Università degli Studi di Padova

GlitchHub TEAM

Norme di Progetto

Versione **1.7.1**

Stato	Verificato
Distribuzione	GlitchHub Team Prof. Vardanega Tullio Prof. Cardin Riccardo

Registro Modifiche

Ver.	Data	Autore	Verificatore	Descrizione
1.7.1	31/03/2026	Riccardo Graziani	Siria Salvalaio	Applicati suggerimenti forniti durante la verifica della v1.7.0
1.7.0	31/03/2026	Riccardo Graziani	Siria Salvalaio	Descritte le norme di codifica per il codice Typescript nella Sezione 2.2.4.2
1.6.1	29/03/2026	Alessandro Dinato	Elia Ernesto Stellin	Applicazione correzioni segnalate durante la verifica di v1.6.0
1.6.0	25/03/2026	Alessandro Dinato	Elia Ernesto Stellin	Aggiornamento delle norme di codifica
1.5.0	25/03/2026	Alessandro Dinato	Elia Ernesto Stellin	Aggiunte norme per l'utilizzo di **Apidog**$_G$ (Sezione 4.2.3.1)
1.4.1	09/03/2026	Riccardo Graziani	Elia Ernesto Stellin	Applicate correzioni a Sezione 4.1.2 rilevate durante la verifica
1.4.0	09/03/2026	Riccardo Graziani	Elia Ernesto Stellin	Modificata Sezione 4.1.2 includendo workflow relativo alla chiusura automatica degli *issue branch* e il collegamento automatico delle *issue* alle relative *pull request* e *parent issue*. Definizione della label *epic* per *parent issue*
1.3.0	26/02/2026	Riccardo Graziani	Elia Ernesto Stellin	Modificata Sezione 4.1.2 per riflettere il nuovo workflow basato su *issue-branch* e *pull request*; Aggiunta Sezione 4.2.3.5.6 per descrivere convenzioni di nomenclatura dei branch automatici
1.2.0	18/02/2026	Elia Ernesto Stellin	Alessandro Dinato	Modificato utilizzo delle maiuscole; Aggiornata Sezione 4.2.3.5.2 in modo tale da rispecchiare struttura della repo; Aggiornata Sezione 3.5.2 per includere informazioni su tracciamento di requisiti e test

Ver.	Data	Autore	Verificatore	Descrizione
1.1.0	17/02/2026	Michele Dioli	Alessandro Dinato	Aggiunta sezioni metriche
1.0.1	17/02/2026	Elia Ernesto Stellin	Alessandro Dinato	Corretta formula per metrica MPC-WD (Sezione 5.1.5.3)
1.0.0	13/02/2026	Elia Ernesto Stellin	Riccardo Graziani	Versione stabile del documento
0.5.1	13/02/2026	Elia Ernesto Stellin	Riccardo Graziani	Sistemati link rotti e riferimenti al glossario
0.5.0	11/02/2026	Elia Ernesto Stellin	Riccardo Graziani	Espansa Sezione 3.1; Create Sezione 3.2.4, Sezione 3.3, Sezione 3.4 e Sezione 3.5
0.4.1	10/02/2026	Riccardo Graziani	Elia Ernesto Stellin	Applicate correzioni a Sezione 2.2.2 rilevate durante la verifica
0.4.0	09/02/2026	Riccardo Graziani	Jaume Bernardi	Aggiunte Sezione 2, Sezione 2.1, Sezione 2.2
0.3.1	07/02/2026	Elia Ernesto Stellin	Alessandro Dinato	Applicate correzioni a Sezione 4 rilevate durante verifica.
0.3.0	07/02/2026	Elia Ernesto Stellin	Alessandro Dinato	Aggiunte Sezione 4, Sezione 4, Sezione 4.2, Sezione 4.3, Sezione 4.4
0.2.1	28/11/2025	Siria Salvalaio	Hossam Ezzemouri	Aggiunta paragrafo Sezione 3.1.3.1.
0.2.0	25/11/2025	Elia Ernesto Stellin	Jaume Bernardi	Ristrutturazione di qualche frase e correzione di link mancanti.
0.1.0	24/11/2025	Elia Ernesto Stellin	Jaume Bernardi	Stesura iniziale di Sezione 1, Sezione 3, introduzione di Sezione 3.1; Stesura di Sezione 3.1.1, Sezione 3.1.2, Sezione 3.1.4, Sezione 3.1.5, Sezione 3.1.6.
0.0.1	15/11/2025	Elia Ernesto Stellin	Jaume Bernardi	Bozza prima stesura

Indice

1. Introduzione

1.1. Scopo del documento

Lo scopo di questo documento è di descrivere il ***Way of Working*** del gruppo *GlitchHub Team* durante lo svolgimento del **progetto didattico**.

A tale scopo, il gruppo ha deciso di prendere come riferimento lo standard internazionale **ISO/IEC 12207:1995**, che definisce una struttura normata per descrivere i processi di cicli di vita del Software.

Più nello specifico, vengono riconosciuti dallo standard tre tipi principali di processi:

- **Processi primari** (Sezione 2): i processi essenziali e imprescindibili per lo svolgimento del progetto
- **Processi di supporto** (Sezione 3): i processi che si integrano con i processi primari per semplificare lo svolgimento del progetto
- **Processi organizzativi** (Sezione 4): i processi che semplificano le procedure di sviluppo

Questo documento ha carattere **incrementale**, per cui, verrà modificato e aggiornato col progredire dello sviluppo del **progetto didattico** a seguito di miglioramenti e modifiche al ***Way of Working*** del gruppo.

1.2. Scopo del prodotto

Il progetto proposto dal **capitolato C7** di **M31 Srl** è inerente alla gestione e acquisizione di dati provenienti da sensori **IoT** distribuiti ed eterogenei a basso consumo energetico, i quali permettono di raccogliere dettagliatamente dati da qualunque fonte.

Nello specifico, lo sviluppo richiesto dal progetto si colloca in un sistema distribuito di acquisizione e smistamento dati da sensori **Bluetooth Low Energy (BLE)**. L'obiettivo del progetto è di sviluppare un'infrastruttura scalabile che comunichi in modo sicuro con sensori non conformati e che ne gestisca i dati raccolti tramite strumenti di monitoraggio e visualizzazione, ad amministratori e utenti finali, garantendo al contempo la totale separazione dei dati tra diversi *tenant*.

Secondo quanto specificato nella **Lettera di Presentazione**, il gruppo si è posto di realizzare questo progetto entro il **27 marzo 2026** con un budget massimo di **Euro 12.975**.

A seguito di una successiva suddivisione delle ore di lavoro previste, il gruppo ha deciso di ricalcolare la distribuzione delle ore di lavoro, arrivando a una stima di **Euro 12.825** per il costo complessivo del progetto. Per ulteriori informazioni relativamente a ciò, è possibile consultare la sezione 4.1 del Piano di Progetto.

La data finale di consegna del progetto è confermata al **27 marzo 2026**.

1.3. Glossario

La creazione e lo sviluppo di un sistema software richiedono una grande operazione di progettazione e analisi del dominio del software, che avviene a priori della scrittura di codice. Il gruppo, perciò, si impegna a raccogliere tali informazioni in una maniera facilmente accessibile in modo tale da favorire una maggiore asincronia ed efficienza nelle attività di progetto.

Il principale tipo di ambiguità che si può creare nello svolgimento del progetto è l'incomprensione del significato dei termini utilizzati dal gruppo. A tale scopo, la nomenclatura adottata da quest'ultimo verrà raccolta nel **glossario**, un **documento incrementale** che definisce ogni parola rilevante per il dominio del progetto.

Come descritto nel **verbale interno del 19 novembre 2025**, il gruppo si impegna ad annotare tutte le parole del glossario che compaiono nei documenti con una G a pedice in questo modo:

parola$_G$

Per una buona comprensione del dominio da parte del gruppo, è fondamentale che ogni membro visioni periodicamente il glossario per rimanere allineato sulle definizioni di dominio.

1.4. Riferimenti

1.4.1. Riferimenti normativi

- **Capitolato d'appalto$_G$** C7
- https://www.math.unipd.it/~tullio/IS-1/2025/Progetto/C7.pdf
  - **Ultimo accesso**: 17 febbraio 2025

1.4.2. Riferimenti informativi

- **Standard ISO/IEC 12207:1995**
  - https://www.math.unipd.it/~tullio/IS-1/2009/Approfondimenti/ISO_12207-1995.pdf
  - **Ultimo accesso**: 17 febbraio 2026
  - **Note**: Questo documento ha una struttura che si ispira a questo standard, ma non ha la pretesa di rispettarlo pienamente.
- **Glossario v0.5.0**
  - https://glitchhub-team.github.io/pdf/RTB/DocumentiInterni/glossary.pdf
  - **Ultimo accesso**: 17 febbraio 2026
- **Verifica e validazione: introduzione** – Lezione T9 del prof. Vardanega
  - https://www.math.unipd.it/~tullio/IS-1/2025/Dispense/T09.pdf
  - **Ultimo accesso**: 17 febbraio 2026
- **Verifica e validazione: analisi dinamica** – Lezione T11 del prof. Vardanega
  - https://www.math.unipd.it/~tullio/IS-1/2025/Dispense/T11.pdf
  - **Ultimo accesso**: 17 febbraio 2026

2. Processi primari

I processi primari della norma **ISO/IEC 12207:1995** definiscono le attività fondamentali del ciclo di vita del software attraverso cinque processi: acquisizione, fornitura, sviluppo, operazione e manutenzione, assicurando la conformità ai requisiti e agli obiettivi di qualità stabiliti.

Tra i **processi$_G$** primari applicati nel progetto si distinguono:

- **Fornitura**
- **Sviluppo**

2.1. Fornitura

Il processo di **fornitura** definisce le attività attraverso cui un fornitore pianifica, realizza e consegna un prodotto software. Il processo comprende la risposta alla richiesta del cliente, la definizione e l'accordo sui termini di fornitura, la pianificazione del progetto, l'esecuzione, il monitoraggio dell'avanzamento e la consegna del prodotti.

L'obiettivo del processo è garantire che il software sia sviluppato e fornito in modo controllato, tracciabile e conforme agli standard di qualità concordati con il proponente.

2.1.1. Strumenti a supporto

- **GitHub**$_G$: come infrastruttura per il controllo di versione
  - **GitHub Issues**: per l'assegnazione degli elementi del *backlog* e la segnalazione di eventuali problemi nella **Repository**$_G$
  - **Github Project**: per la visualizzazione delle task in modalità Kanban, utile ad illustrare lo stato d'avanzamento delle task assegnate;
- **Whatsapp**: come canale di comunicazione all'interno del gruppo;
- **Discord**$_G$: per svolgere le riunioni interne del gruppo;
- **ClickUp**$_G$: come strumento di project management, utilizzato principalmente come piattaforma di organizzazione e **condivisione** di documenti.

Le comunicazioni e gli incontri con l'azienda proponente necessitano invece dei seguenti strumenti:

- **Gmail**: come piattaforma di comunicazione asincrona con la proponente;
- **Microsoft Teams**: per svolgere le riunioni da remoto con l'azienda proponente.

2.1.2. Attività previste

Il processo di fornitura si compone delle seguenti attività di seguito descritte:

- **Inizializzazione**: il fornitore analizza i requisiti contenuti nella richiesta dell'acquirente per valutare la fattibilità tecnica ed economica. In questa fase si decide se partecipare alla fornitura, si definiscono le risorse necessarie e si individuano eventuali requisiti da contrattare con il proponente;
- **Preparazione della risposta**: il fornitore elabora la proposta formale che tenga conto di quanto emerso durante l'attività di inizializzazione;
- **Contrattazione**: riguarda la negoziazione con il proponente, in cui il fornitore presenta la proposta elaborata in precedenza, con l'obiettivo di giungere alla sottoscrizione di un accordo formale;
- **Pianificazione**: il fornitore stabilisce la struttura di gestione e qualità, selezionando, se non specificato nel contratto, il modello di ciclo di vita software adeguato. Sono definite le risorse e le tecnologie necessarie allo sviluppo, a fronte di un'analisi dei rischi associati a ciascuna di esse;
- **Esecuzione e controllo**: il fornitore, dopo aver documentato l'attività di pianificazione, realizza quanto stabilito, monitorando la qualità del prodotto software e lo stato di avanzamento dello sviluppo;
- **Revisione e valutazione**: il fornitore assume la responsabilità di coordinare le attività di comunicazione con il proponente, supportando attivamente riunioni informali e revisioni congiunte. Il fornitore esegue la verifica e la validazione del processo per dimostrare conformità del prodotto ai requisiti;
- **Consegna e completamento**: il fornitore consegna il prodotto finale, garantendo assistenza al proponente a supporto del prodotto consegnato.

2.1.3. Documentazione fornita

Di seguito presentiamo l'elenco completo della documentazione che **GlitchHub Team** consegnerà all'azienda proponente **M31** e ai committenti **Prof. Tullio Vardanega** e **Prof. Riccardo Cardin**.

2.1.3.1. Lettera di candidatura

La **Lettera di Candidatura** è il documento con cui **GlitchHub Team** ha presentato formalmente la propria candidatura al **capitolato d'appalto**$_G$ proposto dall'azienda **M31**.

2.1.3.2. Valutazione dei capitolati

La **Valutazione dei capitolati** è il documento in cui **GlitchHub Team** ha fornito, per ogni **capitolato$_G$**, un'analisi dei rispettivi punti di forza e debolezza, e le motivazioni che hanno spinto il gruppo a scegliere/non scegliere tale **capitolato$_G$**.

2.1.3.3. Dichiarazione degli impegni

La **Dichiarazione degli impegni** è il documento in cui **GlitchHub Team** ha formalizzato la pianificazione economica e organizzativa del progetto. Il documento espone le stime relative all'impegno **orario** suddiviso per singoli componenti e per ruoli, definisce il costo **complessivo** dell'opera e illustra i criteri adottati per la **rotazione** dei ruoli all'interno del gruppo

2.1.3.4. Lettera di presentazione

La **Lettera di Presentazione** è il documento tramite il quale **GlitchHub Team** intende formalizzare la propria candidatura alle revisioni di avanzamento legate alle **baseline$_G$** del progetto didattico, ossia la **Requirements and Technology Baseline (RTB$_G$)** e la **Product Baseline (PB$_G$)**.

2.1.3.5. Analisi dei Requisiti

L'**Analisi dei Requisiti** è il documento in cui **GlitchHub Team** definisce in dettaglio tutti i **requisiti$_G$** del progetto, classificandoli in obbligatori, desiderabili e opzionali. Partendo da un'introduzione sul contesto operativo, l'analisi descrive i **casi d'uso$_G$** individuati e i relativi **requisiti$_G$** specifici, per poi mappare la corrispondenza tra questi elementi attraverso un'apposita tabella di tracciamento

2.1.3.6. Specifica Tecnica

La **Specifica Tecnica** è il documento in cui **GlitchHub Team** descrive in dettaglio l'architettura del sistema, i componenti software che lo compongono e le interfacce tra di essi. Partendo da un'introduzione sul contesto operativo, la specifica tecnica descrive l'architettura di alto livello del sistema, per poi scendere in dettaglio sui singoli componenti software.

2.1.3.7. Piano di Progetto

Il **Piano di Progetto** è il documento in cui **GlitchHub Team** documenta l'evoluzione di ogni **sprint$_G$** confrontando la fase previsionale con quella consuntiva. Per ciascuno **sprint$_G$**, infatti, vengono inizialmente definite le **attività** previste, i **rischi** potenziali e il **preventivo** delle risorse; successivamente, il documento registra le attività effettivamente svolte, il consumo reale delle risorse (con relativo aggiornamento di quelle residue) e gli esiti della **retrospettiva** del gruppo.

2.1.3.8. Piano di Qualifica

Il **Piano di Qualifica** illustra le strategie e le attività operative volte a garantire la **qualità** del prodotto finale da parte del gruppo **GlitchHub Team**. Nello specifico, il documento stabilisce le **metriche** di valutazione applicate sia ai **processi** che al **prodotto** stesso, riportando inoltre i dettagli sui **test** di verifica effettuati

2.1.3.9. Norme di Progetto

Le **Norme di Progetto** definiscono il **Way of Working$_G$** del gruppo **GlitchHub Team**, stabilendo gli **strumenti** da utilizzare, le convenzioni per la **stesura** del codice e della documentazione, e le procedure operative necessarie a garantire uniformità ed efficienza nello sviluppo.

2.1.3.10. Glossario

Il **glossario** raccoglie e definisce la **terminologia** tecnica adottata nel corso del progetto. Il suo obiettivo è costituire un punto di **riferimento** univoco per tutti i membri del gruppo, garantendo una comunicazione chiara e priva di ambiguità.

2.1.3.11. Verbali interni
Rappresentano i **verbali** delle riunioni svolte con la sola presenza dei membri del gruppo.

2.1.3.12. Verbali esterni
Rappresentano i **verbali** delle riunioni svolte con la presenza di persone esterne al gruppo.

2.2. Sviluppo
Il processo di **sviluppo** definisce le attività per la realizzazione del software, comprendendo l'analisi dei requisiti, la progettazione, l'implementazione, l'integrazione e la verifica, al fine di garantire la conformità del prodotto alle specifiche e ai requisiti di qualità stabiliti.

