RepoGenius

Sergio Mancini and Enrico Sorbello

Università degli Studi di Catania

27 Luglio 2024

Abstract

Questo progetto, sviluppato come parte del corso di Ingegneria dei sistemi distrubuiti nel Corso di Laurea Magistrale in Informatica presso l'Università di Catania, ha lo scopo di ralizzare un sistema distribuito che partendo da un link GitHub o da alcuni parametri, esegue un analisi sulla repo di riferimento, in base a elementi come codice, linguaggio, file ecc.

1 Introduzione

Si vuole presentare un sistema distribuito per l'analisi dei contenuti di repository GitHub. Questo progetto rappresenta un'innovativa soluzione per esaminare e valutare progetti software ospitati sulla piattaforma di sviluppo collaborativo più popolare al mondo. Il sistema è progettato per raccogliere dati e informazioni dettagliate dai repository GitHub fornendo un'analisi approfondita basata su vari parametri come il link diretto alla repository, il linguaggio di programmazione utilizzato, la struttura del codice, le metriche di qualità e molti altri criteri rilevanti. Attraverso l'utilizzo di tecnologie distribuite, il sistema garantisce scalabilità, efficienza e affidabilità nell'elaborazione di grandi volumi di dati, supportando sviluppatori, ricercatori e team di progetto nella valutazione e nel miglioramento continuo del codice sorgente e delle pratiche di sviluppo.

2 Tecnologie utilizzate

2.1 Flask

Flask [1] è un micro framework per lo sviluppo di applicazioni web in Python. Permette di creare applicazioni web, sia semplici che complesse. È spesso utilizzato per costruire applicazioni di piccole e medie dimensioni, ma può essere esteso per progetti più grandi grazie alla sua modularità.

2.1.1 Funzionalità principali

- **Routing:** Gestisce le URL e associa ciascuna URL a una funzione.
- **Template:** Utilizza il motore di template Jinja2 per generare HTML dinamico.
- **Gestione delle richieste HTTP:** Supporta GET, POST, PUT, DELETE, ecc.
- **JSON:** Supporta facilmente la gestione dei dati JSON.

2.2 RabbitMQ

RabbitMQ [2] è un sistema di message broker open source che implementa il protocollo Advanced Message Queuing Protocol (AMQP). È utilizzato per gestire e orchestrare la comunicazione tra componenti distribuite di un sistema.

2.2.1 Workflow di RabbitMQ

- **Producer (Produttore):** Invia messaggi al broker.
- **Consumer (Consumatore):** Riceve messaggi dal broker.
- **Queue (Coda):** Una coda dove i messaggi vengono memorizzati fino a quando non vengono consumati.
- **Exchange:** Un punto di entrata per i messaggi che instrada i messaggi alle code in base a regole specifiche.

2.3 Groq

Groq Llama 3 [3] [4] è un'implementazione del modello di linguaggio Llama 3 di Meta, eseguito sull'hardware proprietario di Groq, noto come Language Processing Unit (LPU). Questa integrazione mira a fornire una velocità e un'efficienza senza precedenti nelle attività di generazione del testo.

3 Architettura e Funzionamento della Pipeline di RepoGenius

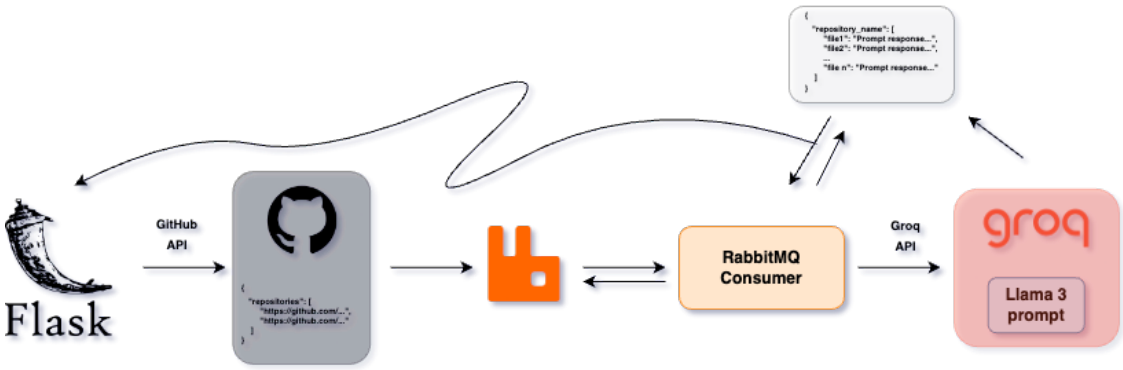


Figure 1: Pipeline

La pipeline di analisi delle repository GitHub è composta da quattro componenti principali: un server Flask, RabbitMQ, un consumer e Groq. Il processo di analisi si svolge come segue:

3.1 Microservizio Flask

Il processo inizia con un microservizio Flask, che funge da interfaccia utente e producer:

- L'utente interagisce con un'interfaccia HTML (`index.html`) dove sceglie se:
  - Inserire manualmente il link di una repository da analizzare.
  - Selezionare parametri come linguaggio e topic per la ricerca automatica di repository.
- Il server genera un JSON contenente i link delle repository da analizzare.
- Questo JSON viene inviato alla coda `Repositories` di RabbitMQ.

