UNIVERSITÀ DELLA CALABRIA

DIPARTIMENTO DI
INGEGNERIA INFORMATICA,
MODELLISTICA, ELETTRONICA
E SISTEMISTICA

DIMES

Corso di laurea Magistrale in ingegneria Informatica - Cybersecurity

INTELLIGENZA ARTIFICIALE

RELAZIONE PROGETTO

QuestMaster

Professore

Prof. Francesco Scarcello

Prof. Davide Mario Longo

Candidato

Vincenzo Silva 257197

Sommario

Indice delle figure

Introduzione

L'obiettivo di questa relazione è quello di descrivere le fasi di realizzazione del progetto, illustrando le scelte progettuali adottate, le tecnologie utilizzate e le soluzioni adottate.

Verranno inoltre presentati i risultati ottenuti e degli esempi di progettazione con la relativa documentazione

Scopo del progetto

QuestMaster è un sistema in due fasi progettato per aiutare gli autori a creare esperienze narrative interattive attraverso tecniche di pianificazione classica (PDDL) e intelligenza artificiale generativa (LLM).

Il sistema comprende una Fase di Generazione della Storia (Fase 1) e una Fase di Gioco Narrativo Interattivo (Fase 2).

Nella prima fase, il sistema assiste l'autore nella generazione di una quest narrativa logicamente coerente sotto forma di problema di pianificazione.

Nella seconda fase, a seguito della quest validata e della generazione dei file PDDL, viene generata un'applicazione web interattiva che consente all'utente finale di esplorare la storia, effettuare delle scelte e arrivare all'obiettivo finale.

Descrizione progetto

In questa sezione vengono riportate una descrizione delle soluzioni progettuali principali del progetto

Il progetto QuestMaster è strutturato con le seguenti tecnologie:

- **LLM**: è stato adottato Gemini, un modello linguistico di grandi dimensioni (Large Language Model) sviluppato da Google DeepMind, progettato per comprendere e generare testo in linguaggio naturale, codice e contenuti multimodali. È caratterizzato da elevate capacità di ragionamento, contestualizzazione e coerenza narrativa, ideali per sistemi che combinano pianificazione e generazione di storie

- **Reflective agent**: In QuestMaster, questa funzione è realizzata attraverso l'uso di Gemini, che riflette sui risultati generati (scelte narrative, risposte, sviluppi della trama) e li confronta con gli obiettivi o il contesto precedente.
  Grazie alle sue capacità di reasoning e metariflessione, Gemini può valutare la coerenza e la qualità narrativa, modificando o ottimizzando la propria strategia per migliorare la continuità e la profondità della storia interattiva.

- **Planner automatizzato**: come planner automatizzato basato su tecniche di pianificazione classica (PDDL – Planning Domain Definition Language) è stato adottato Fast Downward.
  È uno degli strumenti più usati nella ricerca sull'intelligenza artificiale per tradurre descrizioni di problemi (domini e stati iniziali) in sequenze ottimali di azioni che portano a un obiettivo.

- **Backend**: realizzato in Python, linguaggio versatile e ampiamente utilizzato per l'intelligenza artificiale e lo sviluppo web. Gestisce la logica applicativa, l'integrazione con il modello LLM, la pianificazione (PDDL).

- **Frontend**: sviluppato con JavaScript, HTML e CSS, fornisce l'interfaccia grafica e interattiva del sistema. Consente agli utenti di esplorare la storia, effettuare scelte e ricevere risposte in tempo reale dal modello, garantendo un'esperienza utente fluida e coinvolgente.

Traccia

QuestMaster is a two-phase system designed to help authors create interactive narrative experiences through classical planning techniques (PDDL) and generative AI (LLMs). It comprises a **Story Generation Phase (Phase 1)** and an **Interactive Story Game Phase (Phase 2)**.

Phase 1: Story Generation

Objective

Assist authors interactively in creating a logically consistent narrative quest represented as a PDDL planning problem.

Input

A **Lore Document** containing the following elements:

- **Quest Description**: Description of the adventure, including the initial state, goal, and possible obstacles. It also includes contextual information and background of the story's world.
- **Branching Factor**: Minimum and maximum number of actions available at each narrative state.
- **Depth Constraints**: Minimum and maximum number of steps to reach the goal.