2.2.1. Strumenti a supporto
- **Angular**$_G$: come framework di sviluppo frontend;
- **Gin**$_G$: come framework di sviluppo backend;
- **Go**$_G$: come linguaggio di programmazione per lo sviluppo dei servizi di publish e subscribe;
- **Visual Studio Code**: per la codifica del software;
- **StarUML**: come strumento per la redazione dei **diagrammi dei casi d'uso**$_G$.
- **Gofumpt**: come strumento per la formattazione del codice Go;
- **Golangci-Lint**: come strumento per l'analisi del codice Go;
- **ESLint**: come strumento per l'analisi del codice Typescript;
- **Prettier**: come strumento per la formattazione del codice Typescript.

2.2.2. Attività previste
- **Implementazione del processo**: in cui viene stabilita l'articolazione delle fasi di ingegnerizzazione, garantendo che ogni attività di sviluppo sia coerente con la strategia di realizzazione scelta;
- **Analisi dei requisiti di sistema**: in cui si individuano le necessità dell'utente finale e si definiscono le funzionalità che il Sistema deve soddisfare;
- **Progettazione dell'architettura di sistema**: in cui si stabilisce l'architettura di alto livello del Sistema identificando gli elementi hardware e software atti a soddisfare i requisiti individuati;
- **Analisi dei requisiti software**: in cui si stabiliscono e documentano i requisiti software (funzionali, prestazionali, di interfaccia, di sicurezza e di qualità) per ogni elemento software identificato nel Sistema;
- **Progettazione dell'architettura software**: ossia trasformare i requisiti in un'architettura che descriva la struttura del software, identificando i componenti software e le interfacce esterne ed interne;
- **Progettazione dettagliata del software**: ossia la progettazione in dettaglio dei componenti software individuati, fino alla definizione delle singole unità software;
- **Codifica e test del software**: ossia lo sviluppo delle unità software individuate per ogni singolo componente, seguito da test effettuati sulle singole unità per verificarne il corretto funzionamento in isolamento;
- **Integrazione del software**: ossia l'integrazione delle unità software e dei componenti software, testando gli aggregati man mano che vengono sviluppati per verificarne il corretto funzionamento;
- **Test di qualificazione del software**: ossia lo svolgimento dei test di qualificazione per dimostrare che il prodotto software soddisfi i requisiti definiti.
- **Integrazione di sistema**: ossia integrare tutte le componenti sviluppate nel Sistema;

- **Test di qualificazione del sistema**: ossia eseguire test di qualificazione sul Sistema completo per valutare la conformità ai requisiti e assicurare che il Sistema sia pronto per la consegna;
- **Installazione del software**: ossia la fornitura del prodotto software nell'ambiente operativo secondo quanto specificato nel contratto;
- **Supporto all'accettazione del software**: in cui si fornisce supporto alle revisioni e ai test di accettazione dell'acquirente per completare la consegna formale del prodotto software.

In relazione alla definizione delle **baseline$_G$** previste per il progetto, ossia la Requirements and Technology Baseline (**RTB$_G$**) e la Product Baseline (**PB$_G$**), **GlitchHub Team** ha identificato l'**analisi dei requisiti** (§ 2.2.3) e la **codifica** (§ 2.2.4) come attività principali dell'**RTB$_G$**. Di contro, sono state identificate la **progettazione dell'architettura software**, la **progettazione dettagliata del software** e la **codifica** come attività principali della **PB$_G$**. Di conseguenza sono state approfondite le attività principali relative all'**RTB$_G$**, mentre verranno stese (o ampliate) in seguito le sezioni relative alle attività principali della **PB$_G$**.

2.2.3. Analisi dei Requisiti

L'attività di **analisi dei requisiti** ha lo scopo di comprendere, individuare e definire in modo completo e strutturato tutti i **requisiti$_G$** che il **sistema** dovrà soddisfare. Essa consente di formalizzare le esigenze del proponente e di tradurle in **requisiti$_G$** chiaramente identificabili e verificabili.
Il risultato di tale attività è documentato nel documento di **Analisi dei Requisiti$_G$**, che raccoglie l'insieme dei **casi d'uso$_G$** individuati e i **requisiti$_G$** ad essi associati, costituendo il riferimento principale per le successive fasi di progettazione e sviluppo del sistema.

2.2.3.1. Casi d'uso

I **casi d'uso$_G$** sono identificati secondo la seguente nomenclatura:

UC[Primario].[Secondario]

in cui:

- **UC** indica **Use Case**, ossia la traduzione inglese di «**caso d'uso$_G$**»;
- **[Primario]** è un numero progressivo che identifica univocamente il **caso d'uso$_G$** all'interno dell'**Analisi dei Requisiti$_G$**
- **[Secondario]** è un numero progressivo che identifica un **caso d'uso$_G$** correlato in modo esclusivo al caso **primario**.

Ogni **caso d'uso$_G$** è inoltre corredato da un titolo che ne riassume lo scopo e da una descrizione testuale; per ulteriori dettagli si rimanda alla sezione introduttiva sui **casi d'uso$_G$** del documento di **Analisi dei Requisiti$_G$**.

2.2.3.2. Requisiti

I **requisiti$_G$** sono identificati secondo la seguente nomenclatura:

R[Tipologia]-[Codice]-[Priorità]

in cui:

- **R** indica l'abbreviazione di **requisito$_G$**
- **[Tipologia]** indica il tipo di requisito tra i seguenti valori:
  - **F** per **Funzionale**;
  - **NF** per **Non Funzionale**;
  - **D** per **Dominio**;
  - **V** per **Vincolo**;

- **[Codice]** identifica i **requisiti**$_G$ per tipologia, ed è composto da un numero progressivo univoco nell'ambito della tipologia;
- **[Priorità]** indica la priorità di ogni **requisito**$_G$ tra i seguenti valori:
  - **Obb** per **Obbligatorio**;
  - **Des** per **Desiderabile**;
  - **Opt** per **Opzionale**.

Per una descrizione più approfondita della tipologia e della priorità di un **requisito**$_G$, si rimanda alla sezione introduttiva sui **requisiti**$_G$ del documento di **analisi dei requisiti**$_G$.

2.2.4. Codifica

L'attività di **codifica**, svolta dai programmatori, consiste nel tradurre la progettazione definita dai **progettisti** nel prodotto software finale. In questa sezione vengono illustrate le **norme** e le **convenzioni** che i programmatori sono tenuti a rispettare, con l'obiettivo di:

- **Ottimizzare** la revisione: velocizzare le attività di verifica e collaudo del codice attraverso una scrittura più chiara.
- **Assicurare** l'evoluzione del sistema: migliorare la manutenibilità e la futura estensibilità del software prodotto.
- **Garantire l'uniformità** qualitativa: certificare che il codice sia conforme agli standard di qualità precedentemente fissati.

2.2.4.1. Dev Container

Per ogni **repository** del progetto sono stati configurati dei **Dev Container**$_G$ per garantire un ambiente di sviluppo uguale per ogni membro del gruppo e con tutti gli strumenti a supporto già definiti a monte. I diversi **Dev Container**$_G$ hanno bisogno di una configurazione minima o in alcuni casi nulla, in ogni caso si può trovare i passi per il setup nel file **README.md** presente in ogni repository.

L'avvio di un **Dev Container**$_G$ avviene tramite l'estensione **Dev Containers** di Visual Studio Code, infatti una volta installata sarà necessario premere **Cmd+Shift+P** (su MacOS) o **Ctrl+Shift+P** (su Windows e Linux) per aprire la Command Palette e scegliere l'opzione **Reopen in Container**.

Una volta aperto il **Dev Container**$_G$ sarà possibile accedere a tutti gli strumenti a supporto già configurati, e iniziare a lavorare sul codice del progetto.

2.2.4.2. Stile di codifica Typescript

In questa sezione vengono definiti gli standard implementativi per lo sviluppo in linguaggio **Typescript** all'interno del framework **Angular**$_G$, al fine di garantire l'uniformità del codice prodotto dal team.

2.2.4.2.1. Best practices

- **Signals**: i servizi devono esporre lo stato dell'applicazione tramite **signals**, rispettando l'approccio di programmazione reattiva **signal-first** introdotta da **Angular**$_G$ nella versione 17+;
- **Livelli di accesso**: il progetto segue una rigorosa politica di accesso, riassunta nelle seguenti regole:
  - applicare sempre `readonly` alle dipendenze iniettate con `inject()` e ai signal;
  - usare `private` come livello di accesso predefinito; elevare a `protected` solo ciò che il template richiede;
  - non usare `public` per campi o metodi dei componenti, ad eccezione di input, output e hook del ciclo di vita (`NgOnInit`, `NgOnDestroy`, etc...);

  - nei servizi, esporre lo stato esclusivamente tramite `.asReadonly()` o `computed()`, mai il signal scrivibile;
  - nei servizi, i metodi che costituiscono l'API pubblica sono marcati `public` mentre i metodi ausiliari interni sono `private`;
  - nei componenti, i metodi richiamati dal template sono `protected` mentre quelli usati solo nella classe sono `private`;
  - i signal interni dei servizi usano il prefisso `_`, mentre l'alias pubblico corrispondente è `public readonly`;
- **SRP (Single Responsibility Principle)**: ogni classe, interfaccia o modulo deve avere una singola responsabilità, evitando di creare elementi troppo complessi o con responsabilità multiple;
- **HTTP pattern**: l'interazione con il backend avviene attraverso una struttura a tre livelli, in cui i **componenti** comunicano con i servizi, i **servizi** comunicano con gli adapters e gli **adapters** comunicano con il backend;
- **Smart/Dumb components**: i componenti devono essere suddivisi in **smart** e **dumb**, in cui i primi gestiscono la logica di business e le comunicazioni con i servizi, mentre i secondi si occupano esclusivamente della presentazione dei dati ricevuti dai componenti smart.
- **Self-contained dialogs**: le finestre di dialogo sono trattate come componenti **smart** che comunicano con i servizi e gestiscono il proprio stato di errore.
- **Commenti**: commentare il codice dove non può essere autoesplicativo, evitando di riempirlo di commenti superflui;
- **Lingua**: scrivere il codice in inglese per evitare nomi ambigui.

2.2.4.2.2. Dependency Injection

Il pattern di **Dependency Injection**$_G$ è ampiamente utilizzato all'interno del framework **Angular**$_G$, e rappresenta un principio fondamentale per la progettazione e lo sviluppo di applicazioni modulari, scalabili e manutenibili.

Nel progetto, viene ampiamente utilizzato attraverso la funzione `inject()` di **Angular**$_G$, che consente di accedere ai servizi e alle dipendenze in modo semplice e diretto all'interno dei componenti, dei servizi e di altri elementi dell'applicazione.

2.2.4.2.3. Angular Signals

I **signals** di **Angular**$_G$ rappresentano un paradigma reattivo per la gestione dello stato e dei dati all'interno dell'applicazione. Essi consentono di creare flussi di dati reattivi che si aggiornano automaticamente quando i dati sottostanti cambiano, migliorando la reattività e la manutenibilità del codice.

Nel progetto, i **signals** vengono utilizzati per gestire lo stato dell'applicazione in modo efficiente e reattivo, consentendo ai componenti di aggiornarsi automaticamente quando i dati cambiano.

2.2.4.2.4. Organizzazione cartelle

L'organizzazione delle cartelle si basa sul raggruppamento per **tipologia** di file, in cui ogni cartella contiene file con la stessa estensione. In particolare, all'interno di ogni repository, i file sono organizzati in questo modo:

- `src/app/adapters`: contiene le definizioni degli **adapters** utilizzati per tradurre dati inviati dal backend in formati definiti nel frontend;
- `src/app/guards`: contiene le definizioni dei **guards** utilizzati per proteggere le rotte dell'applicazione;

- `src/app/interceptors`: contiene le definizioni degli **interceptors** utilizzati per intercettare le richieste `HTTP` e modificare le intestazioni o gestire gli errori;
- `src/app/models`: contiene le definizioni dei **modelli** utilizzati per rappresentare i dati all'interno dell'applicazione. I modelli sono inoltre raggruppati in sotto-cartelle organizzate per dominio (es. `auth`, `gateway`, `sensor`, etc...);
- `src/app/pages`: contiene le definizioni delle **pagine** dell'applicazione, organizzate in sotto-cartelle per feature (es. `dashboard`, `login`, `tenant`, etc...). Inoltre, all'interno di ogni cartella di feature possono essere presenti le seguenti sotto-cartelle:
  - `pages/components`: contiene le definizioni dei **componenti** utilizzati all'interno della pagina;
  - `pages/dialogs`: contiene le definizioni delle **finestre di dialogo** utilizzate all'interno della pagina;
- `src/app/services`: contiene le definizioni dei **servizi** utilizzati per gestire la logica di business e le comunicazioni con il backend, organizzati in sotto-cartelle per dominio (es. `auth`, `gateway`, `sensor`, etc...);
- `src/app/utils`: contiene le definizioni delle **funzioni di utilità** utilizzate all'interno dell'applicazione;
- `src/environments`: contiene la definizione dei file di **configurazione** dell'applicazione, come ad esempio `environment.ts`.

I file di test (con estensione `.spec.ts`) sono organizzati in modo speculare alla struttura dei file di codice, e si trovano all'interno della stessa cartella del file di codice a cui si riferiscono.

2.2.4.2.5. Convenzioni di nomenclatura

- **File**: utilizzare il formato `kebab-case`, indicando inoltre il tipo di file attraverso un suffisso (es. `dashboard.page.ts`, `auth.service.ts`, `sensor.model.ts`, `login-form.component.ts` etc...);
- **Classi e interfacce**: utilizzare il formato `PascalCase` (es. `DashboardPage`, `AuthService`, `SensorModel`, `LoginFormComponent` etc...);
- **Variabili e metodi**: utilizzare il formato `camelCase` (es. `getUsers()`, `currentUserSession`, etc...);
- **Signals privati**: indicare i signal privati con il prefisso _ (es. `private _userSignal`);
- **Costanti**: utilizzare il formato `UPPER_SNAKE_CASE` per valori immutabili definiti a livello globale o di modulo;

2.2.4.2.6. Strumenti di supporto

- **ESLint**: è lo strumento di analisi del codice Typescript adottato dal team. ESLint fornisce un insieme di regole per identificare problemi di qualità e stile nel codice, migliorando la manutenibilità e la consistenza del progetto.
- **Prettier**: è lo strumento di formattazione del codice Typescript adottato dal team. Prettier applica regole di formattazione rigorose e configurabili per garantire un aspetto uniforme del codice.

2.2.4.3. Stile di codifica Go

In questa sezione vengono definiti gli standard implementativi per lo sviluppo in linguaggio **Go** al fine di garantire l'uniformità del codice prodotto dal team.

2.2.4.3.1. Best practices

- **Funzioni**: Scrivere funzioni e metodi con una singola responsabilità, evitando di creare funzioni troppo lunghe o complesse;
- **Gestione degli errori**: Gestire gli errori in modo esplicito e coerente, utilizzando il pattern di ritorno degli errori di Go;

- **Variabili**: Utilizzare nomi di variabili, funzioni e tipi descrittivi e significativi, evitando abbreviazioni non comuni o ambigue;
- **Commenti**: Commentare il codice dove non può essere autoesplicativo, evitando di riempirlo di commenti superflui;
- **Lingua**: scrivere il codice in inglese per evitare nomi ambigui come `GetUtente()`
- **DIP (Dependency Inversion Principle)**: evitare di dipendere da implementazioni concrete, ma piuttosto da astrazioni, come interfacce o tipi di dati generici.
- **DRY (Don't Repeat Yourself)**: evitare la duplicazione del codice, creando funzioni o metodi riutilizzabili per le operazioni comuni.

2.2.4.3.2. Dependency injection

In ogni microservizio è utilizzato il pattern **Dependency Injection**$_G$ tramite il framework **Uber Fx**$_G$. È fondamentale il suo utilizzo per evitare l'accoppiamento tra i componenti software, migliorare la testabilità e la manutenibilità del codice.

2.2.4.3.3. Architettura esagonale

I seguenti componenti appartenenti all'architettura esagonale seguiranno le seguenti regole:

- **Inbound adapter**: la struttura terminerà con **Controller**;
- **Inbound port**: l'interfaccia terminerà con **UseCase**;
- **Service**: la struttura terminerà con **Service**;
- **Outbound port**: l'interfaccia terminerà con **Port**;
- **Outbound adapter**: la struttura terminerà con **Adapter**;
- **DTO**: la struttura terminerà con **DTO** e potrà essere utilizzata solo negli **inbound adapter**;
- **Entity**: la struttura terminerà con **Entity** e potrà essere creata negli **outbound adapter** ed utilizzata nelle **repository**;

2.2.4.3.4. Organizzazione cartelle

L'organizzazione delle cartelle si basa sul concetto di **Package by Feature**, dove la maggior parte del codice necessario per una feature risiede all'interno della stessa cartella e, conseguentemente, dello stesso **package**.