3.2 RabbitMQ

RabbitMQ è un message broker, all'interno della soluzione proposta, RabbitMQ:

- Mantiene la coda `Repositories` dove transitano i link delle repository da analizzare.
- Facilita la comunicazione asincrona tra il server Flask e il consumer

3.3 Consumer Python

Il consumer è responsabile dell'elaborazione dei dati:

- Ascolta continuamente la coda `Repositories` di RabbitMQ
- Quando riceve un link, scarica la repository corrispondente
- Per ogni file rilevante (es. `.py`, `.html`, `.css`, `.c`, ...):
  - Invia una richiesta API a Groq utilizzando il modello `llama3-8b-8192`
  - Fornisce il codice e un prompt specifico per l'analisi

3.4 Groq (llama3-8b-8192)

Groq esegue l'analisi del codice:

- Legge il codice fornito
- Risponde al prompt analizzando il codice:
  - Spiega la funzionalità del codice
  - Fornisce una valutazione della qualità
  - Conta i cicli `for` e `while`
  - Calcola il numero di righe
  - Propone un refactoring del codice

3.5 Risultato Finale

Al termine dell'analisi:

- Il consumer crea un JSON per la repository analizzata, contenente:
  - Il nome di ciascun file
  - La risposta del prompt di Groq per ogni file
- Questo JSON viene restituito al microservizio Flask.
- Il microservizio Flask formatta i risultati e li visualizza in una pagina HTML (`display.html`)

4 Implementazione

L'implementazione di RepoGenius è suddivisa in tre componenti principali: il microservizio Flask con relativa interfaccia utente e il consumer. Di seguito, analizzeremo ciascun componente in dettaglio.

4.1 Microservizio Flask

Il microservizio Flask gestisce le richieste dell'utente e coordinando le operazioni con RabbitMQ e il consumer. Ecco alcune delle funzionalità chiave implementate in `app.py`:

```
1
2 def trova_repositories(linguaggio='', risultati_per_pagina='', pagine='', topic=''):
3     repositories = []
4     if pagine:
5         try:
6             pagine = int(pagine)
7         except ValueError:
8             print("Errore: 'pagine' deve essere un numero intero.")
9             return []
```

```
    else:
        pagine = 1
    for pagina in range(1, pagine + 1):
        query_parts = []
        if linguaggio:
            query_parts.append(f"language:{linguaggio}")
        if topic:
            query_parts.append(f"topic:{topic}")

        query = '+'.join(query_parts)
        url = f"https://api.github.com/search/repositories?q={query}&sort=stars"

        if risultati_per_pagina:
            url += f"&per_page={risultati_per_pagina}"

        url += f"&page={pagina}"

        risposta = requests.get(url)
        if risposta.status_code == 200:
            dati = risposta.json()
            repositories.extend(dati['items'])
        else:
            print(f"Errore: {risposta.status_code}")
            return []

    return repositories

@app.route("/search", methods=["POST"])
def search():
    data = request.json
    linguaggio = data.get("linguaggio")
    topic = data.get("topic")
    risultati_per_pagina = int(data.get("risultati_per_pagina", 10))
    pagine = int(data.get("pagine", 1))

    repositories = trova_repositoriesfixed(linguaggio, risultati_per_pagina, pagine,topic)
    if repositories:
        crea_file_repository(repositories)
        return jsonify({"status": "success", "repositories": [repo['html_url'] for repo in
    repositories]})
    else:
        return jsonify({"status": "error", "message": "Errore nella ricerca dei repository"
    }), 500
```

La funzione trova_repositories ha il compito di cercare repository su GitHub in base ai criteri specificati, nella soluzione proposa nello specifico sono:

- **Linguaggio:** specifica il linguaggio di programmazione dei repository da cercare.
- **Risultati_per_pagina:** specifica quanti risultati restituire per ogni pagina (predefinito è 10).
- **Pagine:** specifica il numero di pagine di risultati da recuperare (predefinito è 1).
- **Topic:** specifica un topic (argomento) per filtrare i repository.