Workflow

1. **PDDL generation and validation**
    - The system produces a PDDL file that models the proposed adventure, including the initial state, goal, actions, and other elements. Each line is accompanied by a comment that textually describes what that specific line of PDDL code does.
    - The system uses a classical planner (e.g., Fast Downward) to validate if the current story (PDDL) formulation allows at least one valid path from the initial state to a goal state.
2. **Interactive Refinement Loop (if no valid PDDL model)**
    - If no valid solution exists: an automated LLM agent (**Reflection Agent**) refines the PDDL file by identifying logical inconsistencies or narrative gaps. It then suggests specific modifications and interacts with the author through a suitable chat interface, seeking approval or further input before finalizing the refinements.
3. **Phase 1 Output**
    - A complete and validated PDDL domain and problem file.
    - A finalized lore file (if updated to consider the new version of the PDDL file).

Phase 2: Interactive Story Game

Objective

Create an interactive web-based narrative experience using the completed PDDL and lore files generated in Phase 1.

Workflow

1. **HTML Generation**
    - Use an LLM agent to generate an HTML interactive web-based implementation of the adventure described in the Lore file and modeled by the computed PDDL file.
    - Optionally, generate state-specific images representing the narrative context and upcoming choices.

Project files to be delivered

- **Project Files**: Zip file with implemented code
- **Example of a Quest output:** Input Lore document, and the output files (PDDL and HTML).

Struttura del progetto

In questa sezione viene descritta la struttura del progetto e del contenuto delle cartelle

Dentro la cartella QuestMaster sono presenti i seguenti file e cartelle:

- **Lore:** è un file di testo che descrive l'avventura. Al suo interno sono presenti una Descrizione della Quest (descrizione dell'avventura, include lo stato iniziale, l'obiettivo e i possibili

ostacoli). Comprende inoltre informazioni contestuali e il background del mondo narrativo in cui si svolge la storia. Fattore di diramazione (Branching Factor): numero minimo e massimo di azioni disponibili in ciascuno stato narrativo. Vincoli di profondità (Depth Constraints): numero minimo e massimo di passaggi necessari per raggiungere l'obiettivo.

- **Questmaster_phase1:** è il file python che si occupa della creazione dei file generati nella fase 1, l'integrazione con il modello LLM e la pianificazione (PDDL).

- **Run_phase1:** Questo file funge da punto di ingresso per la prima fase di QuestMaster, coordinando l'integrazione tra Gemini (LLM) e Fast Downward (planner PDDL) per generare la base logica della storia

- **QuestMaster:** è la pagina frontend che si occupa di generare un'esperienza interattiva attraverso l'utilizzo di un agente LLM e i file PDDL generati nella fase 1.

- **Phase1_output/:** cartella al cui interno sono presenti i file PDDL generati nella fase 1.

- **Fast-downward-24.06.1/:** all'interno di questa cartella è presente il planner automatizzato basato su tecniche di pianificazione classica per la validazione dell'output prodotto nella fase 1.

- **Proxy.js:** questo script implementa un server proxy Node.js basato su Express, utilizzato nella Fase 2 per gestire la comunicazione tra l'applicazione web e il modello di generazione visiva ospitato su Hugging Face.

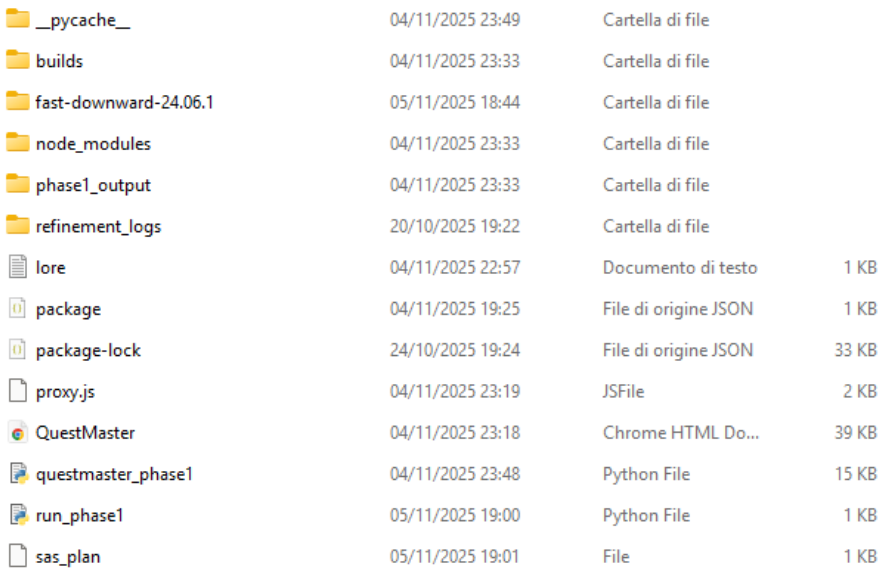


Figura 1- composizione cartella

Fase 1

La Fase 1 è stata schematizzata nei seguenti bullet point:

- **Accessibilità e semplicità di esecuzione**: L'avvio della Fase 1 è stata gestita attraverso l'utilizzo della CLI. Il file che si occupa di avviare è il run_phase1.py, al suo interno è stato strutturato ogni collegamento per permettere una corretta gestione degli input/output prodotti.

