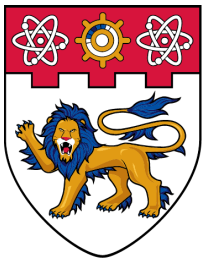


NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

GitLLM

SC4052 Cloud Computing Assignment 2

Choice 2: Github API and Code Search

Academic Year 2024-2025 Semester 2

Demonstration Video: GitLLM

GitHub

Name	Matriculation Number
Lee Alessandro	U2120619H

Table of Contents

Introduction

This report presents **GitLLM**, an innovative Software as a Service (SaaS) application that combines the power of Large Language Models (LLMs) with GitHub's extensive code repositories to create a more intuitive and effective code discovery experience.

GitHub's native search capabilities, while powerful, require users to understand specific search syntax and qualifiers to effectively find relevant code. This creates a barrier for developers who may know what they're looking for conceptually but struggle to express it within GitHub's structured query constraints. Additionally, even when relevant code is found, developers must manually analyze and compare implementations to extract meaningful insights and best practices.

GitLLM addresses these challenges through two core functionalities - Natural Language GitHub Search and Code Pattern Analyzer.

This report details the technical implementation of GitLLM, including its API integration approach, the prompt engineering techniques employed, and the methods used to transform between unstructured and structured data formats. It also examines current limitations of the system and explores potential future enhancements to further improve code discovery and analysis processes for developers.

Feature 1: Natural Language GitHub Search

Figure 1: Data Flow Diagram for Natural Language GitHub Search

Motivation

The natural language search feature was born from a frustration faced by developers: the rigid syntax and complex qualifiers required by GitHub's native search functionality. While GitHub's search API is powerful, it requires users to understand specific query syntax, qualifiers and boolean operators to perform effective searches. This is a significant learning curve especially for developers new to GitHub APIs.

Innovation

The idea behind this feature was to create a more accessible interface where developers can simply describe what they are looking for in plain English. Instead of requiring them to remember syntax like *`language:typescript path:src/components NOT path:test`*, they can simply type queries like *`Show me a code that configures TypeORM to interact with the database in NestJS`*. This removes the technical barrier of search syntax knowledge, allowing developers to focus on finding the code they need rather than how to properly formulate a search query. This feature allows users to do **Code Search, Repository Search, User Search** in plain english.

Core Workflow

Figure 1 depicts the core workflow of this feature:

1. **Request query translation:** Next.js receives a natural language search query from the user.

   The user's query is embedded within a carefully crafted prompt sent to the Gemini LLM (Model: Gemini-1.5-flash), where it analyzes the query's intent and constructs an appropriate GitHub Search API query string.
2. **Structured Search Query:** Based on the LLM's analysis, it returns a structured search query and searchTarget (code/repositories/none) to the backend.
3. **Execute Search**: Based on the searchTarget receives from the LLM, the backend will call different endpoints:
    a. **(Code Search) searchTarget `code` calls /search/code endpoint:** Used when searching for specific code snippets, functions, or examples within files
    b. **(Repository Search) searchTarget `repositories` calls /search/repositories endpoint:** Used when searching for entire repositories/projects
    c. **(User Search) searchTarget `none`:** Used when the query cannot be translated to either code or repository search (e.g., asking for users or issues)
4. **Raw Search Results:** The raw results from the GitHub API are processed and displayed in the frontend.
    a. For code searches, this involves an additional step of fetching the actual file content for a limited number of results to provide snippets by calling the endpoints
        i. **/repos/{owner}/{repo}/contents/{path} endpoint:** Used to fetch the actual file content
        ii. **/repos/{owner}/{repo} endpoint:** Used to fetch repository metadata (stars, forks, etc.)

