: l'installazione e la configurazione devono essere rapide per non rallentare il progetto.

4. **Documentazione delle procedure:** se la $verifica_{\mathrm{G}}$ ha esito positivo, l'$amministratore_{\mathrm{G}}$ redige una guida dettagliata per l'installazione e la configurazione dell'infrastruttura;

5. **Comunicazione al team:** una volta completata la $documentazione_{\mathrm{G}}$, i membri del team vengono informati sulle procedure da seguire e devono implementarle autonomamente;

6. **Supporto in caso di difficoltà:** se un componente riscontra problemi durante l'installazione o la configurazione, l'$amministratore_{\mathrm{G}}$ fornisce assistenza diretta per risolvere eventuali problematiche;

7. **Sessioni di supporto aggiuntive:** nel caso di procedure particolarmente complesse, il $responsabile_{\mathrm{G}}$ del progetto, in accordo con l'$amministratore_{\mathrm{G}}$, può organizzare incontri specifici per spiegare dettagliatamente le operazioni richieste;

5.2.3 Manutenzione

La manutenzione dell'infrastruttura è un'attività essenziale per garantire un ambiente di lavoro stabile, sicuro ed efficiente. Questa responsabilità è affidata all'$Amministratore_{\mathrm{G}}$. Le principali attività di manutenzione dell'infrastruttura comprendono:

- **Monitoraggio e gestione dell'ambiente**: l'*amministratore*$_{\text{G}}$ si occupa della configurazione, controllo e manutenzione dell'infrastruttura, assicurando che *hardware*$_{\text{G}}$, *software*$_{\text{G}}$ e reti siano sempre operativi e aggiornati per supportare al meglio le esigenze del team;
- **Gestione delle risorse informatiche**: la manutenzione non si limita alla configurazione iniziale, ma prevede un monitoraggio continuo delle prestazioni e della capacità delle *risorse*$_{\text{G}}$. Questo include l'aggiornamento dei sistemi, la sostituzione di componenti obsoleti o malfunzionanti e l'ottimizzazione delle configurazioni per garantire efficienza e scalabilità;
- **Prevenzione dei problemi e sicurezza**: oltre alla risoluzione dei guasti, la manutenzione prevede un approccio proattivo, basato su un monitoraggio costante dei log di sistema, sull'analisi delle prestazioni e sull'implementazione di misure di sicurezza per prevenire vulnerabilità e interruzioni del servizio;
- **Gestione delle segnalazioni e interventi correttivi**: in caso di malfunzionamenti, viene utilizzato un *sistema*$_{\text{G}}$ di ticketing per raccogliere, analizzare e risolvere tempestivamente le segnalazioni.

5.2.4 Strumenti

Gli strumenti utilizzati per il processo di gestione dell'infrastruttura sono:

- Jira.

5.3 Miglioramento

5.3.1 Scopo

Il processo di miglioramento mira a ottimizzare continuamente il *way of working*$_{\text{G}}$, aumentando l'efficacia e l'efficienza delle attività senza compromettere la qualità.

5.3.2 Determinazione del processo

Il team definisce i processi del ciclo di vita del *software*$_{\text{G}}$ basandosi sullo Standard ISO/IEC 12207:1997. Questi processi vengono documentati e, quando possibile, viene implementato un *sistema*$_{\text{G}}$ di monitoraggio per garantire il controllo e il miglioramento continuo delle loro applicazioni.

5.3.3 Valutazione del processo

L'attività di valutazione di un processo consiste nell'analizzare e misurare sistematicamente come un processo viene eseguito per valutarne l'efficacia, l'efficienza e la conformità al *way of working*$_{\text{G}}$. L'obiettivo principale della valutazione è identificare i punti di forza e le aree di miglioramento del processo, fornendo così una base per ottimizzarlo continuamente.
La valutazione di un processo si concentra su aspetti chiave come:

- Se il processo raggiunge gli obiettivi prefissati;
- Se il processo utilizza le *risorse*$_{\text{G}}$ in modo ottimale;

- Se il processo è conforme al *way of working*$_{\text{G}}$ stabilito;
- Il livello di maturità del processo;
- Le opportunità di miglioramento per ottimizzare il processo.