- **Caratteristiche modello utilizzato**: il nome del modello è gemini-2.5-flash, una versione ottimizzata per velocità e risposte in tempo reale. Inizializzato mediante la libreria google.generativeai con temperatura impostata a 0.4.

- **Ricerca documento Lore:** Se il file è presente, vengono estratte automaticamente le informazioni necessarie alla generazione della storia, ovvero: la descrizione della quest (Quest Description), il valore minimo e massimo del Branching Factor, il valore minimo e massimo del Depth Constraints. Nel caso in cui il file di Lore non sia presente, il sistema richiede all'utente, tramite interfaccia a riga di comando (CLI), di inserire manualmente le stesse informazioni: descrizione della storia, valori minimi e massimi del Branching Factor,
  valori minimi e massimi del Depth Constraints. In questo modo, l'utente non è vincolato alla creazione preventiva di un documento di Lore, ma può scegliere di inserire i dettagli dell'avventura direttamente da console.

- **Generazione dei file PDDL:** Una volta che il sistema ha ricevuto dall'utente la descrizione dell'avventura e le informazioni necessarie, il sistema procede automaticamente alla generazione dei file domain.pddl e problem.pddl, necessarie per la pianificazione automatica. La funzione "generate_pddl(self, lore, branching, depth)"
  si occupa di inviare alla chat di Gemini un prompt strutturato con le informazioni precedentemente inserite. Al modello è stato richiesto di definire predicati, azioni, oggetti e il goal della storia, definendo di inserire dei commenti che spiegano ogni sezione. Al termine il modello LLM restituisce un file JSON strutturato che successivamente viene decodificato.

- **Validazione PDDL:** Al termine della generazione del file JSON strutturato, viene invocato il metodo validate_pddl(self, domain_text, problem_text), che ha il compito di verificare la correttezza logica e sintattica dei file PDDL (domain e problem). Il metodo: Crea una directory temporanea in cui salva i file domain.pddl e problem.pddl generati. Esegue il planner automatizzato Fast Downward tramite un comando di sistema, utilizzando la strategia di ricerca astar(blind()). Analizza l'output prodotto da Fast Downward per determinare il risultato della validazione

- **Proposta di Correzione:** nel caso in cui il planner automatizzato Fast Downward non riesca a trovare un piano coerente e valido, viene attivato il metodo "reflection_agent(self, domain_text, problem_text, error_message, planner_output)" che utilizza il modello Gemini come agente riflessivo. Questo analizza l'errore restituito dal planner, insieme ai file PDDL generati, e propone una versione corretta dei file domain e problem. Il metodo genera un prompt dettagliato che descrive il problema, l'output del planner e il contenuto dei file, chiedendo a Gemini di individuare la causa dell'errore, correggere il codice PDDL e fornire una breve spiegazione in linguaggio naturale delle modifiche. L'utente può visualizzare e approvare le correzioni proposte o fornire un feedback, consentendo al sistema di rielaborare i file in modo iterativo fino a ottenere una versione valida e coerente.

- **Approccio Human-in-the-loop:** questo approccio è l'integrazione dell'intervento umano all'interno di un processo automatizzato, con l'obiettivo di combinare la precisione e la velocità dei modelli di intelligenza artificiale con il giudizio e la creatività dell'essere umano. Nel contesto di *QuestMaster*, l'autore o il designer può supervisionare, correggere o approvare i risultati generati dal sistema (come piani PDDL, trame narrative o correzioni proposte dal reflective agent). In questo modo si garantisce un equilibrio tra automazione e controllo umano, migliorando la qualità, la coerenza narrativa e l'affidabilità complessiva del processo di generazione.

- **Generazione Output finale:** Al termine dello script vengono generati quelli che sono gli output finale del processo. Nella cartella "phase1_output" vengono generati i seguenti file: domain.pddl, problem.pddl e report.json. Quest'ultimo file è un recap delle informazioni inserite e un report dei processi eseguiti all'interno dello script

Fase 2

Al termine della Fase 1 (con conseguente generazione dei file PDDL) è possibile eseguire la seconda fase.