Prompt Engineering

```
const prompt = `
**Objective:** Transform a natural language query into an optimized GitHub Search API query string.

**Context:** GitHub provides two primary search APIs relevant here:
*   \`/search/code\`: Finds code snippets within files.
*   \`/search/repositories\`: Finds repositories.

**Your Task:**
1.  **Analyze Intent:** Determine if the user wants to find specific code ('code' target) or repositories ('repositories' target). Default to 'code' unless the query strongly implies searching for repositories (e.g., mentions "repository", "project",
or is a single word likely a username).
2.  **Construct Query String:** Build the GitHub search query string based on the analyzed intent.
    *   **Keywords:** Extract the core search terms.
    *   **Qualifiers:** Intelligently apply relevant GitHub search qualifiers based on the natural language. Key qualifiers include:
        *   \`repo:owner/name\`: Use ONLY if a specific repository is clearly mentioned.
        *   \`user:username\`: Use for queries targeting a specific user's code/repos.
        *   \`language:lang\`: Use if a programming language is specified.
        *   \`path:path/to/dir\`: Use if a specific directory or path is mentioned.
        *   \`extension:ext\`: Use if a file extension is mentioned.
        *   \`in:file,path\`: Use if the query specifies searching within file contents or paths.
        *   Numeric/Date: \`stars:>\`, \`forks:>\`, \`size:>\`, \`created:>\`, \`pushed:>\`.
        *   Boolean: \`fork:true/only\`, \`archived:false\`.
    *   **Logical Operators:** Interpret "and", "or", "not" (and similar terms) using GitHub's \`AND\`, \`OR\`, \`NOT\` operators. \`AND\` is often implicit.
    *   **Syntax Rules:**
        *   Keywords first, then qualifiers (e.g., \`"database connection" repo:expressjs/express\`).
        *   Qualifiers format: \`qualifier:value\`.
        *   Separate all terms and qualifiers with spaces.
        *   If \`language:\` is used, do not repeat the language in the keywords.
        *   The final string MUST NOT include the \`q=\` prefix.
3.  **Handle Ambiguity/Unsupported:**
    *   If the query is clearly unsupported (e.g., searching issues, users directly), set \`searchTarget\` to \`"none"\`.
    *   If the query is a single word likely a username, default to a repository search: \`searchTarget: "repositories"\`, \`githubQueryString: "user:username"\`.

**Output Format:** Respond ONLY with a valid JSON object matching this exact structure:
\`\`\`json
{
  "searchPlan": {
    "searchTarget": "repositories" | "code" | "none",
    "githubQueryString": "The constructed query string (keywords + qualifiers)",
    "queryAssessment": "A brief evaluation of how well the query maps to GitHub search capabilities.",
    "constructionRationale": "Concise explanation of the chosen target, qualifiers, and any assumptions.",
    "inferredIntent": "A short summary of what the user is likely trying to achieve."
  }
}
\`\`\`

**Example:**
Natural language query: "Find express database connection examples not in tests"
Expected JSON Output:
\`\`\`json
{
  "searchPlan": {
    "searchTarget": "code",
    "githubQueryString": "\\"express database connection examples\\" NOT path:test NOT path:tests",
    "queryAssessment": "Good mapping to code search with exclusion.",
    "constructionRationale": "Identified keywords 'express database connection examples'. Inferred exclusion of test directories using 'NOT path:'. Target is 'code' as user asks for examples.",
    "inferredIntent": "Find code examples for Express database connections, excluding test files."
  }
}
\`\`\`

**User's Natural language query:** ${query}
`;
```

Figure 2: The Prompt located in /frontend/src/app/api/search/route.ts (Also in Appendix)

The Role of the LLM

As depicted in Figure 2/Appendix, the LLM is tasked via a detailed prompt to:

a. **Analyze Intent**: Determine if the user wants code examples or entire repositories.
b. **Extract Keywords**: Identify the core terms of the search.
c. **Apply Qualifiers**: Intelligently add GitHub search qualifiers (like language:, user:, repo:, path:, NOT, etc.) based on the natural language input. For example, “in python” becomes language:python, “by user X” becomes user:X, “excluding tests” becomes NOT path:test NOT path:tests.
d. **Construct Query String**: Build the final query string (without the q= prefix) according to GitHub’s syntax rules.
e. **Assess Query**: Provide a brief assessment of how well the query maps to GitHub’s capabilities.
f. **Explain Rationale**: Justify the chosen target and constructed query.

g. **Infer Intent**: Summarize what the user is likely trying to find.