Se viene usato invece il secondo form (solo link), viene semplicemente aggiunto il link alla lista eseguendo un check sulla sua correttezza, senza fare nessuna chiamata api. Utilizzando i parametri forniti la funzione costruisce una query di ricerca che restityisce gli url delle repo trovate. Se la richiesta all'API di GitHub non va a buon fine, restituisce un messaggio di errore. La funzione search è un endpoint Flask che utilizza trova_repositories per cercare repository su GitHub. dopo averle trovate, le aggiunge a una lista.

```

@app.route('/analysis_results', methods=['POST'])
def analysis_results():
    event_data = request.json
    #print(event_data)

```

```
    socketio.emit('new_analysis_result', event_data)
    name = event_data.get('name', 'default_name')
    directory = "analyzed_repositories"
    if not os.path.exists(directory):
        os.makedirs(directory)
    file_name = os.path.join(directory, f"{name}.json")
    with open(file_name, 'w') as file:
        json.dump(event_data, file, indent=4)
    return jsonify({"status": "success", "results": event_data})

@app.route('/getData/<filename>')
def get_data(filename):
    directory = "analyzed_repositories"
    file_path = os.path.join(directory, f'{filename}.json')
    if not os.path.exists(file_path):
        abort(404)
    with open(file_path, 'r') as json_file:
        data = json.load(json_file)
    return render_template('display.html', data=data)
```

Analysis_results è una route che riceve e gestisce i risultati di un'analisi. Ecco una spiegazione passo per passo:

1. **Ricezione dei dati**:
   - `event\_data = request.json`: Riceve i dati JSON dal corpo della richiesta POST.
2. **Emissione di un evento tramite socket**:
   - `socketio.emit('new_analysis_result', event_data)`: Emette un evento tramite Socket.IO chiamato `'new_analysis_result'` con i dati ricevuti. Questo permette la comunicazione in tempo reale con i client connessi.
3. **Gestione del nome del file**:
   - `name = event_data.get('name', 'default_name')`: Estrae il nome dal JSON ricevuto; se non è presente, usa 'default_name'.
4. **Creazione della directory**:
   - `directory = "analyzed_repositories"`: Specifica la directory dove verranno salvati i file.
   - `if not os.path.exists(directory): os.makedirs(directory)`: Controlla se la directory esiste, altrimenti la crea.
5. **Salvataggio dei dati su file**:
   - `file_name = os.path.join(directory, f"\{name\}.json")`: Costruisce il percorso completo del file.
   - `with open(file_name, 'w') as file: json.dump(event_data, file, indent=4)`: Apre (o crea) il file in modalità scrittura e salva i dati JSON con indentazione per una migliore leggibilità.
6. **Restituzione della risposta**:
   - `return jsonify(\{"status": "success", "results": event\_data\})`: Restituisce una risposta JSON con lo stato di successo e i dati ricevuti.

Get_data invece permette di recuperare e visualizzare i dati salvati in un file JSON.

1. **Costruzione del percorso del file**:
   - `directory = "analyzed_repositories"`: Specifica la directory dove si trovano i file salvati.
   - `file\_path = os.path.join(directory, f'\{filename\}.json')`: Costruisce il percorso completo del file basato sul nome del file passato come parametro nell'URL.

2. **Controllo dell'esistenza del file**:
    - `if not os.path.exists(file_path): abort(404)`: Se il file non esiste, restituisce un errore 404 (File non trovato).
3. **Lettura del file JSON**:
    - `with open(file_path, 'r') as json_file: data = json.load(json_file)`: Apre il file in modalità lettura e carica i dati JSON.
4. **Rendering del template**:
    - `return render_template('display.html', data=data)`: Restituisce una pagina HTML renderizzata utilizzando il template `display.html`, passando i dati JSON come contesto.

4.2 Interfaccia Utente

L'interfaccia utente è implementata utilizzando HTML, CSS (Tailwind) e JavaScript. La pagina `index.html` offre due modalità di input:

1. Ricerca di repository per linguaggio, risultati per pagina e numero di pagine
2. Aggiunta diretta di un singolo repository tramite URL

Un esempio di implementazione del form di ricerca:

```
<form id="searchForm" class="bg-white p-4 rounded shadow-md w-full h-full flex flex-col
    justify-center">
    <h2 class="text-center text-xl font-semibold mb-4">Cerca Repository</h2>
    <div class="mb-4">
        <label for="linguaggio" class="block text-gray-700">Linguaggio</label>
        <input type="text" id="linguaggio" name="linguaggio" class="mt-1 block w-full px-3
    py-2 border border-gray-300 rounded-md shadow-sm focus:outline-none focus:ring focus:
    border-blue-300" required>
    </div>
    <!-- Altri campi del form -->
    <button type="submit" class="bg-blue-500 text-white px-4 py-2 rounded">Cerca</button>
</form>
```