5.3.4 Miglioramento del processo

Il miglioramento di un processo deve seguire un approccio strutturato e ciclico (ciclo di Deming), che prevede una serie di fasi interconnesse:

- **Pianificazione del miglioramento (Plan)**: sulla base della valutazione, si definiscono obiettivi specifici e azioni correttive per il miglioramento del processo, stabilendo scadenze, responsabilità e *risorse*$_{\text{G}}$ necessarie, senza includere la pianificazione del progetto;
- **Esecuzione delle azioni di miglioramento (Do)**: le azioni pianificate vengono implementate attraverso modifiche alle procedure, l'introduzione di nuove tecnologie o la formazione del personale, con una *documentazione*$_{\text{G}}$ accurata delle modifiche e delle azioni intraprese;
- **Valutazione dell'efficacia del miglioramento (Check)**: dopo l'implementazione, si valuta l'impatto delle modifiche confrontando i risultati con gli obiettivi prefissati, per verificare se i miglioramenti desiderati sono stati raggiunti;
- **Azione e consolidamento (Act)**: se i miglioramenti sono efficaci vengono consolidati, altrimenti si analizzano le cause e si pianificano nuove azioni correttive, riprendendo il ciclo di Deming.

5.4 Formazione

5.4.1 Scopo

Lo scopo del processo di formazione è aiutare i membri del gruppo a sviluppare le competenze necessarie per lavorare in modo efficace e produttivo. Questo processo si propone di garantire che le persone abbiano le conoscenze giuste per raggiungere gli obiettivi del progetto e rispettare gli standard di qualità.

5.4.2 Pianificazione

Ogni membro del team ha la libertà di scegliere come formarsi, seguendo un approccio pratico e personalizzato, in base alle proprie esigenze e interessi. Inoltre, chi ha più esperienza e competenze in un determinato campo è incoraggiato ad aiutare e supportare i colleghi con meno esperienza, creando un ambiente di apprendimento collaborativo. Questo permette a tutti di crescere insieme, condividendo conoscenze e *risorse*$_{\text{G}}$ in modo continuo e dinamico.

5.4.3 Raccolta del materiale

Il materiale per la formazione viene principalmente cercato e trovato *online*$_{\text{G}}$, sfruttando le *risorse*$_{\text{G}}$ disponibili su internet. Ogni membro del team può accedere a tutorial, corsi, articoli, video e altre *risorse*$_{\text{G}}$ gratuite per approfondire le proprie competenze in modo autonomo

6 Metriche di qualità

6.1 Metriche di qualità del processo

6.1.1 Fornitura

- **CV (Cost Variance):** Misura la deviazione dei costi rispetto al $budget_{\mathrm{G}}$, se il costo è negativo significa che si è sforato il limite del $budget_{\mathrm{G}}$($SPI_{\mathrm{G}} > 1$: in anticipo rispetto ai tempi pianificati, $SPI_{\mathrm{G}} < 1$: in ritardo rispetto ai tempi pianificati).
  CV = EV_{G} - AC.

- **PV (Planed Value):** Il valore pianificato, ovvero il costo stimato del lavoro previsto in un determinato momento del progetto.

- ***EV*$_{\mathbf{G}}$ (Earned Value):** Rappresenta il valore guadagnato, che rappresenta il costo stimato del lavoro effettivamente completato in quel momento.

- **AC (*Actual Cost*$_{\mathbf{G}}$):** Rappresenta il costo effettivo, cioè quanto è stato realmente speso fino a quel punto.

- **CPI (Cost Performance Index$_{\mathbf{G}}$):** Indica se il progetto sta spendendo meno o più del previsto (CPI > 1: sotto $budget_{\mathrm{G}}$, CPI < 1: sopra $budget_{\mathrm{G}}$).
  La formula è CPI = $\frac{EV_{\mathrm{G}}}{\mathrm{AC}}$.