Key GitHub Qualifiers Utilized

The prompt guides the LLM to use various GitHub search qualifiers, including:

a. **repo:owner/name:** For searching within a specific repository.
b. **user:username:** For searching code/repos owned by a specific user.
c. **language:lang:** To filter by programming language.
d. **path:path/to/dir:** To search within specific directories or file paths.
e. **extension:ext:** To filter by file extension.
f. **in:file,path:** To specify searching within file contents or paths.
g. **Numeric/Date Qualifiers:** stars:, forks:, size:, created:, pushed: (often with >, <, or ranges).
h. **Boolean Qualifiers:** fork:true/only, archived:false.
i. **Logical Operators:** AND (often implicit), OR, NOT.

LLM Output Structure

This structured response in Figure 3 allows the backend to reliably extract the necessary information (searchTarget, githubQueryString) to proceed in calling the respective GitHub APIs. Furthermore, the githubQueryString and constructionRationale are also used to display it on the frontend.

```
interface SearchPlanResponse {
  searchPlan: {
    searchTarget: 'repositories' | 'code' | 'none'; // Target API endpoint
    githubQueryString: string; // Query string for GitHub API (without q=)
    queryAssessment: string; // Brief assessment of query suitability
    constructionRationale: string; // Explanation of how the query was built
    inferredIntent: string; // The user's likely goal
  };
}
```

Figure 3: LLM Output Structure for Natural Language GitHub Search Feature

Query Examples

a. **Code Search (Figure 5):** `Show me a code that configures TypeORM to interact with the database in NestJS`

b. **User Search (Figure 6):** `Help me find user visionmedia`

c. **Repository Search (Figure 7):** `Find me devops-exercises repository by bregman-arie`