In ogni microservizio le repository sono organizzate in questo modo:

- cartella **internal**: contiene il codice dell'applicazione, organizzato per feature. Inoltre sono presenti ulteriori cartelle per il codice condiviso tra feature:
  - cartella **infra**: codice condiviso fuori dalla business logic
  - cartella **shared**: codice condiviso all'interno della business logic
- cartella **tests**: contiene i test di unità e di integrazione, organizzati per feature;
- file **.env**: contenente le variabili d'ambiente per la configurazione del microservizio;
- file **main.go**: contiene le istruzioni di avvio del microservizio
- file **<name>.creds**: credenziali per l'accesso a NATS
- file **ca.pem**: certificato per l'accesso a NATS via TLS

2.2.4.3.5. Convenzioni di nomenclatura

- **File**: utilizzare il formato `camelCase` (es. `writeToDatabase.go`, `apiController.go`), eccetto nei file di test in cui bisogna aggiungere il suffisso `*_test.go` (es. `writeToDatabase_test.go`);
- **Struct**: utilizzare il formato `PascalCase` nel caso di esportazione dal package, altrimenti utilizzare il formato `camelCase` (es. `type SensorData struct { ... }`);

- **Variabili e Metodi**: utilizzare il formato `CamelCase` per le variabili e i metodi esportati dal package, altrimenti utilizzare il formato `camelCase` (es. `func GetSensorData() { ... }`, `func writeToDatabase() { ... }`);
- **Costanti**: utilizzare il formato `UPPER_SNAKE_CASE` per valori immutabili definiti a livello globale o di package;

2.2.4.3.6. Strumenti di supporto

- **Gofumpt**: è lo strumento di formattazione del codice Go adottato dal team. Gofumpt applica regole più rigorose rispetto a `gofmt`, come l'aggiunta di spazi bianchi per migliorare la chiarezza, e supporta opzioni configurabili per adattarsi alle preferenze del team.
- **Golangci-Lint**: è lo strumento di analisi del codice Go adottato dal team. Golangci-Lint fornisce un insieme di regole per identificare problemi di qualità e stile nel codice, migliorando la manutenibilità e la consistenza del progetto.
  Gli strumenti di **Golangci-lint** utilizzati sono:
  - **gocyclo**: per misurare la complessità ciclomatica del codice, identificando funzioni e metodi che potrebbero essere troppo complessi e difficili da mantenere;
  - **staticcheck**: per identificare pattern di codice che potrebbero essere migliorati;
  - **errcheck**: per assicurarsi che tutti gli errori siano gestiti;
  - **govet**: per identificare problemi di stile e qualità nel codice;
  - **unused**: per individuare variabili e funzioni non utilizzate;
  - **bodyclose**: per assicurarsi che i body delle richieste HTTP siano chiusi;
  - **noctx**: per evitare l'uso di context.Background() o context.TODO();
  - **asasalint**: per applicare regole di stile e qualità specifiche del team.

3. Processi di supporto

Tra i **processi**$_G$ di supporto applicati nel progetto si distinguono:

- **Documentazione**
- **Gestione delle configurazioni**
- **Accertamento della qualità**
- **Verifica**
- **Validazione**

3.1. Documentazione

Il processo di produzione di documentazione è uno dei processi di supporto più importanti in quanto consente il tracciamento e la storicizzazione delle decisioni prese dal gruppo, permettendo a tutti i membri di rimanere allineati in modalità asincrona su queste ultime e sui metodi di lavoro.

Nello specifico, una decisione viene documentata solamente se questa si può tradurre in un'azione tracciabile e concreta, che necessariamente influisce sul ciclo di vita del prodotto.

Quindi, questo processo riguarda tutte le attività di pianificazione, progettazione, sviluppo, produzione e modifica dei documenti necessari al gruppo per lavorare secondo una linea di lavoro condivisa, e necessari al proponente e alla committente per osservare l'avanzamento del prodotto.

3.1.1. Strumenti a supporto

- **Typst**$_G$: è un **sistema**$_G$ di composizione tipografica digitale, che permette di redigere documenti con un linguaggio di mark-up intuitivo e più semplice di LaTeX. Il **sistema**$_G$ fornisce anche un linguaggio di scripting che permette di scrivere con semplicità macro e procedure che sempli-

ficano la scrittura di contenuti dinamici. La forza di **Typst**$_G$ risiede nel suo **sistema**$_G$ di templating, utilizzato in modo ubiquitario da *GlitchHub Team* e nel reloading istantaneo che consente una scrittura più rapida dei documenti.

- **GitHub**$_G$: per il controllo del versionamento dei sorgenti **Typst**$_G$ dei documenti
  - **GitHub Issues**: per l'assegnazione degli elementi del *backlog* e la segnalazione di eventuali problemi nella **Repository**$_G$, sono fondamentali per il tracciamento delle azioni prese in seguito a decisioni di gruppo
  - **Github Project**: per la visualizzazione delle task in modalità Kanban, utile ad illustrare lo stato d'avanzamento delle task assegnate
  - **GitHub Pages**$_G$: per l'hosting via sito web della documentazione stabile di progetto
- **ClickUp**$_G$: per la condivisione di appunti e documenti interni
- **Google Calendar**: per il tracciamento delle date di riunione
- **Discord**$_G$: per lo svolgimento delle riunioni interne, effettuate da remoto

3.1.2. Attività previste

Le attività previste per la scrittura di documentazione sono:

- **Produzione**: nella Sezione 3.1.5 sono descritte le metodologie che il gruppo applica nella scrittura dei documenti
- **Manutenzione**: nella Sezione 3.1.6 si descrive l'attività di eventuale modifica di un documento considerato **Verificato**

3.1.3. Caratteristiche e struttura dei documenti

Ogni tipo di documento prodotto ha una struttura e caratteristiche diverse, poiché ciascuno ha uno scopo ben preciso.

Per mantenere coerenza tra più documenti dello stesso tipo, mantenendo un processo di scrittura rapido, ogni tipo di documento prevede l'utilizzo di uno specifico template di **Typst**$_G$.

A prescindere dal proprio tipo, il frontespizio di tutti i documenti tranne i diari di bordo è composto da:

- Il logo dell'Università di Padova, in alto a sinistra;
- Il logo di *GlitchHub Team*, in alto a destra;
- Il titolo del verbale, sotto i loghi;
- Al centro della pagina, la versione del documento;
- Lo stato del documento (descritto nella Sezione 3.1.4.1), sotto il titolo;
- La distribuzione del documento, ovvero i destinatari del documento, sotto lo stato.

Inoltre, dalla seconda pagina di ogni documento, vi è presente la tabella del registro delle modifiche, dove ogni riga rappresenta una versione del documento e con le seguenti colonne:

- Codice che descrive la **versione** (vd. Sezione 3.1.4.2)
- **Data** di scrittura della versione;
- **Autore** della versione;
- **Verificatore** della versione;
- **Descrizione** delle modifiche della versione.

Dopo il registro delle modifiche è presente l'indice del documento il quale ne descrive la struttura delle intestazioni.

Infine, sull'ultima pagina è presente la firma del revisore interno del documento.

3.1.3.1. Documenti incrementali

Alcuni documenti, come il presente *Norme di Progetto*, *Piano di Qualifica* e *Piano di Progetto*, sono **incrementali**, cioè crescono parallelamente allo sviluppo del progetto. Per questo motivo alcune sezioni possono rimanere inizialmente vuote o incomplete, poiché non è ancora possibile definirne i concetti.

Il workflow di questi documenti differisce dagli altri «monolitici», infatti ogni **versione stabile** (vd. Sezione 3.1.4.2) considera tali solo le sezioni complete e verificate (vd. Sezione 3.1.5.2), ignorando quelle vuote. Ogni versione stabile deve essere poi pubblicata nel sito tramite *pull request*, così da rendere disponibile il prima possibile lo stato aggiornato del documento.

3.1.3.2. Verbali

I verbali presentano una versione modificata del frontespizio descritto sopra, in cui:

- Compare l'**ordine del giorno** come lista numerata tra il titolo e la versione;
- Tra lo **Stato** e la **Distribuzione**, compare la lista dei **partecipanti** alla riunione descritta dal verbale.

In generale, i verbali hanno la seguente struttura:

- **Introduzione**: Sezione dove si descrivono le coordinate della riunione in oggetto e un riassunto dello scopo della riunione.
- **Resoconto**: Sezione suddivisa in sotto-sezioni corrispondenti agli elementi dell'**ordine del giorno**. L'ultima sezione deve descrivere l'assegnazione dei compiti decisa durante la riunione per il prossimo periodo.
- Eventualmente può anche figurare una **conclusione**.

Si noti che i verbali esterni presentano la firma del revisore esterno del documento, a fianco della firma del revisore interno.

Il template da utilizzare per la scrittura dei verbali è il file `src/Templates/templateVerbali.typ`.

3.1.3.3. Diari di bordo

I diari di bordo sono degli incontri periodici in aula, organizzati dal prof. Vardanega durante l'orario di lezione, in cui ogni gruppo può esporre eventuali dubbi e difficoltà riscontrate durante lo svolgimento del progetto e porre domande al professore e agli altri gruppi, tramite una serie di diapositive brevi.

Le diapositive dei diari di bordo non seguono la struttura descritta precedentemente ma hanno la seguente organizzazione:

- **Difficoltà riscontrate**: Sezione in cui si enunciano le difficoltà riscontrate dal gruppo.
- **Questioni aperte**: Sezione in cui il gruppo pone eventuali domande al professore e/o agli altri gruppi.

La scrittura delle *slides* di un diario di bordo prevede l'utilizzo del template `src/Templates/templateSlides.typ`

3.1.3.4. Altri documenti

Tutti gli altri documenti prodotti non seguono una struttura specifica, poiché la struttura di ogni documento dipende dalla sua funzione. Ciononostante, è importante utilizzare il template `src/Templates/templateDocumentiGenerici.typ` per la scrittura di tali documenti, poiché quest'ultimo fornisce varie macro e funzioni di **Typst**$_G$ utili.

3.1.4. Convenzioni

Il gruppo si impegna ad attuare sempre le seguenti convenzioni durante la redazione di documentazione in modo tale da mantenere una linea di lavoro consistente e coerente nel tempo.

3.1.4.1. Stato di un documento

Un documento attraversa tre stadi principali durante la sua scrittura:

3.1.4.1.1. «Bozza»

Un documento è una **«bozza»** quando è in fase di scrittura, per cui non è ancora da verificare, finché l'autore corrente non richiede una verifica;

3.1.4.1.2. «Da verificare»

Un documento è **«da verificare»** quando ciò che è stato scritto è considerato completo e pronto per essere verificato;

3.1.4.1.3. «Verificato»

Un documento è **«verificato»** quando un componente del gruppo diverso dallo scrittore ha controllato e sottoscritto il documento, sancendo che ciò che è stato scritto rappresenta in modo veritiero le decisioni, le azioni e le opinioni dei membri del gruppo.

3.1.4.2. Versionamento

Essendo la produzione di documentazione un processo iterativo, è necessario tenere traccia di ogni versione dei documenti prodotti, in modo tale da tracciare, nel tempo, quali modifiche vengono applicate a essi e da chi.

Per maggiori informazioni sul versionamento della documentazione, si consulti la Sezione 3.2.4.

3.1.4.3. Denominazione e locazione file

I documenti in formato PDF sono resi disponibili sul sito web di *GlitchHub Team*.

Tutti i documenti, fatta eccezione per i diari di bordo e il glossario, seguono il seguente schema di collocazione all'interno dell'alberatura del dominio del sito web del gruppo:

`/pdf/`**`[FASE]`**`/`**`[TIPO DOCUMENTO]`**`/`**`[NOME DOCUMENTO]`**`.pdf`

Dove:

- **`[FASE]`** è una tra `Candidatura`, `RTB` e `PB`, a seconda della fase della baseline di progetto a cui appartiene il documento
- **`[TIPO DOCUMENTO]`** corrisponde a:
  - `VerbaliInterni` per i verbali interni
  - `VerbaliEsterni` per i verbali esterni
  - `Generale` per tutti gli altri documenti
- **`[NOME DOCUMENTO]`** rappresenta il nome del file pdf che segue uno standard diverso a seconda del documento:
  - I verbali hanno come nome il giorno della riunione che riassumono in formato **`YYYY-MM-DD`**, dove `YYYY` rappresenta l'anno per esteso, `MM` il numero del mese (`01`–`12`) con 2 cifre e `DD` (`01`–`31`) il giorno con 2 cifre.
  - Gli altri documenti hanno invece `[NOME DOCUMENTO]` pari al nome del tipo di documento in stile CamelCase, ad esempio:
    - `LetteraPresentazione` per la lettera di presentazione
    - `ValutazioneCapitolati` per la valutazione dei capitolati
    - `DichiarazioneImpegni` per la dichiarazione di impegni

- `NormeProgetto` per il documento di norme di progetto
- `PianoDiProgetto` per il piano di progetto.
- `glossary` per il glossario

Si noti che l'utilizzo delle preposizioni (ad es. «di» in «Norme **di** progetto») è facoltativo.

Fanno da eccezione i diari di bordo che sono collocati nella cartella `Slide`;

Si noti che sul sito web il gruppo rende disponibile sempre l'**ultima** versione **verificata** (vd. Sezione 3.1.4.1.3) dei documenti della *baseline* corrente.

3.1.4.4. Metadati

Uno dei punti di forza di **Typst**$_G$ è la funzionalità di codificare metadati all'interno del codice sorgente di un documento. Infatti, tutti i template utilizzati dal gruppo (descritti nella Sezione 3.1.3) richiedono la descrizione di una serie di metadati precisi per ogni documento.

3.1.4.4.1. `htmlId`

Il metadato `htmlId` permette di specificare a quale sottosezione della **pagina web del gruppo** associare l'ancora verso il PDF di questo documento.

Si deve avere cura nell'utilizzare valori di `htmlId` che corrispondano con il layout HTML della pagina.

3.1.5. Produzione

La produzione di un documento avviene nei seguenti passaggi:

1. **Scrittura**, descritta nella Sezione 3.1.5.1
2. **Verifica**$_G$, descritta nella Sezione 3.1.5.2

3.1.5.1. Scrittura

Nel **verbale interno del 27 ottobre 2025**, sono specificate le procedure scelte da *GlitchHub Team* per la scrittura dei verbali, le quali si possono adattare alla scrittura di qualunque documento.

I passaggi descritti sono i seguenti:

1. All'interno della **repository**$_G$ di documentazione di **GitHub**$_G$, viene aperta una **issue** che descrive il compito di scrittura: ciò ha lo scopo di tenere traccia dell'azione e della responsabilità di chi la compie.
2. Quindi, nella **repository**$_G$ si crea un branch secondario nominato `Documentation-[nome attività]`: è importante che questa convenzione sia seguita, poiché all'interno della **repository**$_G$ vengono utilizzate delle automazioni **GitHub Actions**$_G$ che permettono la compilazione dei file sorgenti **Typst**$_G$ in PDF.
3. La stesura del documento avviene sul branch apposito usando Visual Studio Code con l'estensione **Tinymist Typst**, che consente di vedere un'anteprima del documento che si scrive. Inoltre, è richiesto il rispetto delle convenzioni descritte nella Sezione 3.1.4.
4. Una volta terminata la prima stesura, si deve fare il *push* delle proprie modifiche sul branch di lavoro e creare una nuova *Pull Request* in modo tale da fare il *merge* da `Documentation-[...]` a `main`, assegnando i verificatori scelti come Reviewer.

3.1.5.2. Verifica

Ogni documento scritto deve essere verificato da un componente del gruppo diverso dallo scrittore originale, per evitare di travisare, intenzionalmente o meno, le decisioni e azioni prese dal gruppo.

Le procedure di verifica sono riportate di seguito, come descritte nel **verbale interno del 27 ottobre 2025** e nel **verbale interno del 12 novembre 2025**.

1. Quando il primo redattore contrassegna il documento come **Da verificare** (vd. Sezione 3.1.4.1.2), quest'ultimo contatta il verificatore scelto per iniziare la procedura di verifica
2. Quindi, il verificatore scelto ha il compito di leggere attentamente il documento e di suggerire correzioni all'autore originale, se necessario

   a. Se il verificatore suggerisce delle correzioni l'autore è tenuto ad applicarle, quindi, il verificatore dovrà controllare la nuova versione corretta. Questo passo avviene in maniera ciclica, finché il documento non è considerato stabile.

   b. Se invece, esso ritiene che il documento sia corretto e non richieda ulteriori modifiche, il documento è da considerarsi stabile.

Si noti che nel caso di **documenti incrementali**$_G$, la verifica deve avvenire a ogni versione intermedia, considerata «stabile» dall'autore.

Nel caso di questo documento, le versioni «intermedie» sono considerabili incomplete, poiché non tutte le sezioni sono presenti, ma le sezioni che sono già state scritte sono da considerarsi «stabili» e quindi da verificare.

3.1.5.3. Pubblicazione

Una volta che il documento da pubblicare è stato verificato, la relativa **Pull Request**$_G$ dev'essere accettata, eseguendo così il *merge* del documento nel **branch**$_G$ principale (`main`).

Ciò darà inizio all'esecuzione automatica della **GitHub Action**$_G$ che compila il codice Typst del documento in PDF e pubblica il file sul sito web del gruppo.

3.1.6. Manutenzione

Un documento stabile non è da considerarsi finito e immutabile, poiché in seguito alla sua verifica potrebbero emergere errori che non sono stati rilevati né dallo scrittore che dal verificatore.

In tal caso, chi si accorge dell'errore è tenuto a segnalarlo ed eventualmente proporre una modifica, che contribuirà a creare una nuova versione del documento, la quale andrà successivamente verificata da qualcun altro.

3.2. Gestione delle configurazioni

Secondo lo standard **ISO/IEC 12207:1995**$_G$, la **gestione delle configurazioni** è il processo di applicazione di procedure di natura tecnica e amministrativa durante tutto il ciclo di vita del software con i seguenti scopi:

- identificare e definire le parti di un prodotto software e associarle a specifiche **baseline**$_G$
- controllare le modifiche e il rilascio di tali parti;
- registrare e riportare il loro *status* e di eventuali richieste di modifica ad esse;
- assicurare la loro completezza, coerenza e correttezza.

3.2.1. Strumenti a supporto

Il principale strumento a supporto del processo di gestione delle configurazioni è **GitHub**$_G$, il quale permette di gestire il versionamento e i cambiamenti da effettuare di codice e documentazione.