La Fase 2 è stata schematizzata nei seguenti bullet point:

- **Caricamento dei file PDDL:** L'avvio della seconda fase avviene tramite interfaccia web. L'utente carica i file domain.pddl e problem.pddl generati nella Fase 1, utilizzando l'apposita sezione "Carica i File PDDL". Il sistema verifica la presenza di entrambi i file e, in caso di successo, li analizza per estrarre azioni, predicati, oggetti, stato iniziale e goal della missione.

- **Parsing e strutturazione dei dati:** I file PDDL vengono letti e convertiti in strutture dati interne JavaScript tramite funzioni di parsing dedicate (*tokenize*, *readFromTokens*, *parsePDDL*). Queste funzioni traducono la sintassi PDDL in oggetti JSON, consentendo di rappresentare nel browser i concetti di dominio, precondizioni, effetti e stato logico.

- **Inizializzazione del gioco:** Dopo il caricamento dei file, viene avviato il motore logico della fase di gioco attraverso la funzione initGame(). Viene creato lo stato iniziale (INITIAL_STATE) e vengono rese disponibili le prime azioni che il giocatore può compiere. Il sistema genera una breve introduzione narrativa che presenta la missione e il contesto.

- **Generazione dinamica delle azioni:** La funzione *renderActions()* invoca il modello Gemini 2.0 Flash per determinare, in base allo stato corrente PDDL, quali azioni siano logicamente eseguibili. Il modello restituisce un elenco JSON di azioni pertinenti e semanticamente coerenti con la situazione di gioco. Ogni azione viene trasformata in un pulsante interattivo che l'utente può selezionare.

- **Esecuzione delle azioni e aggiornamento dello stato:** Quando il giocatore seleziona un'azione, il sistema la esegue invocando executeAction(). Gli effetti logici dell'azione vengono applicati allo stato corrente tramite la funzione applyEffects(), che aggiorna dinamicamente i predicati attivi (aggiungendo o rimuovendo fatti coerenti con l'azione eseguita). Il nuovo stato viene visualizzato in tempo reale nella sezione "Stato PDDL".

- **Generazione narrativa e visiva**: Dopo ogni azione, la funzione generateNarrativeAndRender() invia al modello Gemini 2.0 Flash un prompt che include lo stato PDDL aggiornato e la descrizione dell'azione. Il modello genera due elementi: Narrativa (un testo immersivo in italiano che descrive in modo coinvolgente la scena successiva e le emozioni del protagonista). Prompt visivo (una breve descrizione per la generazione di un'immagine cinematografica tramite Imagen API, che rappresenta la scena attuale). In questo modo, ogni azione produce sia una narrazione dinamica sia un'immagine coerente con la storia.

- **Rappresentazione interattiva dello stato:** La sezione "Stato PDDL" visualizza, con tag colorati e aggiornamento continuo, i fatti logici che compongono lo stato attuale. Le condizioni positive e negative sono differenziate cromaticamente per migliorare la leggibilità e favorire la comprensione dell'evoluzione narrativa.

- **Gestione del goal e completamento della missione:** Il sistema controlla costantemente se le condizioni del goal definite nel problem.pddl sono soddisfatte.
  In caso positivo, viene visualizzato un messaggio di vittoria con un effetto grafico dedicato e la possibilità di scaricare in locale un file JSON contenente la cronologia della storia e delle azioni svolte (downloadGameHistory()).

- **Approccio Human-in-the-loop:** Anche nella Fase 2 viene mantenuto l'approccio Human-in-the-loop. L'utente interagisce direttamente con l'ambiente generato, influenzando la progressione della storia e validando le scelte narrative prodotte dal modello. In questo modo, l'intelligenza artificiale agisce come assistente creativo, mentre l'utente mantiene il controllo decisionale e direzionale sull'esperienza interattiva.

- **Proxy per la generazione delle immagini:** Il server avvia un'applicazione Express configurata con CORS e body parser JSON per accettare richieste HTTP dal frontend. È definito un endpoint POST /api/imagen che riceve un prompt testuale dal client — ovvero la descrizione visiva della scena generata dal modello linguistico (Gemini). Una volta ricevuto il prompt, il server effettua una richiesta HTTP POST all'endpoint del modello FLUX.1-dev di Hugging Face, passando il token API e il testo da trasformare in immagine. Se la risposta del modello è valida, l'immagine viene convertita in base64 e restituita al frontend sotto forma di oggetto JSON. In caso di errore o risposta non valida, il proxy gestisce le eccezioni restituendo un messaggio di errore dettagliato. Il server ascolta sulla porta 5500, fungendo da intermediario sicuro tra il frontend (che genera i prompt narrativi) e il servizio di generazione immagini, evitando di esporre direttamente le credenziali API sul lato client.