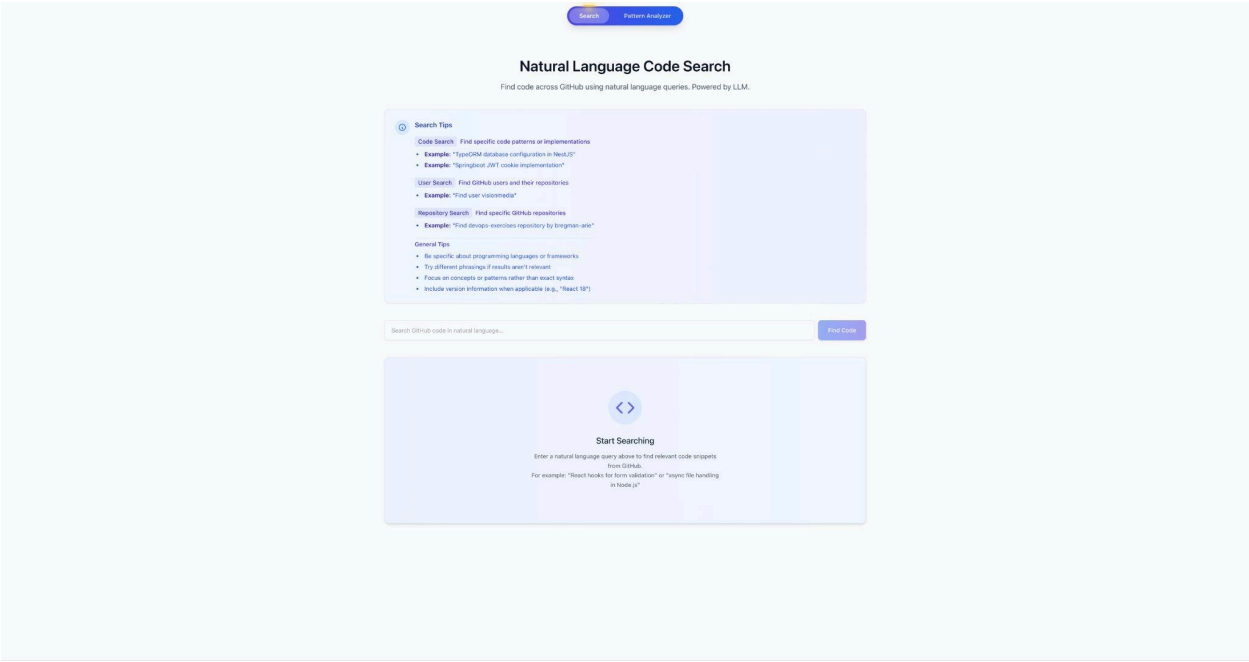


Figure 4: GitLLM - Natural Language Code Search Tab

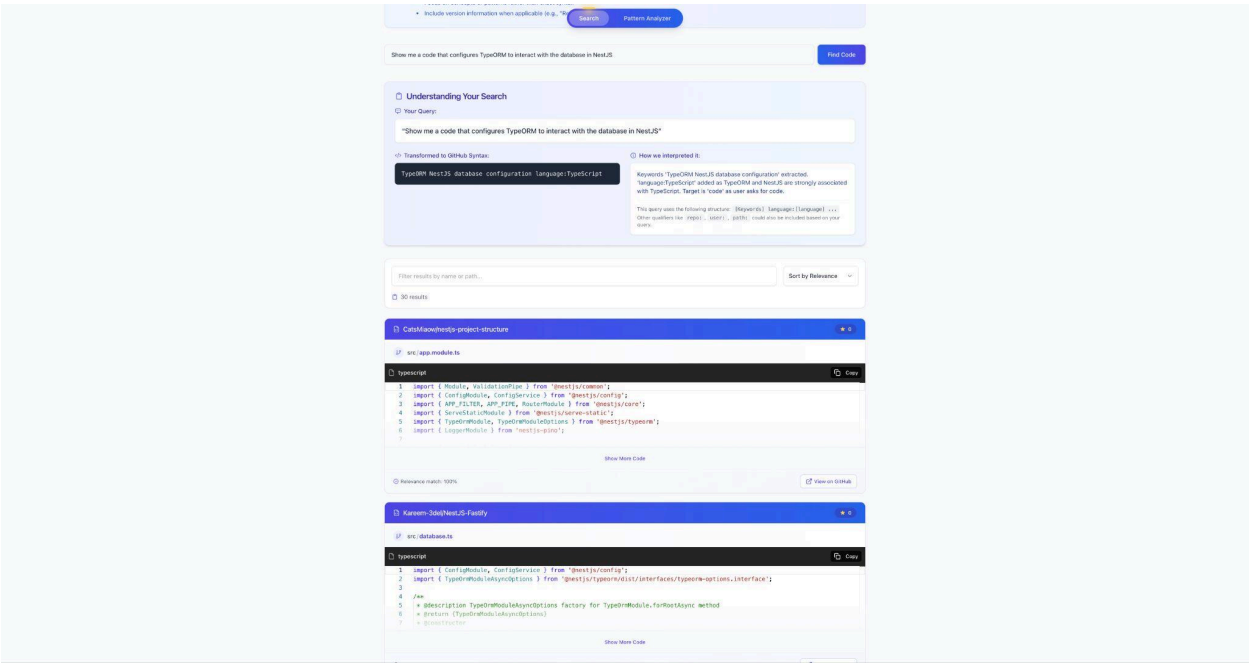


Figure 5: GitLLM - Code Search

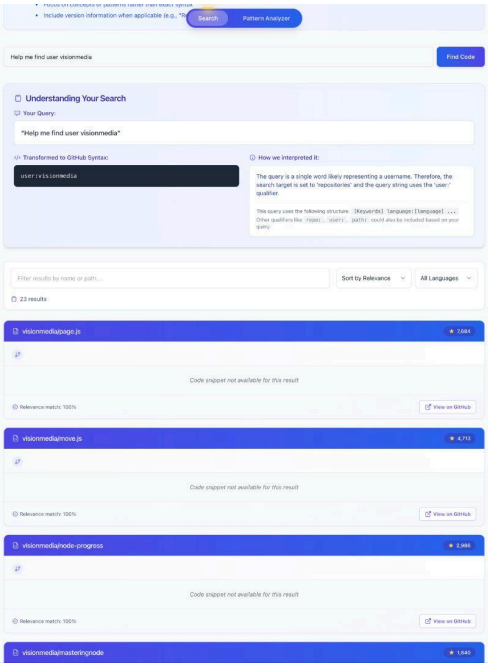


Figure 6: GitLLM - User Search

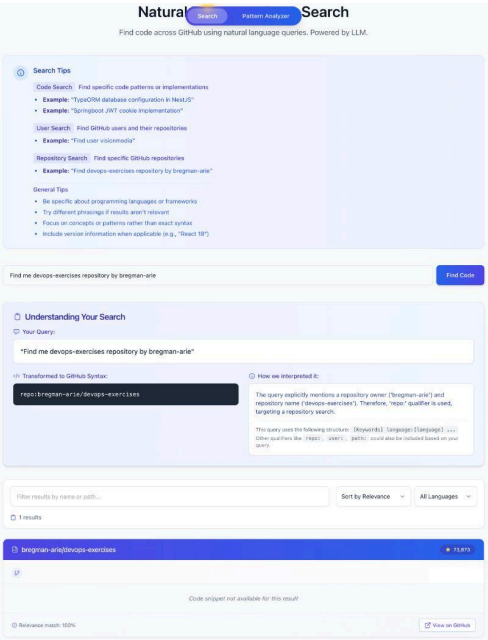


Figure 7: GitLLM - Repository Search

Feature 2: Code Pattern Analyzer

Figure 8: Data Flow Diagram for Code Pattern Analyzer

Motivation

The code pattern analyzer feature was developed out of a personal need faced during development work. As developers, we often refer to documentation which provides basic, barebones templates for implementation, but this is frequently insufficient for real-world applications. Documentation examples tend to be simplified, stripped of error handling, edge cases, and production-ready practices that are crucial in actual development. While implementing complex features or integrating with third-party services, I found myself