Per ulteriori informazioni sullo strumento si consiglia la lettura della Sezione 4.2.2

3.2.2. Attività previste

Le attività previste dal processo sono:

- **Identificazione della configurazione**
- **Controllo della configurazione**
- **Registrazione dello stato di configurazione**
- **Valutazione della configurazione**

3.2.3. Identificazione della configurazione

L'attività di **identificazione della configurazione** consiste nell'individuazione di tutte le componenti, sia documentali che composte da codice, che formeranno il prodotto da sviluppare.

Secondo lo standard **ISO/IEC 12207:1995**$_G$, è necessario stabilire per ogni prodotto di software e le sue versioni:

- La documentazione che ne stabilisce la **baseline**$_G$
- I riferimenti alla versione;
- Altri dettagli di identificazione.

Nello specifico, questa attività avverrà durante la fase di progettazione, in cui verrà schematizzata l'architettura del *software*, che poi verrà implementata dai **programmatori**$_G$.

Per quanto riguarda l'identificazione della configurazione della documentazione, si consulti la Sezione 3.1.

3.2.4. Controllo della configurazione

L'attività di **controllo della configurazione** disciplina le richieste di modifica alla documentazione o al codice, le quali potranno dovranno essere accettate o meno.

A tale scopo, *GlitchHub Team* ha deciso numerosi strumenti offerti da **GitHub**$_G$, descritti di seguito.

- **GitHub Issues**$_G$: Il gruppo utilizza le *issues* per descrivere le modifiche da apportare a documentazione e codice; una *issue* dev'essere assegnata al singolo membro che si assume la responsabilità di portare a termine la modifica. Una qualunque *issue* è composta da:
  - Un identificativo numerico univoco all'interno della **repository**$_G$ in cui è stata creata;
  - Un nome che descrive brevemente la modifica da compiere;
  - Una descrizione facoltativa che fornisce ulteriori informazioni sull'attività da svolgere;
  - L'assegnatario che si assumerà la responsabilità di compiere la *task*;
  - Una serie di etichette o «*label*» che descrivono il tipo di modifica da compiere;
  - Il collegamento con uno o più **GitHub Projects**$_G$
  - La **milestone**$_G$ a cui corrisponde la modifica;
  - La relazione dell'*issue* con altre *issue*.

  Per maggiori informazioni sull'utilizzo delle *issue* durante l'organizzazione delle attività, si consulti la Sezione 4.1.
- **GitHub Project**$_G$: Il gruppo utilizza un *project* collegato a tutte le **repository**$_G$ istituite dal gruppo, per organizzare le attività negli *sprint* in modo da massimizzare la parallelizzazione del lavoro. Infatti, il *project* usato permette di associare a ogni *issue* dei campi aggiuntivi che facilitano una buona organizzazione temporale delle *task* da compiere, e di visualizzare le *issue* in *board* **Kanban**$_G$ e in diagrammi di **Gantt**$_G$.
- **Pull Request**$_G$ (PR) e ***branch protection***: All'interno delle **repository**$_G$ del gruppo, il **branch**$_G$ `main` è **protetto**, ovvero non è possibile applicarvi direttamente delle modifiche: per fare ciò

è necessario utilizzare le *PR*. Queste, infatti, sono uno strumento che necessitano che tutte le modifiche applicate al branch principale siano verificate da un altro membro del gruppo detto *Reviewer*, il quale può fornire l'approvazione diretta o richiedere un'ulteriore revisione. Questo strumento è fondamentale per un buon controllo della configurazione, in quanto costringe tutti i membri del gruppo a lavorare con un *way of working* corretto che considera la verifica come passo fondamentale di qualunque attività di progetto.

3.2.5. Registrazione dello stato di configurazione

L'attività di **registrazione dello stato di configurazione** consiste nel tracciare lo stato e la storia di tutte le componenti del progetto. A tale scopo, il gruppo ha istituito una convenzione di versionamento della documentazione, riportata di seguito come descritta originariamente nel **verbale interno del 27 ottobre 2025**.

Il gruppo ha deciso di adottare il sistema di versionamento **Semantic Versioning**$_G$ (abbreviato SemVer) per i documenti, in cui ogni versione è descritta da 3 numeri naturali separati da punto (**`MAJOR.MINOR.PATCH`**), dove:

- **`MAJOR`**: rappresenta le modifiche sostanziali applicate al documento
  - Se `MAJOR = 0`, la versione è da considerarsi non «stabile», ovvero soggetta in futuro a cambiamenti rapidi e sostanziali. In particolare, ciò rappresenta che il documento è in fase di prima stesura;
  - Se `MAJOR = 1`, allora il documento è nella versione di prima stesura «stabile», ovvero **verificata** da un altro membro del gruppo;
  - Ogni valore `MAJOR >= 1`, rappresenta una nuova stesura del documento sostanzialmente diversa dal precedente. Per cui, il numero `MAJOR` va aumentato solo in caso di modifiche sostanziali alla struttura o al contenuto del documento.
- **`MINOR`**: va aumentato a ogni nuova revisione sostanziale che non modifica la struttura o il significato del contenuto del documento,
- **`PATCH`**: va aumentato a ogni revisione che corregge la formattazione, refusi o punteggiatura del testo.

All'incremento di un numero di versione, tutti i numeri alla sua destra vengono messi a 0, per cui la versione `MINOR` successiva alla `0.1.1` non può essere `0.2.1`, ma deve essere necessariamente `0.2.0`.

In questo sistema, quindi:
- la prima **bozza** del documento è rappresentata dalla versione `0.0.1`,
- la prima **stesura** ancora non verificata dalla versione `0.1.0`
- la prima **stesura** verificata dalla versione `1.0.0`
- e così via, fino al completamento del documento

Inoltre, ogni documento versionato presenterà il proprio **registro delle modifiche**, descritto nella Sezione 3.1.3.

3.2.6. Valutazione della configurazione

L'attività di **valutazione della configurazione** consiste nel controllo che il software prodotto presenti una completezza funzionale rispetto ai requisiti rilevati.

A tale scopo, il documento di **analisi dei requisiti**$_G$ presenta una sezione di **tracciamento dei requisiti** che verrà usata dal gruppo durante la fase di progettazione e sviluppo come riferimento per verificare che il software prodotto sia adeguato alle aspettative e richieste della **proponente**$_G$.

3.3. Accertamento della qualità

Secondo lo standard **ISO/IEC 12207:1995**$_G$, il processo di **accertamento qualità** consiste nell'accertare in modo adeguato che i processi e i prodotti dei cicli di vita del progetto siano conformi ai requisiti specificati.

Perché questo processo sia privo di *bias* e di conflitti d'interesse, è fondamentale che avvenga in modo svincolato dalle persone responsabili dell'applicazione del processo controllato o dello sviluppo del componente controllato.

3.3.1. Attività previste

Le attività previste dal processo sono le seguenti:

- **Implementazione del processo**: L'implementazione consiste nella creazione di un processo di accertamento della qualità su misura per il progetto in modo tale da garantire che i prodotti di progetto siano conformi alle attese e ai requisiti rilevati.
- **Accertamento della qualità di prodotto**: Consiste nel controllare che il software prodotto rispetti i requisiti rilevati, tramite i test prodotti dai **verificatori**$_G$ e la misurazione delle metriche di qualità del prodotto.
- **Accertamento della qualità di processo**: Consiste nel controllo e nella misurazione delle metriche di qualità dei processi, per assicurarsi che i processi rispettino gli standard di qualità stabiliti dal gruppo.

Per svolgere le attività di **accertamento della qualità di prodotto e processo**, il gruppo ha stabilito nel documento di **Piano di Qualifica**$_G$ le metriche di qualità di prodotto e processo e il **cruscotto di valutazione** che riporta l'andamento delle misurazioni di tali durante lo svolgimento del progetto. Si noti che queste misurazioni vanno effettuate al termine di ogni *sprint*.

3.4. Verifica

Il processo di **verifica** ha lo scopo di determinare se un prodotto di progetto sia conforme ai requisiti e rispetti le condizioni imposte sul prodotto dalle attività precedenti: l'obiettivo del processo si può sintetizzare nella domanda *«Did I build the system **right**?»*, ovvero «Ho costruito il sistema **correttamente**?».

In generale, gli esiti di questo processo, ovvero le misurazioni delle metriche di qualità, sono racchiuse nel documento di **Piano di Qualifica**$_G$.

3.4.1. Attività previste

Le attività previste da questo processo, secondo lo standard **ISO/IEC 12207:1995**$_G$ sono le seguenti:

- **Implementazione del processo**: L'attività di implementazione consiste nella rilevazione dei processi e prodotti di progetto che presentano criticità e/o richiedono di essere verificati, e nell'istituzione e documentazione di un processo di verifica che documenti e risolva tutte le non conformità da esso rilevate.
- **Attività di verifica**: Consiste nell'applicazione del processo stabilito dal passaggio precedente, che ha lo scopo di controllare l'efficacia di:
  - **processi**$_G$, i quali devono rispettare gli standard stabiliti dal gruppo e dalla **proponente**$_G$
  - **requisiti**$_G$, i quali devono essere coerenti con le aspettative della **proponente**$_G$, fattibili e verificabili;
  - **progettazione**$_G$, la quale deve rispecchiare tutti i requisiti rilevati ed essere ad essi tracciabile, e dev'essere corretta e coerente;

- **codice**$_G$, il quale deve rispettare progettazione e requisiti ed essere ad essi tracciabile, e dev'essere verificabile e rispettoso degli standard di codifica del *team*;
- **integrazione**$_G$ del sistema, la quale deve consentire a tutte le componenti del software prodotto di essere completamente integrabili e compatibili tra loro;
- **documentazione**$_G$, la quale dev'essere adeguata, completa e coerente, e dev'essere prodotto con puntualità.

3.4.2. Implementazione del processo

Il gruppo è giunto alla conclusione che, per quanto concerne il processo di verifica, la priorità principale è soddisfare le aspettative della **proponente**$_G$: ciò è possibile tramite la realizzazione di test approfonditi che garantiscano il più possibile la conformità con le attese di **M31**.

Il principale punto critico che è stato rilevato sin da subito dal gruppo è la necessità che il **branch**$_G$ principale di tutte le **repository**$_G$ usate dal gruppo contenga solamente elementi corretti.

Infatti, tale branch deve soltanto contenere tutti gli elementi che soddisfano la **baseline**$_G$ corrente, ovvero tutti i prodotti di progetto che sono stati adeguatamente verificati e quindi considerabili corretti: ciò è ampiamente facilitato dalle regole di *branch protection* di **GitHub**$_G$, descritte nella Sezione 3.2.4.

Nella Sezione 3.4.3 si descrive i dettagli dell'attività di verifica applicata del gruppo durante lo svolgimento del progetto.

3.4.3. Attività di verifica

Per quanto concerna la verifica della **documentazione**, il gruppo ha determinato sin da subito la necessità di processi di applicare ad essa dei processi di verifica rigidi e tempestivi che possano garantirne la correttezza formale e contenutistica. Infatti, lo scopo di verificare i documenti redatti è di garantire che vengano rappresentate correttamente le opinioni e le decisioni prese dal gruppo, onde evitare malintesi durante il dialogo interno al gruppo e con la **proponente**$_G$.

Per informazioni più dettagliate sul processo di verifica della documentazione, si consulti la Sezione 3.1.5.2.

Per quanto concerne la verifica del **codice** prodotto, questa sezione verrà ampliata una volta iniziate le attività di sviluppo dell'**MVP**$_G$, in quanto il codice prodotto per il **PoC**$_G$ non richiede di essere testato tramite le procedure descritte successivamente.

Tutte le informazioni specifiche relative alla verifica verranno riportate nel documento di **Piano di Qualifica**$_G$. In generale, però, si può applicare il processo di verifica al codice sorgente del *software* tramite l'**analisi statica** (descritta nella Sezione 3.4.3.1) e l'**analisi dinamica** (descritta nella Sezione 3.4.3.2).

3.4.3.1. Analisi statica

L'**analisi statica** comprende tutte le attività di verifica che non prevedono l'esecuzione dell'oggetto di test, ma si concentra sulla sintassi e la correttezza del contenuto di quanto scritto con lo scopo di rilevare problemi prima che si possano presentare durante la sua esecuzione.

Questa può avvenire usando **metodi formali**, i quali dimostrano formalmente che l'oggetto in esame soddisfa specifiche proprietà, e tramite **metodi di lettura**, i quali sono meno formali ma comunque efficaci nel rilevare problematiche nell'oggetto di test.

I principali **metodi di lettura** sono:

- ***Walkthrough***: Metodo di analisi che cerca di rilevare difetti leggendo l'oggetto in esame con una modalità ad ampio spettro: quindi, si è consapevoli dell'esistenza di un problema ma non si quale questo possa essere e dove trovarlo. Questo metodo è costoso e poco applicabile perché non automatizzabile, ma potenzialmente molto efficace.
- ***Inspection***: Metodo di analisi che invece rileva la presenza di difetti tramite una lettura mirata dell'oggetto: quindi, si sa quali siano i problemi da rilevare e i test si concentrano esclusivamente sulla rilevazione di questi ultimi. Questo metodo si rivela meno costoso e più facilmente automatizzabile, ma non è in grado di determinare se l'oggetto in esame è totalmente corretto, in quanto non si può avere completa certezza che i test scelti siano esaustivi.

In generale, i metodi di *inspection* sono preferibili per il progetto in quanto sono realizzabili tramite la verifica di *checklist*, ovvero di liste di controllo automatizzabili che possono rilevare gli errori più frequenti nella scrittura del codice.

3.4.3.2. Analisi dinamica

L'**analisi dinamica** comprende tutte le attività di verifica che prevedono l'esecuzione dell'oggetto in esame, in modo tale da eliminare i *fault*, ovvero gli elementi del prodotto che hanno un comportamento inatteso («failure») rilevato durante l'esecuzione del codice. Essendo il software immutabile di natura, possono essere introdotti in esso tali *fault* solamente a causa di errori umani.

Quindi, l'**analisi dinamica** consiste nell'esecuzione di più oggetti di prova, detti *test*, che corrispondono a diverse esecuzioni degli oggetti di prova. Ogni *test* studia il comportamento di una singola parte di codice su un insieme finito di casi di prova. I test dinamici devono essere:

- **ripetibili**: ovvero eseguibili con gli stessi esiti su diversi ambienti di esecuzione, in modo tale che solo le modifiche al codice possano causare esiti diversi dei test.
- **automatizzabili**: ovvero eseguibili automaticamente dalla macchina, in quanto la procedura di esecuzione manuale dei test è uno spreco di risorse, essendo essa pienamente automatizzabile tramite l'utilizzo di:
  - *driver*: componente attiva fittizia che eseguono il test, sostituendosi alla procedura «`main()`» del programma
  - *stub*: componenti passive fittizie che simulano parti del sistema esterne all'oggetto in esame ma utili per l'esecuzione del *test* stesso
  - *logger*: componente secondario che registra l'esito dei test senza interferire con la loro esecuzione

I principali tipi di test sono i seguenti:

- **Test di unità**
- **Test d'integrazione**
- **Test di sistema**
- **Test di regressione**
- **Test di accettazione**, i quali non corrispondono propriamente a verifiche del codice ripetibili e automatizzabili, ma sono inclusi per completezza.

La convenzione usata da *GlitchHub Team* per identificare i test è la seguente:

`T[Tipo]-[Numero]`

Dove:

- **`T`** è l'abbreviazione di «Test»
- **`[Tipo]`** è una lettera che indica la tipologia di test

- `U` per i test di Unità
- `I` per i test d'Integrazione
- `S` per i test di Sistema
- `A` per i test di Accettazione

- **`[Numero]`** è un numero identificativo incrementale univoco per la tipologia di test (per cui possono esistere contemporaneamente i test `TU-1` e `TS-1`).

Inoltre ogni test si può trovare in ogni momento in uno di quattro stati:
- **NI** ovvero **Non implementato**, se il test è stato ideato ma non implementato in codice;
- **I** ovvero **Implementato**, se il test è stato implementato ma non ancora eseguito;
- **NS** ovvero **Non superato**, se il test è stato implementato, eseguito e non superato;
- **Superato**, se il test è stato implementato, eseguito e superato correttamente.

3.4.3.2.1. Test di unità

Un **unità** è definita come la più piccola quantità di software che sia sufficientemente grande da essere verificata in quanto oggetto singolo. Quindi, un componente di un *software* è composto da un insieme di **unità** tra loro integrate, le quali sono composte da uno o più **moduli** cadauna.

I **test di unità**, quindi, verificano la correttezza delle **unità** del *software* e si dividono in due sotto-categorie principali:

- **Test funzionali** (*black-box*), i quali verificano solamente che a ogni input preso in considerazione corrisponda l'output corretto senza considerare la logica interna dell'unità, da cui il nome *«black-box»*. Poiché il dominio di un valore in input può essere potenzialmente infinito, ma il tempo a disposizione per eseguire un test è necessariamente finito, un buon test di unità viene eseguito usando come input un valore per ognuna delle seguenti *classi d'equivalenza*:
  - Classe dei **valori nominali** nel dominio, ovvero tutti i valori validi che l'input può assumere
  - Classe dei **valori illegali inferiori**, ovvero tutti i valori oltre il limite **inferiore** del dominio
  - Classe dei **valori illegali superiori**, ovvero tutti i valori oltre il limite **superiore** del dominio
  - Classe dei **valori legali d'estremo inferiore**, ovvero tutti i valori al confine tra il dominio e il suo limite inferiore
  - Classe dei **valori legali d'estremo superiore**, ovvero tutti i valori al confine tra il dominio e il suo limite superiore
- **Test strutturali** (*white-box*), i quali verificano la logica interna del codice dell'oggetto di verifica, misurando quanti *statement*, *branch* e *decision* vengono eseguiti all'interno di ogni test.