4.3 Consumer

Il consumer è responsabile dell'analisi dei repository. Utilizza la libreria Groq per l'analisi del codice e RabbitMQ per la comunicazione asincrona. Ecco un estratto significativo:

```
def analyze_code(file_path, client):
    with open(file_path, 'r') as file:
        code_content = file.read()

    if len(code_content) > MAX_CODE_LENGTH:
        code_content = code_content[:MAX_CODE_LENGTH]

    while True:
        try:
            chat_completion = client.chat.completions.create(
                messages=[
                    {
    "role": "user",
    "content": f"""Here is a piece of code:
    {code_content}
Answer the following questions:

1. Brief description: Write a brief description of what the code does (maximum 40 words).
2. Code evaluation: Give an overall rating of the code from 1 to 10, considering factors
    like clarity, efficiency, and best practices.
3. Code analysis:
```

```
   a. Number of lines of code: Count the number of lines in the code.
   b. Number of loops: Count the number of loops (for, while) in the code.
   c. Number of if statements: Count the number of if statements in the code.
   d. Number of functions/methods: Count the number of functions and methods in the code.
   e. Line numbers of constructs: List the starting line numbers of each loop, if statement,
     and function/method in the code.

4. Code improvement and refactoring:
   a. Code improvement: Provide specific suggestions on how to improve the code, focusing on
     clarity, efficiency, and best practices.
   b. Original code: Display the original code snippet.
   c. Refactored code: Present a refactored version of the code snippet based on the
    improvement suggestions.

Answers:

1. Brief description:
2. Code evaluation:
3. Code analysis:
   a. Number of lines of code:
   b. Number of loops:
   c. Number of if statements:
   d. Number of functions/methods:
   e. Line numbers of constructs:

4. Code improvement and refactoring:
   a. Code improvement:
   b. Original code:
   c. Refactored code:
    """
}
                ],
                model="llama3-8b-8192",
            )
            return chat_completion.choices[0].message.content
        except Exception as e:
            error_message = str(e)
            if "rate limit" in error_message:
                print(f" [!] Rate limit reached. Retrying in 10 seconds...")
                time.sleep(10)
            elif "context_length_exceeded" in error_message:
                print(f" [!] Message too long. Skipping file {file_path}")
                return "Error: Message too long to be processed."
            else:
                print(f" [!] Unexpected error: {e}")
                return f"Error: {e}"
```

Questa funzione si occupa di analizzare un file di codice sorgente utilizzando un modello di intelligenza artificiale per rispondere a una serie di domande predefinite. Nel contesto della funzione `analyze_code`, il prompt serve a fornire al modello di intelligenza artificiale le istruzioni necessarie per analizzare il codice sorgente. Il prompt contiene una serie di domande e richieste specifiche, strutturate in modo chiaro e dettagliato, che guidano il modello nella produzione delle risposte desiderate. Le principali componenti del prompt includono:

- **Descrizione del Codice**: Una breve descrizione di ciò che il codice fa, limitata a un massimo di 40 parole.
- **Valutazione del Codice**: Una valutazione complessiva del codice su una scala da 1 a 10, considerando fattori come chiarezza, efficienza e aderenza alle migliori pratiche.
- **Analisi del Codice**: Un'analisi dettagliata che comprende:
  - Numero di righe di codice.
  - Numero di cicli (for, while).
  - Numero di dichiarazioni `if`.

  - Numero di funzioni/metodi.
  - Numeri di riga delle varie strutture (cicli, `if`, funzioni/metodi).

- **Miglioramento e Refactoring del Codice**: Suggerimenti specifici per migliorare il codice, la presentazione del codice originale e una versione rifattorizzata del codice basata sui suggerimenti di miglioramento.

```
def analyze_repository(repo_url, counter, lock):
    load_dotenv()
    client = Groq(api_key=os.getenv("GROQ_API_KEY"))

    repo_name = repo_url.split('/')[-1].replace('.git', '')

    repo_path = os.path.join('../Repo', repo_name)

    if not os.path.exists(repo_path):
        try:
            os.makedirs(repo_path)
            print(f" [x] Created directory {repo_path}")
        except OSError as e:
            print(f" [!] Failed to create directory {repo_path}: {e}")
            return

        try:
            subprocess.run(['git', 'clone', repo_url, repo_path], check=True)
            print(f" [x] Cloned {repo_url} into {repo_path}")
        except subprocess.CalledProcessError as e:
            print(f" [!] Failed to clone {repo_url}: {e}")
            return
    else:
        print(f" [x] Repository directory {repo_path} already exists")

    valid_files = []
    for root, _, files in os.walk(repo_path):
        for file in files:
            if file.split('.')[-1] in VALID_EXTENSIONS:
                valid_files.append(os.path.join(root, file))

    if not valid_files:
        print(" [!] No files with valid extensions found. Stopping analysis.")
        return

    analysis_results = {}
    for file_path in valid_files:
        try:
            result = analyze_code(file_path, client)
            analysis_results[os.path.relpath(file_path, repo_path)] = result
            print(f" [x] Analyzed {file_path}")
        except Exception as e:
            print(f" [!] Failed to analyze {file_path}: {e}")

    analyzed_repo_path = os.path.join('../repo_analyzed', repo_name)
    if not os.path.exists(analyzed_repo_path):
        os.makedirs(analyzed_repo_path)

    json_path = os.path.join(analyzed_repo_path, 'analysis_results.json')
    with open(json_path, 'w') as json_file:
        json.dump(analysis_results, json_file, indent=4)
    print(f" [x] Saved analysis results to {json_path}")

    # Delete the repository directory after JSON file creation
    try:
        shutil.rmtree(repo_path)
        print(f" [x] Deleted repository directory {repo_path}")
    except OSError as e:
        print(f" [!] Error deleting repository directory {repo_path}: {e}")