constantly searching GitHub for how other developers had approached similar problems. This was a tedious process - finding popular repositories, examining code snippets, and comparing different implementation approaches without any structured analysis.

Innovation

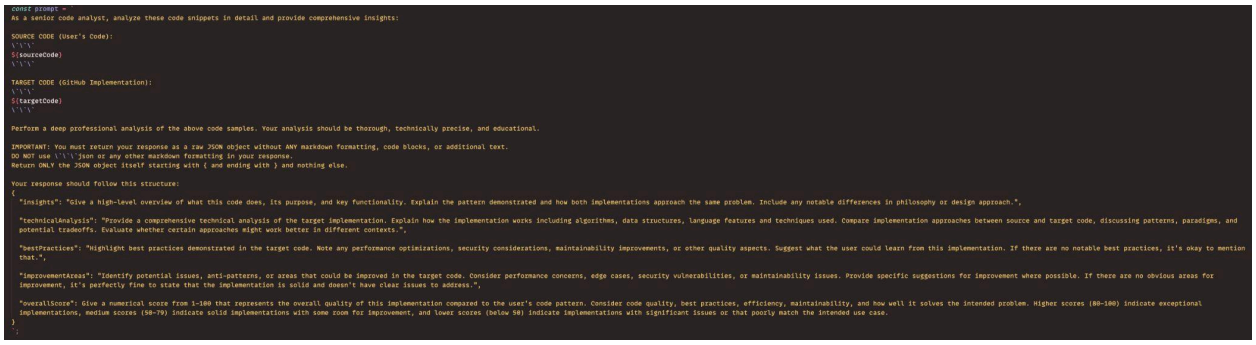
This feature addresses this gap by allowing developers to submit their own implementation (or a template from documentation) and discover how others in the GitHub community have solved the same problem. By leveraging LLM's code understanding capabilities, the system provides not just similar code examples, but detailed technical analysis highlighting differences in approach, best practices, optimization opportunities and potential issues. This tool transforms the common developer practice of "how do others implement this?" from a manual process into a structured, automated analysis that accelerates learning and improves code quality.