- **SPI (Schedule Performance Index$_{\mathbf{G}}$):** Rappresenta l'efficienza temporale con cui il lavoro pianificato è stato completato rispetto a quanto programmato.
  La formula è $SPI_{\mathrm{G}} = \frac{EV_{\mathrm{G}}}{PV_{\mathrm{G}}}$.

- ***BAC*$_{\mathbf{G}}$ (Budget At Completion):** Rappresenta il $budget_{\mathrm{G}}$ totale pianificato per il completamento del progetto.

- **EAC (*Estimated At Completion*$_{\mathbf{G}}$):** Rappresenta l'aggiornamento della stima del valore per la realizzazione del progetto, ovvero il BAC_{G} ricalcolato in base allo stato attuale del progetto. La formula è $EAC_{\mathrm{G}} = \frac{BAC_{\mathrm{G}}}{\mathrm{CPI}}$.

- **VAC (Variance At Completion):** Rappresenta la differenza tra il $budget_{\mathrm{G}}$ previsto e quello attuale alla fine del progetto.
  La formula è VAC = BAC_{G} - EAC_{G}.

- **ETC (Estimated To Completion):** Rappresenta la valutazione del costo supplementare richiesto per portare a termine il progetto.
  La formula è ETC = EAC_{G} - AC.

- **SV (Schedule Variance):** Indica se le attività pianificate del progetto sono in linea, anticipate o in ritardo rispetto alla programmazione.
  La formula è SV = EV_{G} - PV_{G}.

- **BV (Budget Variance):** Indica se, alla data attuale, le spese sostenute sono superiori o inferiori rispetto a quanto originariamente previsto nel $budget_{\mathrm{G}}$.
  La formula è BV = PV_{G} - AC.

6.1.2 Sviluppo

- **SC (Statement Coverage):** Rappresenta la percentuale di istruzioni nel codice che vengono eseguite durante i test. La formula è SC = $\frac{\text{N Statement eseguiti}}{\text{N Statement totali}} * 100$.

6.1.3 Documentazione

- **IG (Indice Gulpease):** Rappresenta un indicatore per analizzare la facilità di lettura di un testo scritto in italiano. L'Indice Gulpease si basa su due variabili linguistiche principali: la lunghezza delle parole e quella delle frasi.
  La formula per determinarlo è: IG = $89 + \frac{300*\text{NF-NL}}{\text{NP}}$, dove:
  - **NF:** Indica il numero delle frasi.
  - **NL:** Indica il numero delle lettere.
  - **NP:** Indica il numero di parole.

  Questo indice fornisce un punteggio che varia da 0 a 100. I possibili punteggi possono essere:
  - **0-55:** Testo incomprensibile.
  - **56-70:** Testo molto difficile.
  - **71-80:** Testo difficile.
  - **81-95:** Testo facile.
  - **95-100:** Testo molto facile.

  Per calcolarlo viene utilizzato un *software*$_{\text{G}}$ *online*$_{\text{G}}$: https://farfalla-project.org/readability_static/

6.1.4 Gestione delle qualità

- **MNS (Metriche Non Soddisfatte):** Rappresenta le quantità di metriche che il progetto non riesce a soddisfare o mantenere.

6.2 Metriche per la qualità del prodotto

6.2.1 Funzionalità

- **ROS (Requisiti Obbligatori Soddisfatti):** Rappresenta la percentuale di requisiti obbligatori che sono stati soddisfatti durante la creazione del prodotto.
  La formula è ROS = $\frac{\text{requisiti obbligatori soddisfatti}}{\text{requisiti obbligatori totali}} * 100$.
- **RDS (Requisiti Desiderabili Soddisfatti):** Rappresenta la percentuale di requisiti desiderabili che sono stati soddisfatti durante la creazione del prodotto.
  La formula è RDS = $\frac{\text{requisiti desiderabili soddisfatti}}{\text{requisiti desiderabili totali}} * 100$.
- **RPS (Requisiti Opzionali Soddisfatti):** Rappresenta la percentuale di requisiti opzionali che sono stati soddisfatti durante la creazione del prodotto.
  La formula è RPS = $\frac{\text{requisiti opzionali soddisfatti}}{\text{requisiti opzionali totali}} * 100$.