- **Output finale**: Al termine della sessione di gioco vengono generati: un file JSON scaricabile che contiene la storia completa, le scelte effettuate e gli stati logici attraversati; la rappresentazione finale del goal raggiunto; una sequenza di immagini narrative che documentano visivamente l'evoluzione dell'avventura.

Esempio funzionamento del sistema QuestMaster

In questa sezione verrà presentato un esempio a scopo illustrativo.

Fase 1

Digitando in console "**python .\run_phase1**" verrà eseguito lo script per la generazione dei file PDDL.

In questo caso è stato creato un file lore.txt con le informazioni dell'avventura.

lore.txt

```
Quest Description:
avventura ambientata nel futuro, sara walker deve combattere contro gli alieni e le macchine.
Per salvare il mondo deve riuscire a tornare indietro nel passato con la macchina del tempo.

Branching Factor: 3-4
Depth Constraints: 3-4

```

Figura 2 - dettaglio lore.txt

Il sistema all'avvio riconosce subito che è presente un file lore.txt ed inizia ad analizzarlo.

```
=== QuestMaster Phase 1: Story Generation ===

File di lore trovato: lore.txt
Lore caricata correttamente dal documento.
Branching: {'min': '3', 'max': '4'}, Depth: {'min': '3', 'max': '4'}
```

Figura 3 - avvio Fase 1

Successivamente avviene la prima fase di generazione dei file PDDL che vengono memorizzati dentro la cartella **phase1_output/initial_attempt/.** Al termine viene avviato il Fast Downward per verificare la correttezza e della validità dei file generati.

```
StoryAgent: Generazione PDDL tramite Gemini Chat...
Tentativo iniziale salvato in 'phase1_output\initial_attempt/'
Validator: eseguo Fast Downward per la validazione...
```

Figura 4 - validazione della storia

Infine, avendo trovato un piano valido, vengono generati i file PDDL definitivi e il report.json all'interno della cartella **phase1_output/.**

```
 Output salvati in 'phase1_output/'
Stato finale: Piano valido trovato
```

Figura 5 - Creazione Output

All’interno della cartella troviamo i seguenti file:

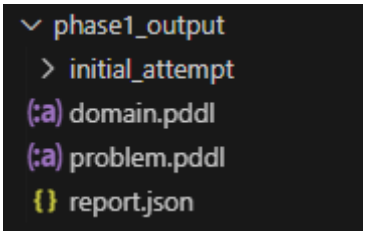


Figura 6 - Contenuto cartella Output

Possiamo notare che all’interno dei file problem.pddl e domain.pddl, sono stati generati i commenti che descrivono brevemente la stringa di codice.

```
; QuestMaster PDDL Domain for Sara Walker's Time Travel Adventure

(define (domain sara-time-travel)
    (:requirements :strips :typing)

    (:types
        character location enemy machine power-source
    )

    (:predicates
        ; Predicate to indicate a character's current location
        (at ?c - character ?l - location)
        ; Predicate to indicate if a character possesses a weapon
        (has-weapon ?c - character)
        ; Predicate to indicate if a character's weapon is charged and ready for use
        (weapon-charged ?c - character)
        ; Predicate to indicate the presence of an enemy at a specific location
        (enemy-present ?e - enemy ?l - location)
        ; Predicate to indicate if an enemy has been defeated
        (enemy-defeated ?e - enemy)
        ; Predicate to indicate the location of the time machine
        (time-machine-at ?m - machine ?l - location)
        ; Predicate to indicate if the time machine is functional
        (time-machine-functional ?m - machine)
        ; Predicate to indicate if the time machine has been charged
        (time-machine-charged ?m - machine)
        ; Predicate to indicate the location of a power source
        (power-source-at ?ps - power-source ?l - location)
        ; Predicate to indicate if a character possesses a power source
        (has-power-source ?c - character ?ps - power-source)
        ; Predicate to indicate if a character knows the time travel protocol
        (knows-time-travel-protocol ?c - character)
        ; Goal predicate: indicates if the character has successfully traveled to the past
        (in-past ?c - character)
    )
```