Core WorkFlow

Figure 8 depicts the core workflow of this feature:

1. **Code Pattern Search Query:** Next.js receives a code pattern snippet from the user, along with optional filter parameters (repository and user filters).
   Using the pattern as a search query, the backend searches for similar code implementations on GitHub by calling the **/search/code endpoint**.
2. **Raw Search Results:** The raw results from the GitHub API returns a list of code files matching the search criteria (limited to relevant files containing similar patterns).
   [Arrows not shown in Figure 8] For each search result (up to 10 matches to avoid excessive API calls), Next.js requests the complete file content and repository details by calling the endpoints
   a. **/repos/{owner}/{repo}/contents/{path} endpoint:** Used to fetch the actual file content
   b. **/repos/{owner}/{repo} endpoint:** Used to fetch repository metadata (stars, forks, etc.)
3. **Request Code Analysis:** For each matching implementation, the LLM performs a deep comparison between the user's code pattern snippet and the discovered implementation.
4. **Structured Analysis:** The LLM generates detailed insights, technical analysis, best practices, and improvement areas, providing a comprehensive comparison. The results are formatted, including all analyses and repository metadata, and sent back to the frontend for display.

Prompt Engineering



```
const prompt =
As a senior code analyst, analyze these code snippets in detail and provide comprehensive insights:

SOURCE CODE (User's Code):
\`\`\`
${sourceCode}
\`\`\`

TARGET CODE (GitHub Implementation):
\`\`\`
${targetCode}
\`\`\`

Perform a deep professional analysis of the above code samples. Your analysis should be thorough, technically precise, and educational.

IMPORTANT: You must return your response as a raw JSON object without ANY markdown formatting, code blocks, or additional text.
DO NOT use \`\`\`json or any other markdown formatting in your response.
Return ONLY the JSON object itself starting with { and ending with } and nothing else.

Your response should follow this structure:
{
  "insights": "Give a high-level overview of what this code does, its purpose, and key functionality. Explain the pattern demonstrated and how both implementations approach the same problem. Include any notable differences in philosophy or design approach.",

  "technicalAnalysis": "Provide a comprehensive technical analysis of the target implementation. Explain how the implementation works including algorithms, data structures, language features and techniques used. Compare implementation approaches between source and target code, discussing patterns, paradigms, and
  potential tradeoffs. Evaluate whether certain approaches might work better in different contexts.",

  "bestPractices": "Highlight best practices demonstrated in the target code. Note any performance optimizations, security considerations, maintainability improvements, or other quality aspects. Suggest what the user could learn from this implementation. If there are no notable best practices, it's okay to mention
  that.",

  "improvementAreas": "Identify potential issues, anti-patterns, or areas that could be improved in the target code. Consider performance concerns, edge cases, security vulnerabilities, or maintainability issues. Provide specific suggestions for improvement where possible. If there are no obvious areas for
  improvement, it's perfectly fine to state that the implementation is solid and doesn't have clear issues to address.",

  "overallScore": Give a numerical score from 1-100 that represents the overall quality of this implementation compared to the user's code pattern. Consider code quality, best practices, efficiency, maintainability, and how well it solves the intended problem. Higher scores (80-100) indicate exceptional
  implementations, medium scores (50-79) indicate solid implementations with some room for improvement, and lower scores (below 50) indicate implementations with significant issues or that poorly match the intended use case.
}
```

Figure 9: The Prompt located in /frontend/src/app/api/pattern-analyzer/route.ts (Also in Appendix)

The Role of the LLM

As depicted in Figure 9/Appendix, the LLM is tasked via a detailed prompt to:

a. **Compare Implementations**: Analyze both the user's code and the GitHub implementation to understand their approaches to the same problem.
b. **Extract Insights**: Provide a high-level overview of what the code does, its purpose, and key functionality.
c. **Perform Technical Analysis**: Delve into implementation details, examining algorithms, data structures, and language features used.
d. **Identify Best Practices**: Highlight quality aspects like performance optimizations, security considerations, and maintainability improvements.
e. **Suggest Improvements**: Identify potential issues, anti-patterns, or areas for improvement in the implementation.
f. **Score Implementation**: Provide a numerical quality score (1-100) that represents the overall quality compared to the user's pattern.