```

```
    with lock:
        counter.value -= 1
        print(f" [x] Active processes: {counter.value}")

    analysis_results["name"]=repo_name
    r = requests.post('http://localhost:5001/analysis_results', json=analysis_results)
    if r.status_code == 200:
        try:
            shutil.rmtree(analyzed_repo_path)
            print(f" [x] Deleted analyzed repository directory {analyzed_repo_path}")
        except OSError as e:
            print(f" [!] Error deleting analyzed repository directory {analyzed_repo_path}:
{e}")
```

La funzione `analyze_repository` si occupa di analizzare un repository GitHub specificato dall'URL, utilizzando il client di un'API (presumibilmente Groq) per analizzare il codice sorgente contenuto nel repository. La funzione gestisce il cloning del repository, la selezione dei file da analizzare, la conduzione dell'analisi e il salvataggio dei risultati. Inoltre, la funzione si occupa di gestire i processi concorrenti utilizzando un contatore e un lock. **Parametri**

- `repo_url`: L'URL del repository GitHub da analizzare.
- `counter`: Un contatore condiviso tra processi per tracciare il numero di processi attivi.
- `lock`: Un lock per sincronizzare l'accesso al contatore tra i processi concorrenti.

Passaggi

1. **Caricamento delle variabili d'ambiente e inizializzazione del client API**:

```
    load_dotenv()
    client = Groq(api_key=os.getenv("GROQ_API_KEY"))
```

2. **Preparazione del percorso del repository**:

```
    repo_name = repo_url.split('/')[-1].replace('.git', '')
    repo_path = os.path.join('../Repo', repo_name)
```

3. **Creazione della directory del repository se non esiste**:

```
    if not os.path.exists(repo_path):
        try:
            os.makedirs(repo_path)
            print(f" [x] Created directory {repo_path}")
        except OSError as e:
            print(f" [!] Failed to create directory {repo_path}: {e}")
            return
```

4. **Cloning del repository**:

```
    try:
        subprocess.run(['git', 'clone', repo_url, repo_path], check=True)
        print(f" [x] Cloned {repo_url} into {repo_path}")
    except subprocess.CalledProcessError as e:
        print(f" [!] Failed to clone {repo_url}: {e}")
        return
```

5. **Raccolta dei file validi da analizzare**:

```
    valid_files = []
    for root, _, files in os.walk(repo_path):
        for file in files:
            if file.split('.')[-1] in VALID_EXTENSIONS:
                valid_files.append(os.path.join(root, file))
```

6. **Analisi dei file validi**:

```
    analysis_results = {}
    for file_path in valid_files:
        try:
            result = analyze_code(file_path, client)
            analysis_results[os.path.relpath(file_path, repo_path)] = result
            print(f" [x] Analyzed {file_path}")
        except Exception as e:
            print(f" [!] Failed to analyze {file_path}: {e}")
```

7. **Salvataggio dei risultati dell'analisi in un file JSON**:

```
    analyzed_repo_path = os.path.join('../repo_analyzed', repo_name)
    if not os.path.exists(analyzed_repo_path):
        os.makedirs(analyzed_repo_path)

    json_path = os.path.join(analyzed_repo_path, 'analysis_results.json')
    with open(json_path, 'w') as json_file:
        json.dump(analysis_results, json_file, indent=4)
    print(f" [x] Saved analysis results to {json_path}")
```

8. **Eliminazione della directory del repository dopo la creazione del file JSON**:

```
    try:
        shutil.rmtree(repo_path)
        print(f" [x] Deleted repository directory {repo_path}")
    except OSError as e:
        print(f" [!] Error deleting repository directory {repo_path}: {e}")
```

9. **Aggiornamento del contatore dei processi attivi**:

```
    with lock:
        counter.value -= 1
        print(f" [x] Active processes: {counter.value}")
```