Figura 7 - Domain.pddl generato

phase1_output > (:a) problem.pddl

```
; QuestMaster PDDL Problem for Sara Walker's Time Travel Adventure

(define (problem sara-mission)
    (:domain sara-time-travel)

    (:objects
        ; Define the main character
        sara - character
        ; Define the primary enemy threat (aliens and machines combined)
        alien-horde - enemy
        ; Define the time machine
        time-machine-alpha - machine
        ; Define the power source needed for the time machine
        temporal-energy-cell - power-source
        ; Define various locations in the future world
        lab-hq ruined-city - location
    )

    (:init
        ; Sara starts at the lab headquarters
        (at sara lab-hq)
        ; Sara already possesses a weapon
        (has-weapon sara)
        ; Sara's weapon is currently uncharged
        (not (weapon-charged sara))
        ; The alien horde is present in the ruined city
        (enemy-present alien-horde ruined-city)
```

Figura 8 - Problem.pddl generato

All'interno del file "report.json" è presente un breve recap delle informazioni inserite nelle fase 1.

phase1_output > {} report.json > ...

```
{
  "lore": "avventura ambientata nel futuro, sara walker deve combattere contro
  "branching": {
    "min": "3",
    "max": "4"
  },
  "depth": {
    "min": "3",
    "max": "4"
  },
  "valid": true,
  "validation_message": "Piano valido trovato"
}
```

Figura 9 - Report.json generato

Fase 2

Per avviare la Fase 2 di QuestMaster è necessario che sia avviato il file “proxy.js” con lo scopo di fungere da proxy intermedio tra il frontend (applicazione web) e il modello di generazione immagini ospitato su Hugging Face.

Avviamo il server node con il comando “node proxy.js”.

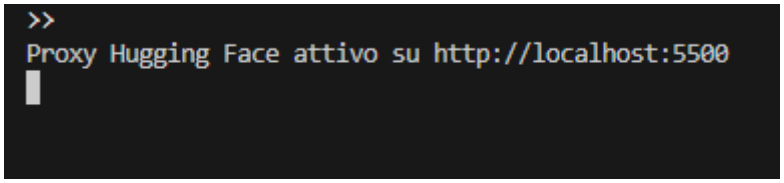


Figura 10 - Proxy attivo

Successivamente, collegandosi all’indirizzo web in cui è ospitata l’applicazione (nel nostro caso **“127.0.0.1:5500/QuestMaster.html”**), viene visualizzata la seguente schermata.

La schermata principale è la seguente:

Figura 11 - schermata principale avvio

Nei campi inseriamo i file PDDL domain e problem generati nella fase 1.

Successivamente premendo sul tato “Avvia la Quest” verrà generata l’avventura in base ai file di input.