6.2.2 Affidabilità

- **PTCP (Passed Test Cases Percentage):** Rappresenta la percentuale di casi di $test_{\mathrm{G}}$ completati con successo rispetto al numero totale di casi di $test_{\mathrm{G}}$ pianificati. La formula è PTCP $= \frac{\text{test superati}}{\text{test totali}} * 100$.

- **CC (Code Coverage$_{\mathbf{G}}$):** Rappresenta il numero di linee di codice Verificate con esito positivo all'interno di un processo di test.
  La formula è CC $= \frac{\text{linee di codice scritte}}{\text{linee di codice totali}} * 100$.

6.2.3 Manutenibilità

- **SFIN (Structure Fan IN):** Rappresenta la quantità di moduli o componenti che interagiscono direttamente o dipendono da un modulo o una funzione specifica. Un valore elevato suggerisce che molte parti del $sistema_{\mathrm{G}}$ fanno affidamento su quel particolare modulo.

- **SFOUT (Structure Fan Out):** Rappresenta la quantità di connessioni o relazioni che un componente o modulo ha con altri elementi del $sistema_{\mathrm{G}}$. Questa misura riflette il numero di moduli che interagiscono o su cui si basa un determinato modulo. Un fan-out elevato può segnalare che un modulo è fortemente dipendente da molti altri.

6.2.4 Efficienza

- **TDE (Tempo di Elaborazione):** Rappresenta il tempo di risposta dal momento in cui vengono inseriti dati all'interno del prodotto $software_{\mathrm{G}}$ al momento in cui vengono visualizzati dall'utente in questo caso l'installatore.

7 Strumenti

In questa sezione vengono presentati tutti gli strumenti utilizzati dal gruppo durante lo svolgimento del progetto.

7.1 Draw.io

https://www.diagrams.net (Ultimo accesso: 2025-04-01)

Diagrams.net è uno strumento gratuito basato su web per la creazione di diagrammi e grafici, utilizzato per creare diagrammi di flusso, mappe concettuali, organigrammi, wireframe e molti altri tipi di rappresentazioni visive. Permette di collaborare in tempo reale e offre una vasta libreria di forme predefinite.

7.2 Discord

https://discord.com (Ultimo accesso: 2025-04-01)

Discord è una piattaforma di comunicazione vocale, video e testuale, molto utilizzata da community, team di gioco e gruppi di lavoro. Consente la creazione di server, canali tematici, chat di testo e videoconferenze. Inoltre, è possibile integrare bot e automatizzare operazioni.

7.3 Docker

https://www.docker.com (Ultimo accesso: 2025-04-01)

Docker$_{\text{G}}$ è una piattaforma che utilizza la virtualizzazione a livello di *sistema*$_{\text{G}}$ operativo per distribuire applicazioni in contenitori isolati (*container*$_{\text{G}}$). *Docker*$_{\text{G}}$ consente di costruire, testare e distribuire applicazioni in modo coerente e ripetibile su vari ambienti.

7.4 GitHub

https://github.com (Ultimo accesso: 2025-04-01)

GitHub$_{\text{G}}$ è un servizio di hosting basato su *Git*$_{\text{G}}$ per il controllo versione e la gestione di codice sorgente. Permette la collaborazione tra sviluppatori, gestione di *pull request*$_{\text{G}}$, gestione *issue*$_{\text{G}}$ e *documentazione*$_{\text{G}}$ dei progetti. È ampiamente usato per progetti open-source.

7.5 Google Calendar

https://calendar.google.com (Ultimo accesso: 2025-04-01)

Google Calendar è un servizio gratuito di Google per la gestione degli appuntamenti e la pianificazione degli eventi. Permette la creazione di calendari condivisi, l'invio di inviti, la sincronizzazione con dispositivi mobili e l'integrazione con altre applicazioni Google.