3.4.3.2.2. Test d'integrazione

I **test d'integrazione** verificano il corretto assemblaggio delle componenti del *software* individuate nella fase di *design* architetturale, rilevando difetti di progettazione o problemi di qualità nei test di unità.

Per verificare una corretta integrazione delle componenti, è necessario eseguire il loro assemblaggio in maniera incrementale e reversibile, utilizzando una delle seguenti strategie:

- **Bottom–up**, in cui si integrano prima le componenti con minori dipendenze e maggiore utilità interna, richiedendo una minore quantità di *stub* durante lo sviluppo, ma mostrando risultati all'utente con maggiore ritardo;
- **Top–down**: in cui, invece, si integrano prima le componenti con maggiori dipendenze d'uso e maggiore utilità esterna, richiedendo l'uso di molti più *stub* durante lo sviluppo, ma rendendo visibili le funzionalità usabili dall'utente molto prima.

3.4.3.2.3. Test di sistema

I **test di sistema** verificano la conformità funzionale del sistema rispetto ai requisiti stabiliti nel documento di **analisi dei requisiti**$_G$. Infatti, la loro stesura avviene in concomitanza con la stesura dei requisiti funzionali del prodotto.

3.4.3.2.4. Test di regressione

I **test di regressione** verificano che alcune correzioni o estensioni applicate a specifiche unità non causino *fault* in parti del sistema esterne all'oggetto in esame. Infatti, di fronte a un problema del genere è necessario valutare le necessità di modifica del sistema e selezionare la soluzione che offra il miglior rapporto costi/benefici.

Nel concreto, i test di regressione consiste nell'esecuzione ripetuta di un sottoinsieme di test di unità, test d'integrazione e test di sistema che hanno causato in passato problemi di regressione.

3.4.3.2.5. Test di accettazione

I **test di accettazione** verificano la conformità del prodotto rispetto ai *requisiti utente* definiti dal **committente**$_G$ nel **capitolato d'appalto**$_G$. Questi test, al contrario, degli altri non possono avvenire in maniera automatizzabile e ripetibile, in quanto vengono eseguiti manualmente in presenza del **committente**$_G$ stesso, ma vengono comunque inclusi nella Sezione 3.4.3.2 per completezza.

3.5. Validazione

Il processo di **validazione** ha l'obiettivo di determinare se i requisiti rilevati e il software prodotto rispettano le aspettative ed esigenze dell'azienda **proponente**$_G$ ciò si può riassumere nella ricerca della risposta alla domanda *«Did I build the **right** system?»*, ovvero «Ho costruito il **giusto** sistema?».

3.5.1. Attività previste

In base allo standard **ISO/IEC 12207:1995**$_G$, le attività previste dal processo sono le seguenti:

- **Implementazione del processo**
- **Attività di validazione**

3.5.2. Implementazione processo

GlitchHub Team ha condotto uno studio approfondito delle richieste della **proponente**$_G$, riassumendo tutti i requisiti da soddisfare nel documento di **analisi dei requisiti**$_G$.

Come già descritto nella Sezione 3.2.6, l'**Analisi dei Requisiti**$_G$ presenta una sezione di **tracciamento dei requisiti**, fondamentale al processo di **validazione**. Infatti, questa permette di controllare se una parte del prodotto funziona correttamente ed è conforme ai requisiti: un requisito si può considerare soddisfatto solo se il codice che lo implementa funziona correttamente e siccome i requisiti sono tracciati, è possibile verificare quali requisiti vengono soddisfatti dal codice e quali no.

Per un tracciamento efficace, *GlitchHub Team* ha deciso di raccogliere tutti i requisiti e i relativi test in relativi file JSON all'interno della cartella `src/tracciamento` della **repository**$_G$ di documentazione. I file JSON all'interno della cartella sono i seguenti:

- `RF.json` che tiene traccia di tutti i requisiti funzionali
- `RNF.json` che tiene traccia di tutti i requisiti non funzionali
- `RD.json` che tiene traccia di tutti i requisiti di dominio
- `TS.json` che tiene traccia di tutti i test di sistema
- `TA.json` che tiene traccia di tutti i test di accettazione

In generale, ogni **requisito** tracciato nei file è descritto da:

- una stringa identificativa univoca (campo «`id`»)
- una descrizione testuale (campo «`desc`»)
- una stringa che descrive l'urgenza del requisito (campo «`urgenza`»)
- una lista che contiene tutti i riferimenti agli use case identificati nell'**Analisi Dei Requisiti**$_G$ (campo «`ref_uc`»)
- una lista che contiene tutti i riferimenti a requisiti del **capitolato**$_G$ (campo «`ref_capitolato`»)

I **test** tracciati, invece, sono descritti nella struttura JSON da:
- Analoghi campi `"id"` e `"descr"`
- Il riferimento al requisito (rilevato dal gruppo o di capitolato) (campo «`ref-req`»)
  - Se il campo si riferisce a un requisito rilevato dal gruppo, allora esso deve contenere l'`id` del requisito considerato
- Lo stato del test, come descritto nella Sezione 3.4.3.2 (campo `"stato"`)

Descrivere i requisiti e i test di sistema con una serie di file strutturati permette di tracciare i requisiti e i test in maniera totalmente automatica, facilitando le misurazioni delle metriche concernenti i requisiti.

3.5.3. Attività di validazione

Per applicare il processo di validazione, il gruppo si impegna a continuare l'attività di **tracciamento dei requisiti** e ad effettuare test di accettazione per verificare la conformità del prodotto con le attese della **proponente**$_G$, ricercando attivamente da essa la maggior quantità di *feedback* possibile relativamente alla conformità del lavoro svolto.

4. Processi organizzativi

I **processi organizzativi** avvengono parallelamente ai processi di progetto contribuendo a un buon andamento di quest'ultimo. Le attività previste da questi processi consentono di migliorare la strutturazione e l'organizzazione dei processi di cicli di vita applicati e di facilitare l'adozione di atti di miglioramento nei confronti di questi ultimi.

Nello specifico, si identificano i seguenti processi organizzativi:
- **Gestione dei processi**
- **Infrastruttura**
- **Miglioramento dei processi**
- **Formazione**

4.1. Gestione dei processi

La **gestione dei processi** è composta dalle attività compiute dal gruppo per pianificare e organizzare in modo efficace ed efficiente i compiti che ciascun membro deve svolgere, in modo tale che non vengano causati ritardi nello svolgimento di tali.

4.1.1. Strumenti a supporto

- **GitHub Issues**$_G$: Il gruppo utilizza l'Issue Tracking System nativo di **GitHub**$_G$ per classificare le attività da compiere e assegnarle ai membri del gruppo. Con l'utilizzo delle **GitHub Actions**$_G$, la loro creazione è ampiamente facilitata.
- **GitHub Actions**$_G$: Il gruppo utilizza le **GitHub Actions**$_G$ per automatizzare la creazione di *branch* specifici per ogni *issue*, in modo tale da facilitare l'applicazione dei workflow di verifica tramite **pull request**$_G$. Sono inoltre presenti altre automazioni che facilitano la gestione delle *issue*, come ad esempio la compilazione automatica di alcuni campi.

- **GitHub Project**$_G$: Il gruppo lo utilizza poiché le *issues* semplici presentano una serie limitata di campi compilabili. I *projects* permettono di rendere le *issues* più descrittive, aggiungendo ad esse ulteriori campi quali la data d'inizio e la data di fine dell'attività, le ore di lavoro previste e le ore effettive. Inoltre, i *projects* permettono di raggruppare le issue provenienti da diverse **repository**$_G$.
- **Pull Request**$_G$: Il gruppo utilizza il sistema di *Pull Request* di **GitHub**$_G$ come strumento di verifica per garantire che tutte le modifiche al codice e alla documentazione siano verificate da un altro membro del gruppo, in modo tale da assicurare la conformità delle modifiche effettuate rispetto agli standard di qualità stabiliti dal gruppo.

4.1.2. Attività previste

Le attività previste nella gestione di processi sono le seguenti:

- Avvio dell'attività e definizione della portata
- Pianificazione
- Esecuzione e controllo
- Revisione e valutazione
- Conclusione

Si noti che la descrizione di questo processo riguarda le attività rendicontabili il cui sviluppo produce prodotti di progetto «esterni», ovvero tutto il codice e la documentazione richiesta dal capitolato e dalle specifiche del progetto didattico. Inoltre, in ogni successiva sottosezione si riportano i passaggi da seguire per gestire le **GitHub Issues**$_G$ relative a specifiche *task*, secondo quanto deciso dal gruppo nel **verbale interno del 30 gennaio 2026** e nel **verbale interno del 24 febbraio 2026**.

Le attività non rendicontabili o di «palestra», ovvero il cui svolgimento non influisce sul budget fissato dal gruppo, seguono un ciclo di vita simile ma che spesso non comprende la fase di revisione e valutazione e una fase di conclusione più semplificata, ma ciononostante vengono tracciate con le **GitHub Issues**$_G$.

4.1.3. Avvio dell'attività e definizione della portata

Innanzitutto, è necessario stabilire i requisiti e le risorse necessarie per completare l'attività. Dopodiché il **responsabile**$_G$ del gruppo deve stabilire la fattibilità del processo controllando la disponibilità delle risorse del gruppo, ovvero controllando che l'attività sia fattibile nell'intervallo temporale fissato. Si noti che i requisiti del processo possono essere discussi con i loro assegnatari.

Il **responsabile**$_G$ individua le attività che il gruppo deve svolgere e le divide in «*task*» assicurandosi che queste siano **atomiche**, **rapide** ed **eseguibili singolarmente**. A ogni *task* corrisponde un'*issue* e un gruppo di *task* correlate possono essere raggruppate in una *parent issue* che ha come assegnatari tutte le persone coinvolte, ma che non viene utilizzata per il conteggio delle risorse consumate per il suo svolgimento. Una *parent issue* è riconoscibile dalla presenza della label `epic`.

Alla creazione di un'*issue* deve essere specificato nel titolo la **sigla** del documento a cui fa riferimento, inserendo tale sigla tra parentesi quadre: in base a ciò un'automazione assegnerà la nuova *issue* alla rispettiva *parent issue*. Le convenzioni di nomenclatura utilizzate per i titoli delle *issue* sono descritte nella Sezione 4.2.3.5.5.

Si noti che le *issue* in questo stato devono essere associate allo stato di **«backlog»**, il quale indice che l'issue è stata rilevata ma ancora non pianificata.

4.1.4. Pianificazione

Per pianificare un'attività è fondamentale comprendere quali siano le risorse richieste, sia temporali che economiche, e le singole *task* di cui è composta. Poiché le attività del progetto richiedono competenza in un'ampia gamma di ambiti diversi, ogni *task* è assegnata a un **ruolo** specifico; si veda la Sezione 4.1.8 per un riassunto delle responsabilità e del valore economico del lavoro di ogni singolo ruolo.

Quando si deve pianificare lo svolgimento di una specifica *task*, il **responsabile**$_G$ crea un'*issue* specifica per un certo ruolo e determina i seguenti campi:

- *Start Date*: la data d'inizio prevista della *task*,
- *Target Date*: la data di fine prevista della *task*,
- *Expected Worked Hours*: il numero d'ore atteso per compiere la *task*,
- *Sprint Role*: il ruolo a cui corrisponde la *task*, può anche essere vuoto in caso l'attività sia di «palestra», ovvero sia una *task* di studio o di produzione di documentazione interna; questo campo è solitamente compilato automaticamente dalle automazioni usate dal gruppo
- *Sprint*: l'iterazione a cui è assegnata questa *task*; questo campo è solitamente compilato automaticamente dalle automazioni usate dal gruppo
- *Priority*: priorità dell'*issue*.

Inoltre, lo stato dell'*issue* rilevata dev'essere impostato a **«Ready»**, il quale indica che l'issue è pronta per essere svolta dall'assegnatario nell'iterazione corrente.

Durante la creazione dell'*issue*, il **responsabile**$_G$ può decidere se creare o meno un *branch* specifico per l'*issue* stessa. Se il **responsabile**$_G$ decide di creare un *branch* specifico, allora questo viene creato automaticamente tramite una **GitHub Action**$_G$ al momento dell'assegnazione dell'*issue* a un membro del gruppo: in questo modo l'assegnatario può lavorare sulla *task* in modo isolato, senza interferire con il lavoro degli altri membri del gruppo, e può facilmente tenere traccia del lavoro svolto per quella specifica *task*.

Le convenzioni di nomenclatura usate nella creazione automatica del *branch* sono descritte nella Sezione 4.2.3.5.6.

Il gruppo applica la procedura di tracciamento sopra descritta anche alle *task* di «palestra» citate in precedenza con lo scopo di tenere traccia del carico effettivo di ogni membro del gruppo, il quale non può essere rappresentato fedelmente dal numero di ore spese su attività solamente rendicontabili.

4.1.5. Esecuzione e controllo

L'esecuzione delle diverse attività è affidata ai diversi ruoli e dev'essere monitorata regolarmente dal **responsabile**$_G$ per accertarsi che ogni membro del gruppo rispetti i propri compiti nelle scadenze prefissate.

Una volta che le *issue* sono state create dal **responsabile**$_G$, la loro gestione viene affidata ai relativi assegnatari. Quando un membro del gruppo inizia a lavorare su una task, la relativa *issue* dev'essere impostata come **«In progress»**.

4.1.6. Revisione e valutazione

Una volta che l'attività è stata svolta, è fondamentale che questa venga verificata da un altro membro del gruppo che copre il ruolo di **verificatore**$_G$.

Il flusso di lavoro per le attività di verifica di documentazione è descritto nella Sezione 3.1.5.2.

Il processo di verifica di un'*issue* segue lo stesso **workflow** sia per *issue* che apportano modifiche che si riscontrano immediatamente nella **repository**$_G$ pubblica (ossia il sito web), sia per *issue* che non apportano modifiche immediate (come nel caso di **documenti incrementali**$_G$).

Il processo di verifica avviene nel seguente modo:
- L'assegnatario dell'*issue* notifica al **verificatore**$_G$ che l'attività è stata completata e che è pronta per essere verificata, aprendo di conseguenza la relativa **pull request**$_G$
- Nella **pull request**$_G$ devono essere indicati:
  - il *reviewer* e l'*assignee*, assegnando ad entrambi il **verificatore**$_G$ individuato per la *issue*;
  - la *milestone* corrispondente all'iterazione in cui è stata svolta la *task*;
- La relativa *issue* deve essere segnata come **«In review»** sul *project*.

Quando viene creata la **pull request**$_G$, un'*automation* di **GitHub Actions**$_G$ assegna automaticamente gli stessi campi previsti per le issue (vedi Sezione 4.1.4). In seguito è compito del **responsabile**$_G$ compilare tali campi con le stesse modalità usate durante la creazione delle issue.

L'unica eccezione riguarda lo *sprint role*: essendo la **pull request**$_G$ uno strumento di **verifica**, l'*automation* assegna automaticamente a questo campo il ruolo di **verificatore**$_G$.

Inoltre l'automation assegna automaticamente alla **pull request**$_G$ aperta la rispettiva *issue* nel campo **development**, migliorando la tracciabilità tra *issue* e relative *task* di verifica.

4.1.7. Conclusione

Per utilizzare una linea di lavoro comune, il gruppo applica una *Definition of Done*, ovvero una definizione di cosa determina se un'attività sia conclusa o meno, ben precisa: *Un attività è conclusa quando è stata approvata definitivamente da un verificatore diverso dall'assegnatario originale*.

Questa definizione garantisce che il gruppo possa determinare in ogni momento quando una task rendicontabile qualunque è conclusa o meno. Questa definizione non è sempre applicabile a tutte le *task* di «palestra», poiché non sempre essere richiedono una verifica da parte di un terzo, ma ciononostante vengono inserite nel **backlog**$_G$ di *task* del gruppo per motivi di tracciamento.

Quando una *task* viene conclusa, il suo assegnatario deve modificare la relativa *issue* impostando i seguenti campi:
- *End Date*: data effettiva di fine della *task*;
- *Worked Hours*: le ore effettive di lavoro sulla *task*;

Il **verificatore**$_G$ che ha approvato la **pull request**$_G$ relativa alla *task* deve modificarne i seguenti campi:
- *End Date*: data effettiva di fine della *task* di verifica;
- *Worked Hours*: le ore effettive di lavoro sulla *task* di verifica;

L'approvazione definitiva della **pull request**$_G$ comporta la chiusura automatica dell'*issue* associata: un'automation, infatti, imposterà l'issue associata come **«Done»**, facilitando il processo di tracciamento delle attività svolte e riducendo il rischio di dimenticanze da parte dei membri del gruppo.

La conclusione di una *task* comporta inoltre l'*eliminazione* del *branch* relativo alla *task* stessa, se questo è stato creato: tutto ciò avviene tramite un'automazione di **GitHub Actions**$_G$ che, a partire dal numero relativo all'*issue* legata alla **pull request**$_G$, identifica il *branch* da eliminare e lo elimina.

4.1.8. Ruoli

Di seguito, sono riportate le descrizioni dei compiti, delle responsabilità e del valore del lavoro di tutti i ruoli che i membri del gruppo possono assumere durante l'esecuzione del progetto.

Si noti che durante una certa iterazione, chiunque può assumere più ruoli a patto che ciò non causi conflitti d'interesse: ad esempio, un **programmatore**$_G$ non può anche essere **verificatore**$_G$ del suo stesso codice.