LLM Output Structure

The structured response in Figure 10 allows the frontend to present the analysis in organized, digestible sections. An overall score is also given to allow users to get a brief overview of which are the better or worse implementations.

```typescript
interface PatternAnalysisResult {
  insights: string; // High-level overview of the code purpose and pattern
  technicalAnalysis: string; // Detailed technical breakdown of implementation
  bestPractices: string; // Quality aspects and learnings from the implementation
  improvementAreas: string; // Potential issues and suggestions for improvement
  overallScore: number; // 1-100 quality score for the implementation
}
```

Figure 10: LLM Output Structure for Code Pattern Analyzer Feature

Query Examples

RolesGuard Code Snippet Retrieved from NestJS documentation was used as code input (Figure 12).

```typescript
@Injectable()
export class RolesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const roles = this.reflector.get(Roles, context.getHandler());
    if (!roles) {
      return true;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    return matchRoles(roles, user.roles);
  }
}
```

Figure 11: RolesGuard Code Snippet Retrieved from NestJS Documentation

Users can input their code snippet on the code editor, and configure the language from the drop-down in the editor. They can further refine their search by including the repository and/or the user name in the filters.

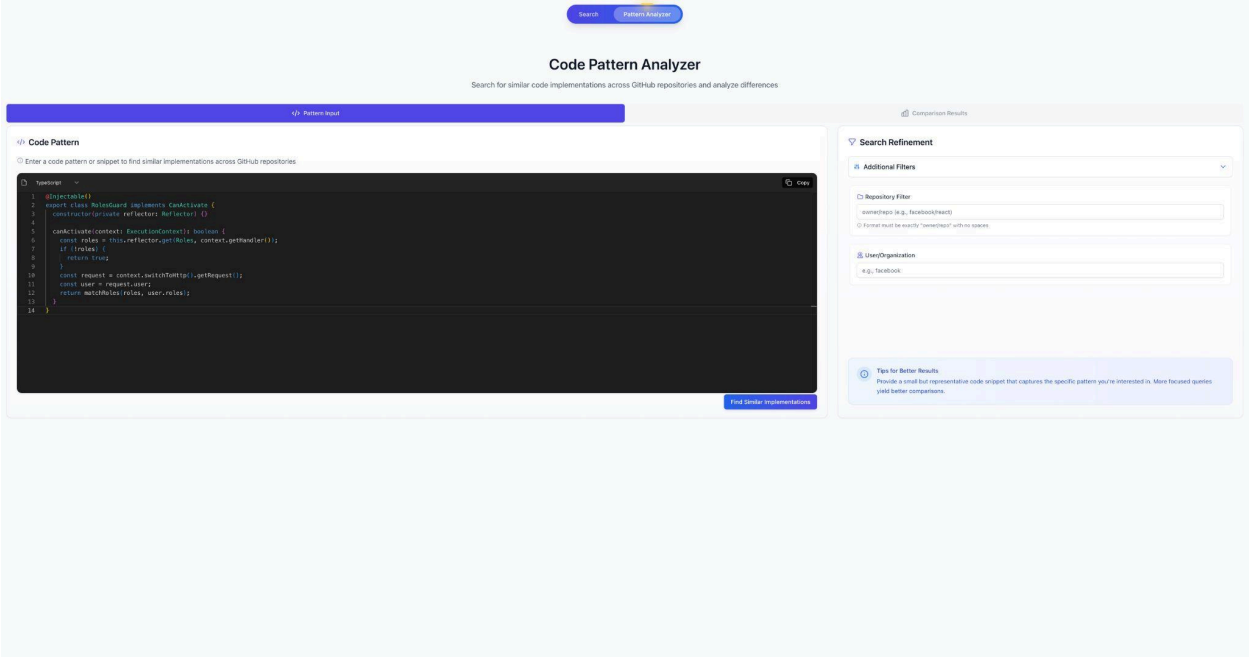


Figure 12: RolesGuard Code Snippet used as input in the code editor (with no filters selected)

After clicking the `Find Similar Implementations` button, we are able to view the various results retrieved on the left hand side (LHS) column `Similar Implementations`, with repository metadata details and overall score given by the LLM displayed as a score badge. Users can toggle between the similar implementations found to switch their view.