7.6 Google Gmail

https://mail.google.com (Ultimo accesso: 2025-04-01)

Google Gmail è un servizio di posta elettronica gratuito fornito da Google, noto per la sua interfaccia pulita e funzionalità avanzate come la ricerca potente, l'archiviazione e la gestione dei filtri e delle etichette. Integrato con altri strumenti Google.

7.7 Google Sheets

https://sheets.google.com (Ultimo accesso: 2025-04-01)

Google Sheets$_{\text{G}}$ è un'applicazione di fogli di calcolo *online*$_{\text{G}}$, parte di Google Drive, che consente di creare, modificare e collaborare su fogli di calcolo in tempo reale. È una valida alternativa a Microsoft Excel, con funzionalità di collaborazione e integrazione con altri strumenti Google.

7.8 Jira

https://www.atlassian.com/software/jira (Ultimo accesso: 2025-04-01)

Jira$_{\text{G}}$ è un *software*$_{\text{G}}$ di gestione progetti e tracciamento dei problemi, ampiamente utilizzato in ambienti agili per gestire *sprint*$_{\text{G}}$, *backlog*$_{\text{G}}$ e flussi di lavoro. Offre funzionalità avanzate per il monitoraggio delle attività, reportistica e integrazione con altri strumenti di sviluppo.

7.9 LaTeX

https://www.latex-project.org (Ultimo accesso: 2025-04-01)

LaTeX$_{\text{G}}$ è un linguaggio di markup per la scrittura e la gestione di documenti complessi, particolarmente usato in ambito accademico e scientifico. *LaTeX*$_{\text{G}}$ è perfetto per la produzione di documenti tecnici, con formule matematiche, bibliografie e strutture complesse.

7.10 Microsoft PowerPoint

https://www.microsoft.com/en-us/microsoft-365/powerpoint (Ultimo accesso: 2025-04-01)

Microsoft PowerPoint è un *software*$_{\text{G}}$ di presentazione parte della suite Microsoft Office. È utilizzato per creare diapositive visive con testo, immagini, grafici e video per presentazioni professionali. Ha strumenti avanzati per animazioni e transizioni.

7.11 Microsoft Teams

https://www.microsoft.com/en-us/microsoft-teams (Ultimo accesso: 2025-04-01)

Microsoft Teams$_{G}$ è una piattaforma di collaborazione che integra chat, videochiamate, archiviazione di file e integrazione con altre applicazioni Microsoft. Viene utilizzata per facilitare la comunicazione e la collaborazione a distanza in team di lavoro.

7.12 Slack

https://slack.com (Ultimo accesso: 2025-04-01)

Slack$_{G}$ è uno strumento di comunicazione per team che unisce chat di testo, canali di discussione, integrazione con altre applicazioni e supporto per bot. È progettato per facilitare la collaborazione, la condivisione di informazioni e il lavoro a distanza.

7.13 StarUML

http://staruml.io (Ultimo accesso: 2025-04-01)

StarUML è uno strumento di modellazione *UML*$_{G}$ (Unified Modeling Language) che consente la progettazione di diagrammi per rappresentare visivamente sistemi *software*$_{G}$ e architetture. Supporta diversi tipi di diagrammi, inclusi quelli di classi, sequenze e attività.

7.14 Visual Studio Code

https://code.visualstudio.com (Ultimo accesso: 2025-04-01)

Visual Studio Code è un editor di codice sorgente gratuito sviluppato da Microsoft. Supporta numerosi linguaggi di programmazione e offre un'ampia gamma di estensioni per il debugging, il controllo versione e la gestione di progetti di sviluppo.

7.15 Whatsapp

https://www.whatsapp.com (Ultimo accesso: 2025-04-01)

Whatsapp è un'applicazione di messaggistica istantanea che consente di inviare messaggi di testo, foto, video, file e fare chiamate vocali e video. È ampiamente utilizzata per comunicazioni personali e professionali, supportando anche chat di gruppo.