4.1.8.1. Amministratore

- **Presenza**: La figura dell'amministratore è presente a ogni iterazione di progetto
- **Compiti/responsabilità**:
  - Gestisce e apporta migliorie all'infrastruttura usata dal gruppo per compiere le attività di progetto, quali l'Issue Tracking System e gli strumenti di verifica;
  - Risolve le problematiche legate all'infrastruttura *as soon as possible*;
  - Con la sua conoscenza approfondita del **way of working**$_G$, redige il documento di **Piano di Qualifica**$_G$.
- **Valore delle ore produttive**: Ogni ora rendicontabile dell'amministratore ha il valore di **20 €/h**

4.1.8.2. Analista

- **Presenza**: La figura dell'**analista**$_G$ è presente principalmente nella prima fase di progetto, tra la **candidatura**$_G$ e l'**RTB**$_G$, in cui si delinea la maggior parte dei requisiti del software. La sua presenza è ampiamente ridotta nella fase compresa tra l'**RTB**$_G$ e il **PB**$_G$, in cui i requisiti delineati sono stabili e soggetti a cambiamenti minori.
- **Compiti/responsabilità**:
  - Identifica gli scenari d'uso del prodotto, tramite sessioni di *brainstorming* e le riunioni esterne con la **proponente**$_G$
  - Identifica i requisiti del Software e li classifica per tipologia e urgenza;
  - Redige il documento di **Analisi dei Requisiti**$_G$.
- **Valore delle ore produttive**: Ogni ora rendicontabile dell'analista ha il valore di **25 €/h**

4.1.8.3. Progettista

- **Presenza**: La figura del progettista è presente nella fase tra l'**RTB**$_G$ e il **PB**$_G$, in cui si delinea il *design* architetturale dell'MVP.
- **Compiti/responsabilità**:
  - Trasforma i requisiti, rilevati dall'**Analista**$_G$, in *design* architetturale;
  - Produce documenti e schemi che spiegano il *design* rilevato;
  - Definisce le scelte tecnologiche.
- **Valore delle ore produttive**: Ogni ora rendicontabile del progettista ha il valore di **25 €/h**

4.1.8.4. Programmatore

- **Presenza**: La figura del **programmatore**$_G$ è presente nella fase di sviluppo del **PoC**$_G$, poco prima della scadenza **RTB**$_G$ e per tutta la fase successiva fino alla produzione dell'**MVP**$_G$.
- **Compiti/responsabilità**:
  - È responsabile dello sviluppo del software, portando il *design* architetturale rilevato dai **progettisti**$_G$ in codice funzionante;
  - Collabora con i **progettisti**$_G$ per assicurarsi che il codice prodotto sia conforme al *design* rilevato;
  - Lavora a stretto contatto con i **verificatori**$_G$ per la produzione e l'esecuzione di **test**$_G$ che verificano il codice prodotto
- **Valore delle ore produttive**: Ogni ora rendicontabile del programmatore ha il valore di **15 €/h**

4.1.8.5. Responsabile

- **Presenza**: La figura del responsabile è presente ad ogni iterazione di progetto
- **Compiti/responsabilità**:
  - Coordina il *team*, assicurandosi che tutti i membri rispettino il *Way of Working* condiviso e le scadenze prefissate
  - All'inizio di ogni iterazione, comprende quali sono le attività che devono essere svolte e le pianifica, assegnando a ogni membro le adeguate *task*;
  - All'inizio di ogni iterazione, compila la sezione del **Piano di Progetto**$_G$ relativa alla previsione delle risorse che verranno consumate e dei rischi che potrebbero occorrere durante lo *sprint*, mentre a fine dello *sprint* ne compila la sezione relativa al consuntivo di periodo;
  - Fornisce l'approvazione finale delle aggiunte alla **repository**$_G$ pubblica;
  - Rappresenta il progetto rispetto agli stakeholders e si occupa della comunicazione con questi ultimi;
  - Redige i verbali interni ed esterni, se presente alle relative riunioni.
- **Valore delle ore produttive**: Ogni ora rendicontabile del responsabile ha il valore di **30€/h**

4.1.8.6. Verificatore

- **Presenza**: La figura del **verificatore**$_G$ è presente per l'intera durata del progetto
- **Compiti/responsabilità**:
  - Garantisce che ogni attività di progetto sia eseguita secondo lo stato dell'arte;
  - Esegue test e revisioni del software approfonditi;
  - Verifica la correttezza di ogni versione della documentazione prodotta;
  - Identifica aree di miglioramento in ciò che verifica, codice o documentazione che sia.
- **Valore delle ore produttive**: Ogni ora rendicontabile del verificatore ha il valore di **15 €/h**

4.1.9. Coordinamento

Il buon svolgimento del progetto è sancito dalla capacità dei membri del gruppo di comunicare e coordinarsi in maniera efficace tra loro e con la proponente.

Ciò avviene grazie a una buona organizzazione delle riunioni tra membri del gruppo e con la proponente grazie e a una scelta di strumenti appropriati per la comunicazione.

4.1.9.1. Riunioni

Le riunioni sono lo strumento principale con il quale è possibile comunicare **internamente** al gruppo o **esternamente** con la proponente e prendere delle decisioni tracciabili. Le riunioni sono fondamentali per permettere al gruppo e/o alla proponente di allinearsi sullo stato di avanzamento dei lavori e di prendere decisioni che vengono sancite con il verbale, ovvero il documento riassuntivo dell'incontro. Per maggiori informazioni sulle caratteristiche dei verbali, si può consultare la Sezione 3.1.3.2.

Il gruppo ha deciso di svolgere una **riunione interna** all'inizio di ogni **sprint**$_G$ con lo scopo di svolgere regolarmente la **retrospettiva**$_G$ sullo sprint passato, la pianificazione delle attività dello sprint venturo ed eventualmente ruotare i ruoli. Nello specifico, la retrospettiva è lo strumento con cui il gruppo ragiona sul *way of working* applicato nello sprint passato in modo tale da poterlo migliorare negli sprint successivi: i risultati della retrospettiva poi si riflettono necessariamente sulla pianificazione futura e sulle decisioni prese sul gruppo.

Le **riunioni esterne**, invece, sono lo strumento con cui il gruppo comunica direttamente con la proponente **M31 Srl**. Insieme all'azienda è stato deciso che ogni settimana avvenga un incontro

in cui questa svolge il ruolo di mentore, ovvero si rende disponibile per chiarimenti sui requisiti o consigli sulle tecnologie e soluzioni da adottare per soddisfarli. Mentre ogni due settimane, si tiene un incontro in cui l'azienda svolge il ruolo di cliente, ovvero osserva e monitora lo stato di avanzamento dei lavori e chiede di vedere dei risultati concreti.

4.1.9.2. Comunicazioni

Per quanto riguarda le comunicazioni interne, *GlitchHub Team* utilizza principalmente **WhatsApp** e **Discord** per, rispettivamente, le comunicazioni asincrone e veloci e le comunicazioni sincrone e strutturate. Infatti, tutte le comunicazioni di servizio che devono essere recapitate in maniera immediata a tutti i membri del gruppo avvengono tramite WhatsApp, mentre le conversazioni più strutturate e le riunioni avvengono tramite il gruppo Discord del *team*.

Eventuali comunicazioni dirette tra membri specifici che non richiedono l'attenzione del resto del gruppo avvengono tramite i messaggi diretti di Whatsapp.

Le comunicazioni esterne con la proponente vengono svolte dal **Responsabile**$_G$ via mail usando l'indirizzo di posta elettronica del gruppo `glitchhubteam@gmail.com`.

4.2. Infrastruttura

Il processo di **infrastruttura** ha l'obiettivo di stabilire e mantenere gli strumenti di supporto a tutti gli altri processi, sia *hardware* che *software*.

4.2.1. Attività previste

Il processo d'infrastruttura è composto dalle seguenti attività:

- Implementazione dell'infrastruttura
- Creazione dell'infrastruttura
- Manutenzione dell'infrastruttura

4.2.2. Implementazione

Durante lo svolgimento del progetto, il gruppo ha appreso i seguenti strumenti che hanno consentito un'organizzazione più efficace delle attività e dei processi di progetto:

- **Apidog**: È un servizio web che facilita la condivisione e la consultazione delle API stabilite per il progetto, consentendo di documentare in maniera chiara e accessibile tutte le API che il prodotto software espone.
- **ClickUp**: È un servizio web che consente una gestione totale dell'organizzazione delle *task* di gruppo. Inizialmente, *GlitchHub Team* ha provato ad utilizzarlo per pianificare le *task* degli *sprint*, ma lo strumento si è rivelato troppo macchinoso e inefficiente da usare, per cui è utilizzato dal gruppo principalmente per la condivisione degli appunti delle riunioni e della pianificazione degli eventi, quali i meeting esterni ed interni.
- **Discord**: È un servizio di messaggistica istantanea e videoconferenza usato da *GlitchHub Team* per svolgere le riunioni interne in maniera virtuale.
- **Git**: È uno dei *Version Control System* (VCS) open-source più usati al mondo: offre un sistema di versionamento efficace del codice e permette di separare lo spazio di lavoro degli utenti in *branch* separate. *GlitchHub Team* ha deciso di utilizzare questo strumento per versionare la propria documentazione e il proprio codice.
- **GitHub**: È un servizio che permette di eseguire l'*hosting* di **repository**$_G$ Git gratuitamente, fornendo molti strumenti che facilitano la collaborazione tra sviluppatori, quali:
  - **GitHub Issues** per il tracciamento del *backlog* delle *task*

- **GitHub Actions** per la gestione delle automazioni e delle pipeline **CI/CD**$_G$
- **GitHub Projects** per una gestione più fine della pianificazione delle *task*, tramite diagrammi di **Gantt**$_G$ e ***Kanban***$_G$ *boards*

GlitchHub Team ha deciso di utilizzare questo strumento per l'*hosting* pubblico delle proprie **repository**$_G$ di documentazione e di codice per il progetto.

- **Google Calendar**: È un servizio di calendario online che consente un'integrazione nativa con molti strumenti di pianificazione delle *task*. Il gruppo ha deciso di usarlo per la pianificazione degli incontri interni ed esterni.
- **Google Mail** (Gmail): È il servizio di posta elettronica dell'ecosistema Google. Viene usato da *GlitchHub Team* per la comunicazione con gli *stakeholders* del progetto.
- **Google Spreadsheets** (Google Fogli): È il foglio di calcolo elettronico online integrato nell'ecosistema Google. Viene usato da *GlitchHub Team* per la compilazione del cruscotto di valutazione.
- **Microsoft Teams**: È il servizio di videoconferenza di Microsoft. Viene utilizzato dal gruppo per le riunioni esterne con la proponente **M31 Srl**.
- **Script in Python e Go**: Vengono utilizzati dal gruppo per l'automazione di *task* ripetitive, quali la compilazione automatica dei file **Typst**$_G$, la compilazione dei campi delle *issue* o il conteggio delle ore automatico.
- **Typst**: È un sistema di composizione tipografica digitale moderno che consente di scrivere documentazione in maniera rapida, consentendo un alto grado di personalizzazione dei documenti. Il gruppo utilizza Typst per la redazione di tutta la documentazione, usando degli script per automatizzare la loro compilazione in PDF.
- **WhatsApp**: È un servizio di messaggistica istantanea.

4.2.3. Creazione

La creazione dell'infrastruttura è l'attività iniziale di impostazione degli strumenti infrastrutturali usati dal gruppo. Di seguito sono riportati i dettagli per ogni strumento utilizzato.

4.2.3.1. Apidog

Per l'utilizzo di **Apidog**, il gruppo ha creato un ambiente condiviso, nel quale sono stati creati 3 progetti:

- **Gin Backend**: in cui sono documentate tutte le API del server **HTTP** scritto in **Gin**$_G$. Sono utili per mantenere coerenza tra le chiamate API dal frontend e gli endpoint effettivamente esposti dal backend.
- **NATS**: in cui sono documentati tutti i **subject**$_G$ usati per la comunicazione **Fire-and-Forget**$_G$ attraverso **NATS**$_G$, così da mantenere coerenza tra i messaggi inviati e quelli effettivamente ricevuti. È importante distinguere l'utilizzo di NATS o NATS JetStream per dare indicare il client da utilizzare.
- **NATS JetStream**: in cui sono documentati tutti i **subject**$_G$ usati per la comunicazione via **NATS JetStream**$_G$.

La definizione degli **endpoint**$_G$ avviene tramite 3 step:

1. Scelta del nome dell'endpoint e il protocollo utilizzato (NATS, HTTP, etc.);
2. Definizione schema per la **richiesta** dell'endpoint in questione, comprendente di nome, tipo, dominio e descrizione dei campi;

3. Definizione schema per la **risposta** dell'endpoint in questione (se necessaria), comprendente di nome, tipo, dominio e descrizione dei campi.

4.2.3.2. ClickUp

Per usare **ClickUp**, il gruppo ha creato un ambiente condiviso, nel quale è possibile condividere documenti testuali e pianificare, assegnare e gestire attività.

4.2.3.3. Discord

Il gruppo ha creato un server **Discord** privato per le comunicazioni e le riunioni interne. Non sono state necessarie altre configurazioni particolari.

4.2.3.4. Git

Essendo **Git** uno strumento usato localmente dai membri del gruppo, non sono richiesti particolari passaggi di configurazione per farlo funzionare.

Gli unici passaggi che ogni membro del gruppo deve compiere quando clona le **repository**$_G$ del progetto sono l'impostazione di nome utente e email che devono coincidere con il nome utente e l'email usate per accedere a **GitHub**$_G$.

4.2.3.5. GitHub

Di seguito sono riportati tutti gli strumenti dell'ecosistema **GitHub** utilizzati da *GlitchHub Team*

4.2.3.5.1. GitHub Organization

Prima di iniziare le attività di progetto, il gruppo ha istituito la **GitHub Organization**$_G$ di *GlitchHub Team*, in cui sono raccolte tutte le **repository**$_G$ e tutti il **GitHub Project**$_G$ utilizzati dal gruppo.

La pagina principale dell'*organization* introduce brevemente il gruppo e i suoi membri.

4.2.3.5.2. Repository documentazione (`GlitchHub-Team/GlitchHub-Team.github.io`)

La **repository**$_G$ di documentazione del gruppo è stata istituita con il nome `GlitchHub-Team.github.io`, poiché ciò consente la creazione di una **GitHub Page**$_G$, ovvero una pagina web statica utilizzabile gratuitamente che il gruppo usa per esporre al pubblico le versioni stabili dei prodotti di progetto.

All'interno della repository si trovano il file **README.md** che la descrive dettagliatamente, il file **.gitignore** che permette di impedire il versionamento per certi file specifici e le seguenti cartelle:

- `.github/`, in cui sono contenuti i file relativi ai workflow di **GitHub Actions**$_G$ e gli **issue templates**$_G$ descritti;
- `script/` in cui sono presenti i sorgenti dei vari script utilizzati dal gruppo per la compilazione dei file Typst e di altre automazioni utilizzate dal gruppo durante la redazione dei documenti;
- `src/` in cui sono presenti i sorgenti dei file di documentazione;
- `website/` in cui è presente il sorgente della **GitHub Page**$_G$.

Inoltre, la cartella `src/` contiene al suo interno:

- Le cartelle `Candidatura/` e `RTB/` sono contenuti i sorgenti dei documenti appartenenti rispettivamente alla fase di **candidatura**$_G$ e alla fase **RTB**$_G$
- La cartella `Slide/`, che contiene i sorgenti delle presentazioni dei diari di bordo:
- La cartella `Templates/`, che contiene i template di Typst utilizzati dal gruppo
- La cartella `assets/`, che contiene tutti gli asset usati nei documenti, tra cui immagini, diagrammi e font personalizzati;
- La cartella `lib/`, che contiene le librerie usate dai file Typst;

- La cartella `tracciamento/`, che contiene tutti i file JSON di tracciamento dei requisiti descritti nella Sezione 3.5.2;
- Il file `glossary.json`, che contiene tutti i termini del glossario usati nei documenti, con le relative definizioni e link alle sezioni in cui sono descritti.

4.2.3.5.3. Repository PoC (`GlitchHub-Team/PoC`)

La **repository**$_G$ che contiene il codice sorgente del **PoC**$_G$ contiene al suo interno:

- La cartella `.github/`, come nella Sezione 4.2.3.5.2
- La cartella `.vscode/`, che contiene elementi di configurazione comuni per l'editor di testo Visual Studio Code;
- La cartella `evaluation/`, che contiene dei documenti che riportano la valutazione delle tecnologie scelte;
- La cartella `src/` che contiene il codice sorgente del PoC;
- Il file `README.md` che descrive in modo dettagliato il PoC e le istruzioni per avviarlo;
- Il file `docker-compose.yml` che contiene la configurazione di **Docker Compose** per l'esecuzione del PoC su qualunque computer.

4.2.3.5.4. Repository GitHub Actions (`GlitchHub-Team/actions`)

Questa **repository**$_G$ viene usata dal gruppo per raggruppare tutte le **GitHub Actions**$_G$ comuni a tutte le altre repository.

Si notano principalmente le seguenti *action*:

- `.github/workflows/issue-action.yml` che consente di assegnare automaticamente qualunque **Github Issue**$_G$ e **pull request**$_G$ al **GitHub Project**$_G$ del gruppo, compilando in modo automatico i campi *Sprint* e *Sprint Role* dell'*issue*;
- `.github/workflows/branch-action.yml` che consente di creare un *branch* specifico per ogni **GitHub Issue**$_G$ creata, con il nome che segue la convenzione descritta nella Sezione 4.2.3.5.6;
- `.github/workflows/delete-issue-branches.yml` che consente di eliminare il *branch* relativo a un'*issue* quando questa viene chiusa attraverso l'approvazione della relativa **pull request**$_G$.