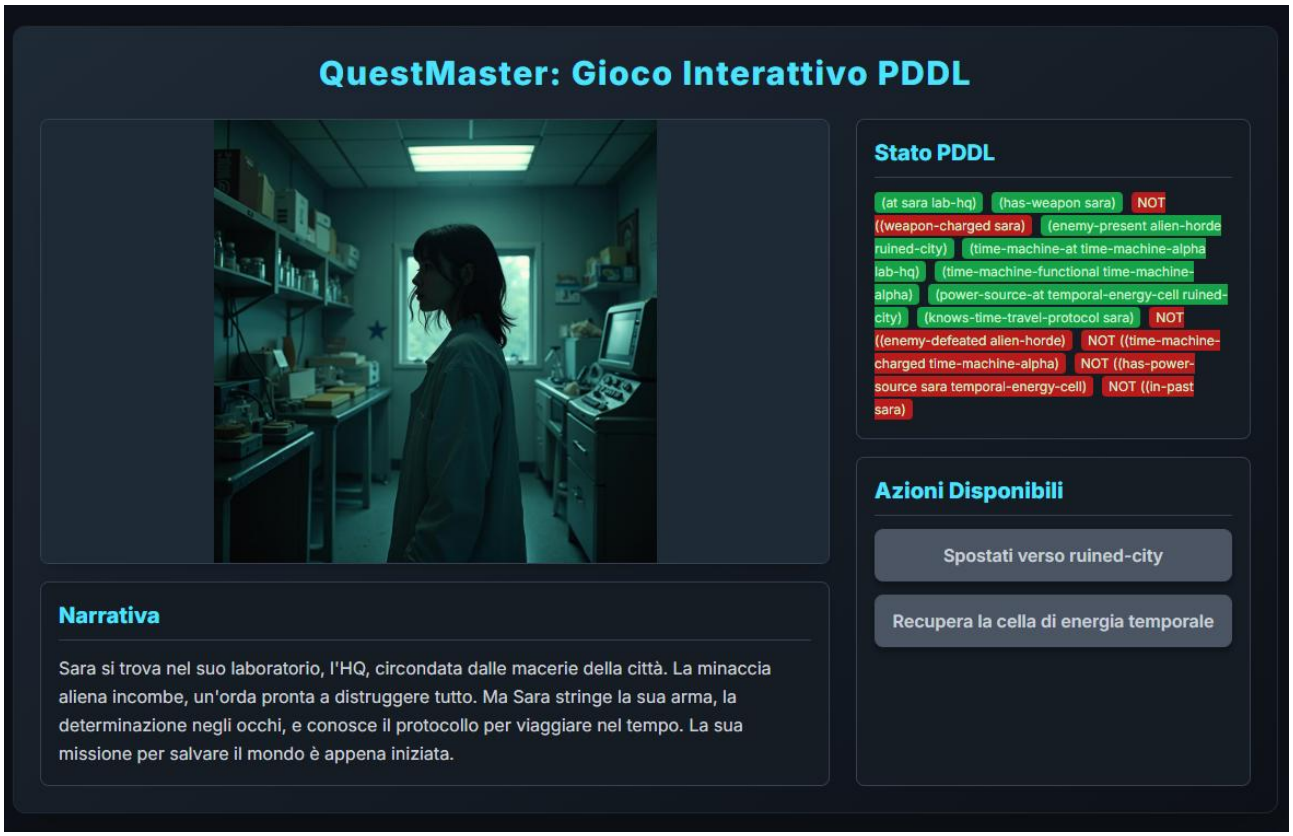


Figura 12 - esempio scelta narrativa

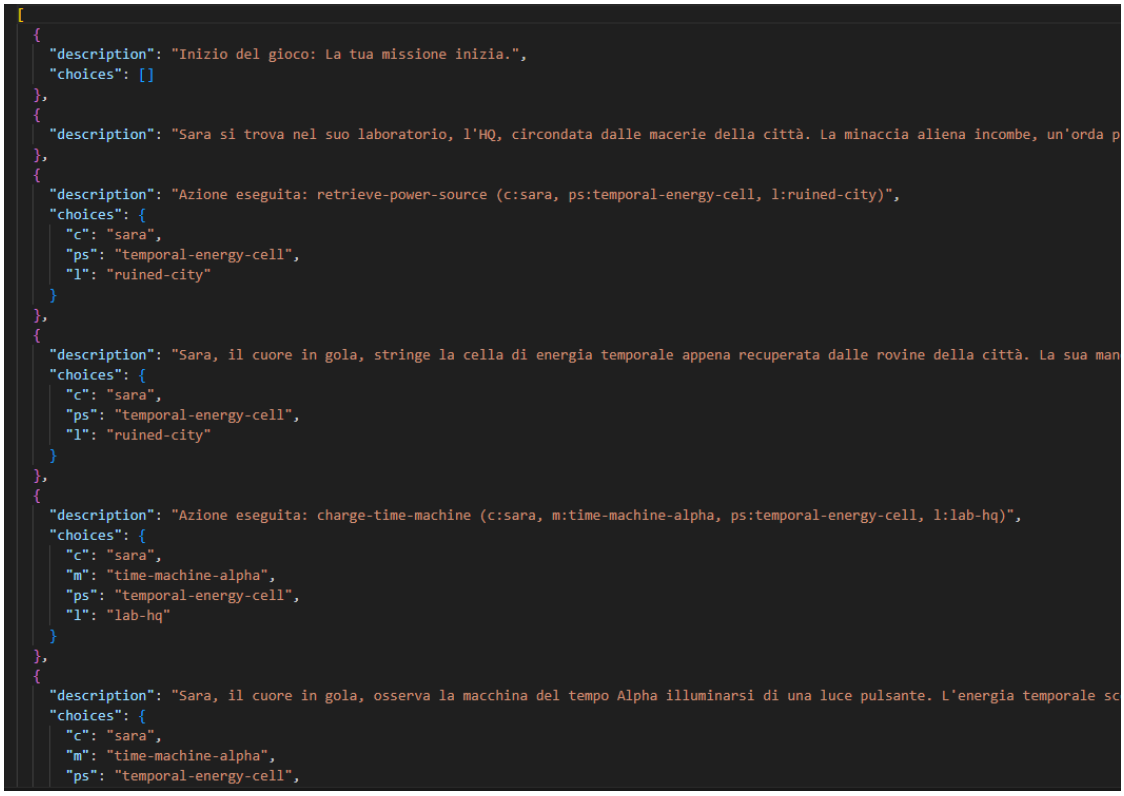
All'interno della web application, da sinistra verso destra, sono rappresentati i seguenti elementi:

- L'immagine generata, che illustra graficamente la scena narrativa prodotta dall'intelligenza artificiale.
- La narrativa, una breve descrizione dell'attività e dello stato attuale della storia.
- Lo stato PDDL, che mostra i predicati e le possibili azioni disponibili nel contesto corrente.
- Le azioni disponibili, che consentono all'utente di proseguire l'avventura interattiva.