10. **Invio dei risultati dell'analisi a un endpoint remoto e eliminazione della directory del repository analizzato**:

```
    analysis_results["name"] = repo_name
    r = requests.post('http://localhost:5001/analysis_results', json=analysis_results)
    if r.status_code == 200:
        try:
            shutil.rmtree(analyzed_repo_path)
            print(f" [x] Deleted analyzed repository directory {analyzed_repo_path}")
        except OSError as e:
            print(f" [!] Error deleting analyzed repository directory {
    analyzed_repo_path}: {e}")
```

5 Configurazione e Setup

Per configurare e avviare RepoGenius, seguire questi passaggi:

1. Installare le dipendenze necessarie:

```
    pip install -r requirements.txt
```

2. Configurare le variabili d'ambiente in un file `.env`:

```
    RABBITMQ_HOST=localhost
    GROQ_API_KEY=your_groq_api_key_here
```

3. Avviare il server Flask e RabbitMQ (tramite `docker-compose`):

```
    docker-compose up --build
```

4. Avviare il consumer:

```
    cd rabbitmq/rabbitmq-consumer
    python consumer.py
```

6 Analisi delle Prestazioni

Le prestazioni di RepoGenius dipendono da diversi fattori:

- Velocità di risposta dell'API di GitHub per la ricerca dei repository
- Capacità di elaborazione di RabbitMQ
- Velocità di analisi del codice da parte di Groq
- Numero di file e dimensione dei repository analizzati

Per ottimizzare le prestazioni, il sistema utilizza il multiprocessing per analizzare più repository contemporaneamente:

```
def worker_main(repo_url, counter, lock):
    with lock:
        counter.value += 1
        print(f" [x] Active processes: {counter.value}")
    analyze_repository(repo_url, counter, lock)

def callback(ch, method, properties, body):
    repo_url = body.decode()
    print(f" [x] Received {repo_url}")
    process = multiprocessing.Process(target=worker_main, args=(repo_url, counter, lock))
    process.start()
```

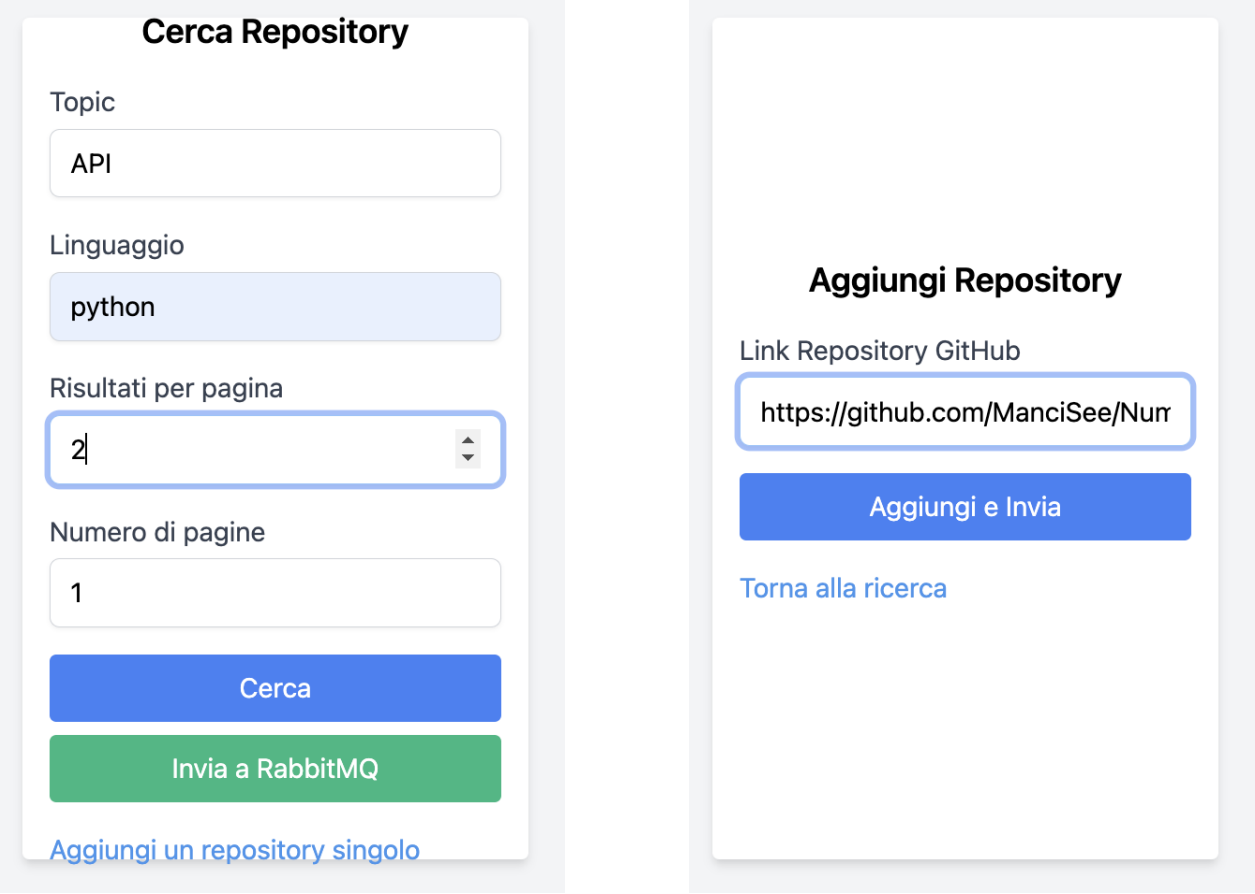
Questa implementazione permette di scalare l'analisi su più core del processore, migliorando significativamente il throughput del sistema.