In generale per utilizzare queste *action* in una **repository**$_G$ specifica, è necessario inserire sul branch `main` di quest'ultima un file nella cartella `.github/workflows/` che descriva il flusso dell'action.

4.2.3.5.5. GitHub Issues

Il gruppo ha deciso di utilizzare le **GitHub Issues**$_G$ per il tracciamento delle attività su tutte le **repository**$_G$, disponendo una serie di **issue templates**$_G$ per facilitarne la creazione.

Il nome di un'*issue* segue la seguente convenzione:

`[scope] attività`

Dove:

- **`scope`** rappresenta ciò che l'issue modifica:
  - Se l'issue modifica un documento, allora sarà l'abbreviazione del suo nome tra parentesi quadre. I possibili `scope` sono:
    - «`NdP`», ossia **Norme di Progetto**$_G$
    - «`AdR`», ossia **Analisi dei Requisiti**$_G$
    - «`PdP`», ossia **Piano di Progetto**$_G$
    - «`PdQ`», ossia **Piano di Qualifica**$_G$
    - «`ST`», ossia **Specifica Tecnica**$_G$
    - «`Gloss`» o «`Glossario`»

  - Se l'issue concerne il **PoC**$_G$, allora si userà la dicitura «`PoC`»
- **`attività`** rappresenta una breve descrizione delle modifiche apportate dall'*issue*.

Per maggiori informazioni riguardanti la gestione delle *issues* durante lo svolgimento della relativa *task*, si consiglia di consultare la Sezione 4.1.2.

4.2.3.5.6. GitHub branches

Il gruppo ha deciso di affiancare la creazione di un *branch* dedicato ad ogni *issue*, in modo da rendere più semplice isolare le modifiche effettuate da ogni singolo membro e tenere traccia del lavoro svolto.

Il nome di un *branch* dedicato a un'*issue* segue la seguente convenzione:

`issue-[issue-number]`

Dove:

- **`[issue-number]`** rappresenta il numero identificativo dell'*issue* a cui il *branch* è legato, che viene assegnato automaticamente da GitHub al momento della creazione dell'*issue*.

Per maggiori informazioni riguardanti la gestione dei *branch* durante lo svolgimento della relativa *issue*, si consiglia di consultare la Sezione 4.1.2.

4.2.3.5.7. GitHub Project

Il gruppo ha deciso di utilizzare un **GitHub Project**$_G$ per raggruppare tutte le *issue* delle diverse **repository**$_G$ in un luogo unico e per assegnare a ogni *issue* dei campi aggiuntivi che descrivono l'organizzazione delle relative *task*. La descrizione dei campi aggiuntivi forniti dal *project* è presente nella Sezione 4.1.2.

Inoltre, il *project* impostato consente di visualizzare anche le *issue* in diagrammi di **Gantt**$_G$ (separati per membro del gruppo o per sotto-attività) e di visualizzare anche una board **Kanban**$_G$ per una visualizzazione rapida di quali *task* sono in **backlog**$_G$, quali sono pronte per essere eseguite, su quali *issue* si sta lavorando al momento e su quali sono in revisione.

4.2.3.6. Google Calendar

Il gruppo ha creato un calendario condiviso in cui il **Responsabile**$_G$ può pianificare nuovi eventi o modificare quelli presenti permettendo a ogni membro dell gruppo di visualizzarli sui propri dispositivi e di ricevere le relative notifiche di promemoria.

4.2.3.7. Google Mail

Il gruppo ha creato la casella di posta all'indirizzo `glitchhubteam@gmail.com` per tutte le comunicazioni con gli *stakeholders* del progetto, ovvero i professori Tullio Vardanega e Riccardo Cardin e l'azienda proponente M31 Srl.

Nella casella di posta, sono state configurate le etichette automatiche per identificare immediatamente i tipi di messaggi ricevuti e per archiviare i messaggi superflui.

4.2.3.8. Google Spreadsheets

Il file condiviso di **Google Spreadsheets** è stato creato dal gruppo per contenere le informazioni relative al cruscotto di valutazione, poi inserito all'interno del **Piano di Qualifica**$_G$.

Il file non ha richiesto particolari configurazioni.

4.2.3.9. Microsoft Teams

La piattaforma **Microsoft Teams** viene usata dalla proponente, per cui il gruppo non ne ha alcun controllo.

4.2.3.10. Script in Python e Go

Gli script utilizzati dal gruppo sono i seguenti nella repository di documentazione:

- Il file `compile.sh` permette di compilare tutti i file sorgenti Typst in file PDF;
- Il file `script/renderHTMLwithPDFs.go` permette di modificare la **GitHub Page**$_G$ in modo tale da rendere accessibili pubblicamente i documenti compilati in PDF;
- La cartella `src/RTB/DocumentiEsterni/sprintPdPGenerator` contiene lo script necessario per generare automaticamente le tabelle di consuntivo di periodo e di preventivo a finire di un determinato sprint, tramite il calcolo delle ore di lavoro attese ed effettive delle *issues* collegate al **GitHub Project**$_G$ del gruppo.
- La cartella `script/scriptPdQ` contiene lo script per la rilevazione dei dati utili al calcolo delle metriche presenti nel **Piano di Qualifiche**$_G$

4.2.3.11. Typst

Prima di iniziare a redigere la documentazione, il gruppo ha stabilito un ambiente di Typst nella repository di documentazione per la creazione di documenti eleganti ed omogenei, creando degli appositi *templates* e funzioni personalizzate.

4.2.3.11.1. Template

I *template* utilizzati dal gruppo risiedono nella cartella `src/Templates` e forniscono gli strumenti per usare uno stile standard nella redazione dei documenti.

Per maggiori informazioni riguardanti le caratteristiche dei documenti e dei relativi *template* è possibile consultare la Sezione 3.1.3.

4.2.3.11.2. Funzioni personalizzate

Nei template e nella cartella `src/lib` sono messe a disposizione le seguenti funzioni *helper* che facilitano la redazione dei documenti:

- `issue()` e `issue_full()` consentono di creare rapidamente un link a una **issue**$_G$ di qualunque **repository**$_G$ del gruppo;
- `repo()` consente di creare rapidamente un link a una delle **repository**$_G$ del gruppo;
- `gloss()` consente di visualizzare una parola con la convenzione delle parole di glossario, ovvero in grassetto e con una «G» a pedice.

4.2.3.12. WhatsApp

Il gruppo ha creato un gruppo WhatsApp privato per le comunicazioni interne. Non sono state necessarie configurazioni particolari.

4.2.4. Manutenzione

L'attività di manutenzione del'infrastruttura è fondamentale per assicurare che il *way of working* del gruppo rimanga allo stato dell'arte. Si affida, pertanto, alla figura dell'**amministratore**$_G$ la responsabilità di controllare frequente il regolare funzionamento dell'infrastruttura e di risolvere il prima possibile eventuali problemi che possono insorgere nel suo utilizzo da parte del gruppo.

4.3. Miglioramento

Il processo di **miglioramento** è definito dallo standard *ISO/IEC 12207:1995* come l'insieme di attività atte alla creazione, valutazione, misurazione, controllo e miglioramento dei processi di cicli di vita del software.

I processi di miglioramento continuo del *way of working* sono delle attività fondamentali per consentire al gruppo di aumentare la propria efficacia ed efficienza del proprio lavoro a ogni iterazione.

4.3.1. Attività previste

Le attività previste da questo processo sono le seguenti

- Inizializzazione dei processi
- Valutazione dei processi
- Miglioramento dei processi

4.3.2. Inizializzazione dei processi

Per poter attuare il processo di miglioramento, è prima necessario stabilire e documentare tutti i processi organizzativi per ogni processo di ciclo di vita del software: questo documento ha questo preciso scopo. Inoltre, è fondamentale stabilire dei meccanismi di controllo dei processi per permetterne il miglioramento.

4.3.3. Valutazione dei processi

Quando i processi sono stati stabiliti e documentati è necessario anche stabilire un processo di valutazione degli stessi, il quale dev'essere anch'esso documentato e applicato. Per poter applicare il processo di valutazione è necessario revisionare regolarmente i processi applicati in modo tale da garantire la loro idoneità al progetto ed efficacia.

GlitchHub Team ha redatto il documento di **Piano di Qualifica**$_G$ con lo scopo di misurare la qualità dei processi applicati a ogni *sprint* in maniera quantificabile e misurabile, in modo tale da poter trarre delle conclusioni su quali siano i difetti del *way of working* del gruppo e come questi si possano migliorare in modo continuo.

4.3.4. Miglioramento dei processi

Le misurazioni e i controlli compiuti nella Sezione 4.3.3 hanno lo scopo di rilevare quali siano i processi che presentano problematiche; una volta rilevati, è necessario individuare delle soluzioni che possano migliorarne l'efficacia ed efficienza, aggiornando anche la relativa documentazione.

4.4. Processo di formazione

Il processo di **formazione** ha lo scopo di far sì che i membri del gruppo sviluppino le abilità necessarie per affrontare il progetto e che le affinino costantemente, in modo tale da produrre i prodotti di progetto in maniera sempre più efficiente.

4.4.1. Attività previste

Le attività previste da questo processo sono:

- Implementazione del processo
- Sviluppo del materiale di formazione
- Implementazione del piano di formazione

4.4.2. Implementazione del processo

Secondo lo standard *ISO/IEC 12207:1995*, per poter attuare il processo di formazione è fondamentale condurre una revisione dei requisiti del progetto, in modo tale da acquisire e sviluppare le risorse e abilità necessarie per il suo buon svolgimento.

Dopo un'attenta analisi dei requisiti del progetto, il gruppo ha stabilito che è necessario studiare e approfondire le seguenti tecnologie:

- Per lo sviluppo del codice del progetto:
  - Il linguaggio di programmazione **Go**$_G$ e il framework **Gin**$_G$
  - Il linguaggio **Typescript**$_G$ insieme al framework **Angular.js**$_G$ e alla libreria Chart.js
  - Il sistema di virtualizzazione **Docker**$_G$
  - Il sistema di *messaging* **NATS JetStream**$_G$
  - I sistemi di *observability* **Grafana**$_G$ e **Prometheus**$_G$
- Per la scrittura e la compilazione della documentazione:
  - Il sistema di composizione tipografica digitale **Typst**
  - Il linguaggio **Go** per gli script di compilazione
- Per la gestione del versionamento del codice:
  - Il *Version Control System* (VCS) **Git**$_G$
  - La piattaforma di *hosting* di repository **GitHub**$_G$
  - Il linguaggio **Python**$_G$ per la creazione di **GitHub Actions**$_G$ utili

4.4.3. Sviluppo del materiale di formazione

Di seguito sono riportate le risorse utilizzate dal gruppo per imparare le tecnologie identificate.

4.4.3.1. Angular e Typescript

- Documentazione ufficiale di *Typescript*
- Documentazione ufficiale di *Chart.js*
- Documentazione ufficiale di *Angular.js*
- *Crash course* su *Angular.js* – Non ufficiale

4.4.3.2. Docker

- *Get started* ufficiale di *Docker*
- Documentazione ufficiale di *Docker Compose*

4.4.3.3. Git e GitHub

Risorse per l'approfondimento di **Git**:

- Tutorial ufficiale di *Git*
- Manuale di riferimento ufficiale di *Git*

Risorse per l'approfondimento di **GitHub**:

- Documentazione ufficiale di *GitHub*
- *Get started* ufficiale di *GitHub*
- Documentazione ufficiale dell'API di *GitHub*
  - Questa risorsa si rivela particolarmente utile per lo sviluppo delle **GitHub Actions**$_G$

4.4.3.4. Go e Gin

Risorse per l'approfondimento di **Go**:

- Documentazione ufficiale di *Go*
- *Tour of Go*, tutorial interattivo di *Go*

Risorse per l'approfondimento di **Gin**:
- Documentazione ufficiale di *Gin*
  - Si noti che è questa documentazione è carente sotto alcuni punti vista, per cui si consiglia l'utilizzo di articoli non ufficiali per una maggiore comprensione *hands-on* di Gin
- *Building user authentication and authorization API in Go using Gin and Gorm – Verdotte Aututu* (non ufficiale)

4.4.3.5. Grafana e Prometheus
- Documentazione ufficiale di *Prometheus*
- *Creating Grafana Dashboards for Prometheus: A Beginner's Guide – Ayooluwa Isaiah* (non ufficiale)
- *Introducing Prometheus with Grafana: Metrics Collection and Monitoring – Arindam Paul* (non ufficiale)
- *Monitoring with Prometheus and Grafana: A Comprehensive Guide* (non ufficiale)

4.4.3.6. NATS
- Documentazione ufficiale di *NATS*
- Documentazione ufficiale di *NATS JetStream*

4.4.3.7. Python
Essendo Python un linguaggio molto semplice, è possibile utilizzare una *cheatsheet* come https://learnxinyminutes.com/python/ per approfondirlo in maniera rapida.

4.4.4. Implementazione del piano di formazione
Ogni membro di *GlitchHub Team* si impegna a ricavare in ogni *sprint* le risorse temporali necessarie per approfondire le tecnologie usate, in modo tale da permettere una formazione omogenea e comprensiva di tutte le tecnologie utilizzate. La formazione personale di ogni membro avverrà principalmente in maniera asincrona, ma sfruttando anche le sessioni di *brainstorming* come strumenti di formazione collettiva.

Si noti anche che lo sviluppo del **Proof of Concept**$_G$ e dell'**MVP**$_G$ costituiranno i momenti di maggiore formazione e pratica nell'uso delle tecnologie sopra riportate.

5. Metriche di qualità
La definizione operativa delle metriche di qualità all'interno delle Norme di Progetto ha lo scopo di normare le attività di misurazione, stabilendo per ogni metrica **cosa** misurare, **come** calcolarne il valore.

Gli **obiettivi metrici** (soglie accettabili e ottime) e i **valori rilevati** per ogni sprint sono documentati nel **Piano di Qualifica**$_G$, il quale contiene il cruscotto di valutazione con l'andamento storico delle misurazioni.

Ogni metrica è descritta in termini di:
- **Identificativo**: codice univoco della metrica
- **Nome**: denominazione della metrica
- **Descrizione**: definizione operativa dello scopo della metrica
- **Formula**: espressione matematica per il calcolo del valore
- **Unità di misura**: unità in cui il valore è espresso

Le metriche di processo sono identificate dalla sigla **MPC** (**M**etrica di **P**rocesso e **C**ontrollo), mentre le metriche di prodotto dalla sigla **MPD** (**M**etrica di **P**ro**d**otto).

5.1. Metriche di qualità del processo

5.1.1. Fornitura

5.1.1.1. MPC-PV: Planned Value

- **Descrizione**: Valore economico del lavoro che si era pianificato di completare entro la fine dello sprint. Rappresenta il costo previsto per le attività pianificate.
- **Formula**:

$$\mathrm{PV} = \sum_{i=1}^{n} (\text{Ore Previste}_i \times \text{Tariffa Oraria}_i)$$

  dove n è il numero di task pianificate nello sprint e la tariffa oraria corrisponde al ruolo assegnato alla task.
- **Unità di misura**: Euro (€)

5.1.1.2. MPC-AC: Actual Cost

- **Descrizione**: Costo effettivamente sostenuto per le ore lavorate nello sprint.
- **Formula**:

$$\mathrm{AC} = \sum_{i=1}^{n} (\text{Ore Effettive}_i \times \text{Tariffa Oraria}_i)$$

  dove n è il numero di task completate nello sprint e la tariffa oraria corrisponde al ruolo assegnato alla task.
- **Unità di misura**: Euro (€)

5.1.1.3. MPC-EV: Earned Value

- **Descrizione**: Valore del lavoro effettivamente completato, misurato come la quota di lavoro pianificato che è stata portata a termine.
- **Formula**:

$$\mathrm{EV} = \mathrm{PV} \times \left(\frac{\text{Ore Completate}}{\text{Ore Pianificate}} \right)$$

- **Unità di misura**: Euro (€)

5.1.1.4. MPC-BAC: Budget At Completion

- **Descrizione**: Budget totale preventivato per l'intero progetto. Questo valore è fissato alla stipula del contratto e può solo diminuire nel tempo.
- **Formula**:

$$\mathrm{BAC} = 12.825€$$

  Nota: Il valore considerato fino allo **sprint 9** è di 12.975 € (vd. Piano di Progetto, sezione 4.1)
- **Unità di misura**: Euro (€)

5.1.1.5. MPC-EAC: Estimated At Completion

- **Descrizione**: Stima del costo totale finale a completamento del progetto, basata sull'efficienza dei costi misurata dal CPI (Cost Performance Index).
- **Formula**:

$$\mathrm{EAC} = \frac{\mathrm{BAC}}{\mathrm{CPI}}$$

dove:

$$\mathrm{CPI} = \frac{\mathrm{EV}}{\mathrm{AC}}$$

Il **CPI** (Cost Performance Index) rappresenta l'efficienza dei costi:
- $\mathrm{CPI} > 1$: il progetto sta producendo valore a un costo inferiore al previsto
- $\mathrm{CPI} = 1$: il progetto è in linea con il budget
- $\mathrm{CPI} < 1$: il progetto sta spendendo più del previsto