Dopo aver selezionato le azioni proposte e completato il percorso narrativo, l'applicazione notifica l'utente con un messaggio di conferma che indica il completamento dell'avventura.

Figura 13 - esempio fase conclusiva

Da qui è possibile ricominciare con l'avventura o scaricare il report della storia appena completata, dove l'utente può visualizzare le proprie scelte e la storia appena conclusa.



```
[
  {
    "description": "Inizio del gioco: La tua missione inizia.",
    "choices": []
  },
  {
    "description": "Sara si trova nel suo laboratorio, l'HQ, circondata dalle macerie della città. La minaccia aliena incombe, un'orda p
  },
  {
    "description": "Azione eseguita: retrieve-power-source (c:sara, ps:temporal-energy-cell, l:ruined-city)",
    "choices": {
      "c": "sara",
      "ps": "temporal-energy-cell",
      "l": "ruined-city"
    }
  },
  {
    "description": "Sara, il cuore in gola, stringe la cella di energia temporale appena recuperata dalle rovine della città. La sua man
    "choices": {
      "c": "sara",
      "ps": "temporal-energy-cell",
      "l": "ruined-city"
    }
  },
  {
    "description": "Azione eseguita: charge-time-machine (c:sara, m:time-machine-alpha, ps:temporal-energy-cell, l:lab-hq)",
    "choices": {
      "c": "sara",
      "m": "time-machine-alpha",
      "ps": "temporal-energy-cell",
      "l": "lab-hq"
    }
  },
  {
    "description": "Sara, il cuore in gola, osserva la macchina del tempo Alpha illuminarsi di una luce pulsante. L'energia temporale sc
    "choices": {
      "c": "sara",
      "m": "time-machine-alpha",
      "ps": "temporal-energy-cell",
```


Figura 14 - JSON scaricabile

Considerazioni Finali

L'intelligenza artificiale ha rappresentato un supporto significativo nella realizzazione di questo progetto. Durante i diversi test effettuati, è stato osservato che, nonostante l'ampia libertà creativa, le proposte generate dal modello LLM di Gemini tendevano spesso a risultare simili tra loro sia nella struttura narrativa che nello stile descrittivo. Inoltre, in presenza di argomenti o termini particolarmente complessi, l'IA talvolta produceva risultati al limite della plausibilità.
Una delle difficoltà incontrate nel processo di sviluppo ha riguardato l'utilizzo dei generatori di immagini text-to-image: la maggior parte di quelli disponibili sul mercato è a pagamento, mentre le versioni gratuite impongono un limite massimo di immagini generabili, causando frequenti rallentamenti nel flusso di lavoro.
È stato invece molto utile poter usufruire, in qualità di studente, del pacchetto Gemini Pro, che ha consentito un impiego più esteso delle sue notevoli capacità.

Infine, nella fase di sviluppo dell'interfaccia web, è risultato complesso gestire il parsing delle soluzioni generate dal modello. In diversi casi, venivano proposte scelte duplicate o semanticamente molto simili, e, nonostante il prompt specificasse la necessità di generare contenuti esclusivamente in lingua italiana, il modello non sempre riusciva a rispettare tale vincolo.
Parallelamente, si è posta l'esigenza di mantenere la struttura dei nodi coerente con la logica narrativa originaria, evitando ripetizioni o percorsi incoerenti. La gestione dello stato interno dell'applicazione e l'aggiornamento dinamico dei nodi hanno richiesto particolare attenzione, soprattutto per garantire la

persistenza del contesto e un’esperienza utente fluida.
Nel complesso, queste difficoltà si sono rivelate occasioni di miglioramento progressivo del sistema, contribuendo a una maggiore affidabilità e alla definizione di un flusso di lavoro più solido e adattabile.

Link progetto

Il codice sorgente del progetto è visualizzabile:

https://github.com/vincenzosilva96/QuestMaster