7 Demo Utente

Questa sezione fornisce una guida passo-passo su come utilizzare RepoGenius, illustrando le principali funzionalità dell'interfaccia utente con l'ausilio di screenshot.

7.1 Pagina Iniziale

All'apertura di RepoGenius, l'utente viene accolto dalla pagina iniziale.



(a) Scelta dei parametri

(b) Aggiunta di un link specifico

Figure 2: Homepage

Come mostrato in Figura 2, la pagina iniziale presenta due opzioni principali:

- Ricerca di repository per linguaggio o per topic
- Aggiunta diretta di un repository tramite URL

Per cercare repository in base al linguaggio di programmazione:

1. Inserire il topic desiderato nel campo "Topic"
2. Inserire il linguaggio desiderato nel campo "Linguaggio"
3. Specificare il numero di risultati per pagina
4. Indicare il numero di pagine da analizzare

5. Cliccare su "Cerca"

6. Per inviare le repositories a RabbitMQ cliccare su "Invia a RabbitMQ"

7.2 Problematiche riscontrate

Durante la realizzazione del progetto, è emerso un problema relativo alla lunghezza dell'input da fornire al modello LLaMA 3. Le API gratuite utilizzate non consentivano di elaborare input troppo grandi, si è deciso per tanto di applicare un limite alla lunghezza del codice in input pari a mille righe.

7.3 Visualizzazione dei Risultati

Dopo l'invio di una richiesta, RepoGenius inizierà l'analisi dei repository.

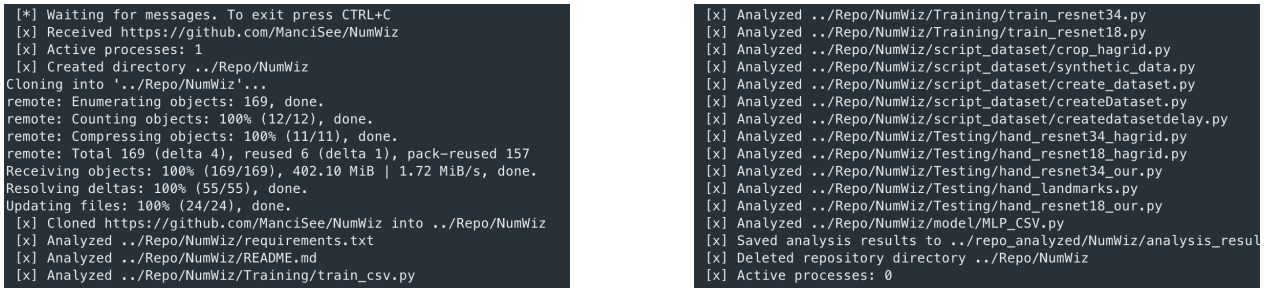


Figure 3: Analisi dei file della repository

Come si vede in Figura 3, il repository viene scaricato e dopo inizia l'analisi per ogni file presente, infine la repository verrà cancellata

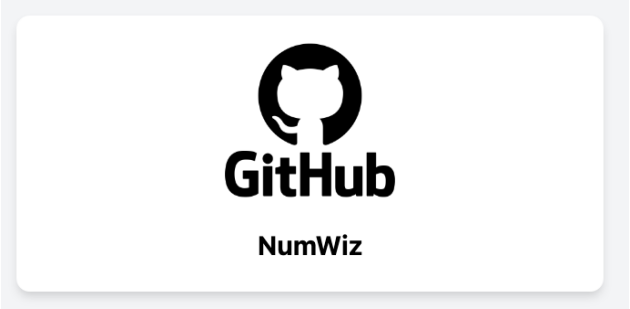


Figure 4: Card della repository

Come si vede in Figura 4, ogni repository analizzato viene rappresentato da una card contenente il nome del repository.