- **Interpretazione**:
  - $\mathrm{EAC} < \mathrm{BAC}$: il progetto finirà sotto budget
  - $\mathrm{EAC} = \mathrm{BAC}$: il progetto finirà in linea con il budget
  - $\mathrm{EAC} > \mathrm{BAC}$: il progetto finirà sopra budget
- **Unità di misura**: Euro (€)

5.1.1.6. MPC-ETC: Estimated To Complete

- **Descrizione**: Stima del costo rimanente per completare il progetto, ottenuta sottraendo il costo cumulativo effettivo dalla stima a completamento.
- **Formula**:

$$\mathrm{ETC} = \mathrm{EAC} - \mathrm{AC}_{\mathrm{cumulativo}}$$

dove $\mathrm{AC}_{\mathrm{cumulativo}}$ è la somma dei costi effettivi sostenuti dall'inizio del progetto fino allo sprint corrente.
- **Unità di misura**: Euro (€)

5.1.1.7. MPC-CV: Cost Variance

- **Descrizione**: Deviazione dal budget nello sprint corrente. Indica se il lavoro completato è costato più o meno di quanto valga.
- **Formula**:

$$\mathrm{CV} = \mathrm{EV} - \mathrm{AC}$$

- **Interpretazione**:
  - $\mathrm{CV} > 0$: sotto budget (situazione favorevole)
  - $\mathrm{CV} = 0$: in linea con il budget
  - $\mathrm{CV} < 0$: sopra budget (situazione sfavorevole)
- **Unità di misura**: Euro (€)

5.1.1.8. MPC-SV: Schedule Variance

- **Descrizione**: Deviazione dalla pianificazione nello sprint corrente. Indica se il progetto è in anticipo o in ritardo rispetto al piano.
- **Formula**:

$$\mathrm{SV} = \mathrm{EV} - \mathrm{PV}$$

- **Interpretazione**:
  - $\mathrm{SV} > 0$: in anticipo rispetto al piano
  - $\mathrm{SV} = 0$: in linea con il piano
  - $\mathrm{SV} < 0$: in ritardo rispetto al piano
- **Unità di misura**: Euro (€)

5.1.1.9. MPC-TCR: Task Completion Rate

- **Descrizione**: Percentuale di task completati entro la scadenza dello sprint rispetto al totale dei task chiusi nello sprint (in tempo e in ritardo). Misura la capacità del team di rispettare le scadenze prefissate.
- **Formula**:

$$\text{TCR} = \left(\frac{\text{Task Completati in Tempo}}{\text{Task Completati in Tempo} + \text{Task Completati in Ritardo}} \right) \times 100$$

  Un task è considerato «completato in tempo» se la sua *End Date* è precedente o uguale alla *Target Date* dello sprint.
- **Unità di misura**: percentuale (%)

5.1.1.10. MPC-TS: Task Slippage

- **Descrizione**: Percentuale di task pianificati per lo sprint corrente che non sono stati portati a termine entro la sua conclusione e vengono posticipati allo sprint successivo.
- **Formula**:

$$\text{TS} = \left(\frac{\text{Task Posticipati}}{\text{Task Totali dello Sprint}} \right) \times 100$$

  Un task è considerato «posticipato» se alla chiusura dello sprint il suo stato è diverso da «Done».
- **Unità di misura**: percentuale (%)

5.1.2. Sviluppo

5.1.2.1. MPC-PRCT: Pull Request Cycle Time

- **Descrizione**: Tempo medio che intercorre tra l'apertura di una **Pull Request**$_G$. Monitora l'efficienza del processo di revisione.
- **Formula**:

$$\text{PRCT} = \frac{\sum_{i=1}^{n} (\text{Timestamp Merge}_i - \text{Timestamp Apertura}_i)}{n}$$

  dove n è il numero di Pull Request integrate nello sprint.

5.1.3. Documentazione

5.1.3.1. MPC-IG: Indice di Gulpease

- **Descrizione**: Indice di leggibilità calibrato per la lingua italiana. Valuta la complessità del testo in base alla lunghezza delle parole e delle frasi. Valori più alti indicano maggiore leggibilità.
- **Formula**:

$$\text{IG} = 89 + \frac{300 \times \text{Numero Frasi} - 10 \times \text{Numero Lettere}}{\text{Numero Parole}}$$

- **Unità di misura**: adimensionale (scala 0–100)

5.1.3.2. MPC-CO: Correttezza Ortografica

- **Descrizione**: Numero di errori grammaticali o di battitura rilevati nei documenti tramite strumenti automatici di controllo ortografico.
- **Formula**:

$$\text{CO} = \text{Numero di Errori Ortografici Rilevati}$$

- **Unità di misura**: numero intero (conteggio assoluto)

5.1.4. Verifica

5.1.4.1. MPC-CC: Code Coverage

- **Descrizione**: Percentuale di righe di codice sorgente effettivamente eseguite durante i test automatici. Misura il grado di copertura complessivo della suite di test.
- **Formula**:

$$\mathrm{CC} = \left(\frac{\text{Linee di Codice Eseguite dai Test}}{\text{Linee di Codice Totali}} \right) \times 100$$

- **Unità di misura**: percentuale (%)
- **Strumento di rilevazione**: framework di test con supporto a code coverage (da specificare in fase di sviluppo **MVP**$_G$)

5.1.4.2. MPC-TSR: Test Success Rate

- **Descrizione**: Percentuale di test passati con successo sul totale dei test eseguiti nello sprint.
- **Formula**:

$$\mathrm{TSR} = \left(\frac{\text{Test Superati}}{\text{Test Eseguiti}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

5.1.4.3. MPC-DD: Bug Density

- **Descrizione**: Densità di bug nel codice sorgente, calcolata come numero di bug rilevati per ogni cento righe di codice (CRG). Valori bassi indicano maggiore qualità del codice prodotto.
- **Formula**:

$$\mathrm{DD} = \frac{\text{Numero di Bug Rilevati}}{\mathrm{CRG}}$$

  dove $\mathrm{CRG} = \frac{\text{Righe di Codice Totali}}{100}$.
- **Unità di misura**: bug per CRG (centinaia di righe di codice)

5.1.5. Gestione della qualità

5.1.5.1. MPC-QMS: Quality Metrics Satisfied

- **Descrizione**: Percentuale di metriche di qualità (di processo e di prodotto) che rientrano nel range accettabile definito nel **Piano di Qualifica**$_G$. Rappresenta un indicatore sintetico della salute complessiva del progetto.
- **Formula**:

$$\mathrm{QMS} = \left(\frac{\text{Metriche in Range Accettabile}}{\text{Totale Metriche Monitorate}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

5.1.5.2. MPC-TE: Time Efficiency

- **Descrizione**: Rapporto tra le ore dedicate ad attività produttive (rendicontabili) e le ore totali di lavoro (incluse le attività di «palestra» e studio). Indica quanta parte del tempo impiegato dal team si traduce direttamente in prodotti di progetto.
- **Formula**:

$$\mathrm{TE} = \left(\frac{\text{Ore Produttive Rendicontabili}}{\text{Ore Totali Lavorate}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

5.1.5.3. MPC-WD: Work Distribution

- **Descrizione**: Misura l'equilibrio nella distribuzione del carico di lavoro tra i membri del team. Viene calcolata come la deviazione standard delle ore lavorate dai singoli membri rispetto alla media del team nello sprint. Valori bassi indicano una distribuzione omogenea del carico.
- **Formula**:

$$\mathrm{WD} = \sigma \times 100$$

dove:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} \left(h_i^{\%} - \overline{h^{\%}} \right)^2}{n}}$$

  - n: numero di membri del team
    - Si noti che $n = 7$
  - $h_i^{\%}$: «carico di lavoro» del membro i nello sprint, definito come

$$\frac{h_i}{\sum_{j=1}^{n} h_j} = \frac{\text{ore di lavoro del membro } i}{\text{ore di lavoro complessive di tutti i membri del gruppo}}$$

  - $\overline{h^{\%}}$: media del carico di lavoro di tutti i membri, sempre pari a $\frac{1}{7} \approx 14,29\%$

- **Unità di misura**: percentuale (%)

5.2. Metriche di qualità del prodotto

5.2.1. Funzionalità

5.2.1.1. MPD-CRO: Copertura Requisiti Obbligatori

- **Descrizione**: Percentuale di requisiti obbligatori correttamente implementati e verificati rispetto al totale dei requisiti obbligatori rilevati nell'**Analisi dei Requisiti**$_G$.
- **Formula**:

$$\mathrm{CRO} = \left(\frac{\text{Requisiti Obbligatori Soddisfatti}}{\text{Requisiti Obbligatori Totali}} \right) \times 100$$

  Un requisito è considerato «soddisfatto» quando il relativo test di sistema risulta superato.
- **Unità di misura**: percentuale (%)

5.2.1.2. MPD-CRD: Copertura Requisiti Desiderabili

- **Descrizione**: Percentuale di requisiti desiderabili correttamente implementati e verificati rispetto al totale dei requisiti desiderabili rilevati.
- **Formula**:

$$\mathrm{CRD} = \left(\frac{\text{Requisiti Desiderabili Soddisfatti}}{\text{Requisiti Desiderabili Totali}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

5.2.1.3. MPD-CROP: Copertura Requisiti Opzionali

- **Descrizione**: Percentuale di requisiti opzionali correttamente implementati e verificati rispetto al totale dei requisiti opzionali rilevati.
- **Formula**:

$$\text{CROP} = \left(\frac{\text{Requisiti Opzionali Soddisfatti}}{\text{Requisiti Opzionali Totali}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

5.2.1.4. MPD-AD: API Documentation Coverage

- **Descrizione**: Percentuale di endpoint API pubblici che dispongono di documentazione completa e aggiornata. Un endpoint è considerato «documentato» quando sono presenti: descrizione funzionale, parametri di input con tipo e vincoli, formato della risposta con codici di stato HTTP e almeno un esempio d'uso.
- **Formula**:

$$\text{AD} = \left(\frac{\text{Endpoint API Documentati}}{\text{Endpoint API Totali}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

5.2.1.5. MPD-DL: Data Loss Rate

- **Descrizione**: Percentuale di messaggi persi durante la trasmissione dati tra gateway e infrastruttura cloud. Misura l'affidabilità della catena di comunicazione.
- **Formula**:

$$\text{DL} = \left(\frac{\text{Messaggi Inviati} - \text{Messaggi Ricevuti}}{\text{Messaggi Inviati}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

5.2.2. Affidabilità

5.2.2.1. MPD-BC: Branch Coverage

- **Descrizione**: Percentuale di rami decisionali (ad esempio i rami `true` e `false` di un costrutto `if`) che sono stati eseguiti durante i test. Garantisce che tutte le possibili direzioni del flusso logico siano state verificate.
- **Formula**:

$$\text{BC} = \left(\frac{\text{Rami Decisionali Eseguiti}}{\text{Rami Decisionali Totali}} \right) \times 100$$

- **Unità di misura**: percentuale (%)
- **Strumento di rilevazione**: framework di test con supporto a branch coverage

5.2.2.2. MPD-SC: Statement Coverage

- **Descrizione**: Percentuale di singole istruzioni (statement) percorse durante l'esecuzione dei test. Assicura che non vi siano porzioni di codice mai eseguite dalla suite di test.
- **Formula**:

$$\text{SC} = \left(\frac{\text{Istruzioni Eseguite}}{\text{Istruzioni Totali}} \right) \times 100$$

- **Unità di misura**: percentuale (%)
- **Strumento di rilevazione**: framework di test con supporto a statement coverage

5.2.3. Usabilità

5.2.3.1. MPD-TT: Time on Task

- **Descrizione**: Tempo medio necessario a un utente per completare con successo una funzionalità specifica del sistema per la prima volta, senza assistenza esterna. Misura l'apprendibilità dell'interfaccia utente.
- **Formula**:

$$\mathrm{TT} = \frac{\sum_{i=1}^{n} \text{Tempo di Completamento}_i}{n}$$

  dove n è il numero di utenti coinvolti nella sessione di test di usabilità. Il tempo è misurato dal momento in cui l'utente inizia l'interazione con la funzionalità fino al completamento (o abbandono) della stessa.
- **Unità di misura**: minuti

5.2.4. Efficienza

5.2.4.1. MPD-RT: Response Time

- **Descrizione**: Tempo di risposta del sistema a un input dell'utente, misurato come intervallo tra l'invio della richiesta e la ricezione della risposta completa.
- **Formula**:

$$\mathrm{RT} = \text{Timestamp Risposta} - \text{Timestamp Richiesta}$$

  La misurazione avviene tramite test automatici di performance che simulano le richieste HTTP verso gli endpoint del sistema.
- **Unità di misura**: secondi (s)

5.2.5. Manutenibilità

5.2.5.1. MPD-CS: Code Smell Density

- **Descrizione**: Densità di *code smell* nel codice sorgente. I *code smell* sono indicatori di progettazione debole (metodi troppo lunghi, classi troppo grandi, duplicazione di codice) che, pur non essendo errori bloccanti, rendono il sistema fragile e difficile da manutenere.
- **Formula**:

$$\mathrm{CS} = \frac{\text{Numero di Code Smell Rilevati}}{\mathrm{CRG}}$$

  dove $\mathrm{CRG} = \frac{\text{Righe di Codice Totali}}{100}$.
- **Unità di misura**: code smell per CRG (centinaia di righe di codice)
- **Strumento di rilevazione**: analizzatore statico del codice (es. SonarQube, ESLint, golangci-lint)

5.2.5.2. MPD-COC: Coefficient of Coupling

- **Descrizione**: Misura il grado di interdipendenza tra i diversi moduli del software. Un accoppiamento elevato implica che una modifica in una parte del codice rischia di propagarsi ad altre sezioni (effetto a catena), riducendo la manutenibilità.
- **Formula**:

$$\mathrm{COC} = \frac{\text{Dipendenze Effettive tra Moduli}}{n \times (n-1)}$$

dove n è il numero totale di moduli del sistema e $n \times (n-1)$ rappresenta il numero massimo di dipendenze direzionali possibili. Una «dipendenza» sussiste quando un modulo importa, invoca o referenzia direttamente un altro modulo.
- **Interpretazione**:
  - $\mathrm{COC} = 0$: nessun accoppiamento (moduli completamente indipendenti)
  - $\mathrm{COC} = 1$: accoppiamento massimo (ogni modulo dipende da tutti gli altri)
- **Unità di misura**: adimensionale (scala 0–1)
- **Strumento di rilevazione**: analisi delle dipendenze tramite strumenti di analisi statica o ispezione manuale del grafo delle importazioni

5.2.5.3. MPD-CYC: Cyclomatic Complexity

- **Descrizione**: Misura la complessità logica del codice contando il numero di percorsi linearmente indipendenti attraverso il flusso di controllo (costrutti `if`, `switch`, `loop`). Valori elevati indicano codice difficile da testare e comprendere.
- **Formula**:

$$M = E - N + 2P$$

dove:
  - E: numero di archi nel grafo di controllo del flusso
  - N: numero di nodi (istruzioni) nel grafo di controllo del flusso
  - P: numero di componenti connesse del grafo (solitamente $P = 1$ per una singola funzione)
- **Unità di misura**: numero intero (adimensionale). Il valore riportato nel cruscotto è la media della complessità ciclomatica su tutte le funzioni del sistema.
- **Strumento di rilevazione**: analizzatore statico del codice (es. `gocyclo` per Go, ESLint per TypeScript)

5.2.6. Sicurezza

5.2.6.1. MPD-DE: Data Encryption Coverage

- **Descrizione**: Percentuale di flussi dati classificati come sensibili (dati dei tenant, credenziali di autenticazione, dati IoT in transito) che sono protetti da cifratura, sia a riposo (*at rest*) che in transito (*in transit*).
- **Formula**:

$$\mathrm{DE} = \left(\frac{\text{Flussi Dati Sensibili Cifrati}}{\text{Flussi Dati Sensibili Totali}} \right) \times 100$$

Un «flusso dati sensibile» è ogni canale di comunicazione o punto di persistenza che tratta dati personali, credenziali o dati soggetti a separazione multi-tenant. La classificazione dei flussi sensibili è stabilita in fase di progettazione architetturale.
- **Unità di misura**: percentuale (%)

5.2.7. Metriche di progresso dei test

5.2.7.1. MPT-TS: Test di Sistema Specificati

- **Descrizione**: Percentuale di test di sistema definiti rispetto al numero totale di requisiti funzionali presenti nell'**Analisi dei Requisiti**$_{G}$. Misura il grado di completamento nella specifica dei test.
- **Formula**:

$$\mathrm{TS} = \left(\frac{\text{Test di Sistema Specificati}}{\text{Requisiti Funzionali Totali}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

5.2.7.2. MPT-TE: Test di Sistema Eseguiti

- **Descrizione**: Percentuale di test di sistema specificati che sono stati effettivamente eseguiti. Misura il progresso nell'esecuzione della campagna di test.
- **Formula**:

$$\mathrm{TE} = \left(\frac{\text{Test di Sistema Eseguiti}}{\text{Test di Sistema Specificati}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

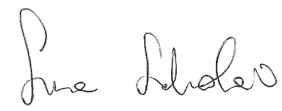
5.2.7.3. MPT-TP: Test di Sistema Superati

- **Descrizione**: Percentuale di test di sistema eseguiti che sono stati superati con successo. Misura la conformità del sistema ai requisiti funzionali verificati.
- **Formula**:

$$\mathrm{TP} = \left(\frac{\text{Test di Sistema Superati}}{\text{Test di Sistema Eseguiti}} \right) \times 100$$

- **Unità di misura**: percentuale (%)

Siria Salvalaio



Firma del revisore interno