7.4 Dettagli dell'Analisi

Cliccando su una card dei risultati, si aprirà una nuova pagina con i dettagli dell'analisi per quel specifico repository.

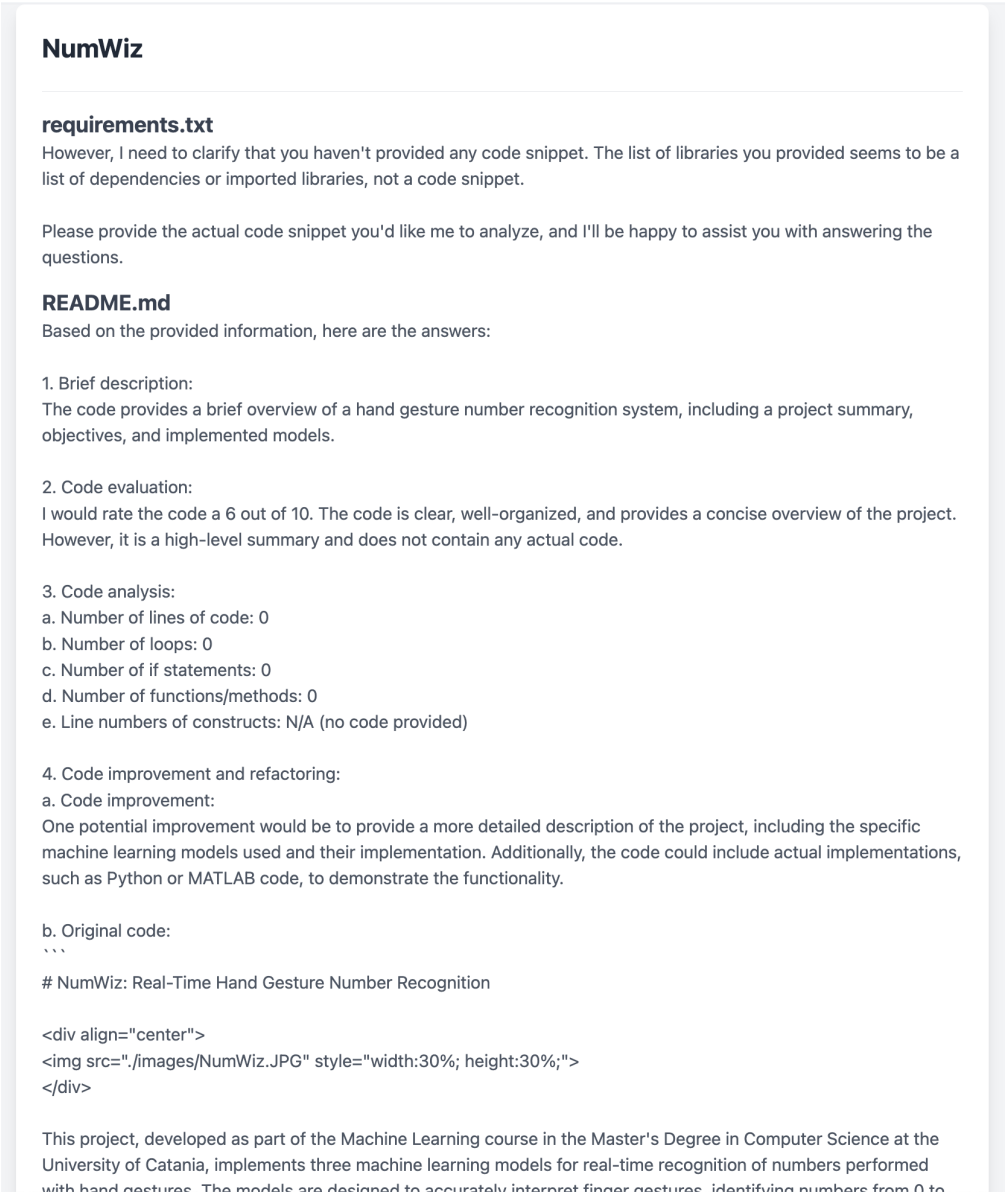


Figure 5: Dettagli dell'analisi di un repository

La Figura 5 mostra un esempio di pagina di dettaglio, dove per ogni file analizzato vengono fornite:

- Una breve descrizione del codice
- Una valutazione complessiva del codice (da 1 a 10)
- Eventuali suggerimenti per il refactoring

8 Bibliografia

References

[1] Flask. Flask. https://flask.palletsprojects.com/.

[2] RabbitMQ. Rabbitmq, 2020. https://www.rabbitmq.com/.

[3] Groq, . https://github.com/groq.

[4] Groq. Llama 3, . https://wow.groq.com/introducing-llama-3-groq-tool-use-models/.