In the `Side-by-Side Comparison` Sub-tab (Figure 13), we can see our input code on the LHS code editor and the implementation code on the right hand side (RHS) coder editor. This allows us to view the differences and similarities in our input code and the implementation code found.

We are also able to view the repository details at the bottom such as the stars, forks, etc., as well as clicking on the `View on GitHub` button to view the repository. Users can also click on the other `View on GitHub` button located on the LHS `Similar Implementations` column to view the specific file selected.

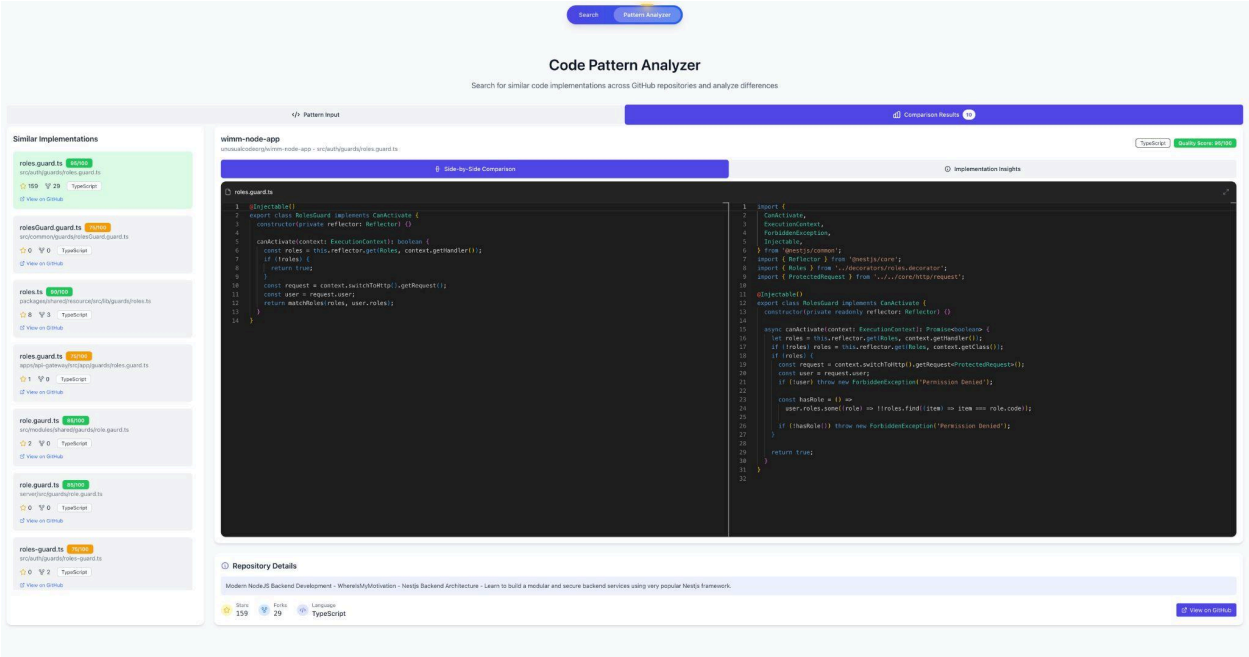


Figure 13: Side-by-Side Comparison Sub-tab

In the `Implementation Insights` Sub-tab (Figure 14), we can see the detailed code analysis such as overview, technical analysis, best practices, and improvement areas.

This allows users to learn from the code implementation found on GitHub through the LLM insights.

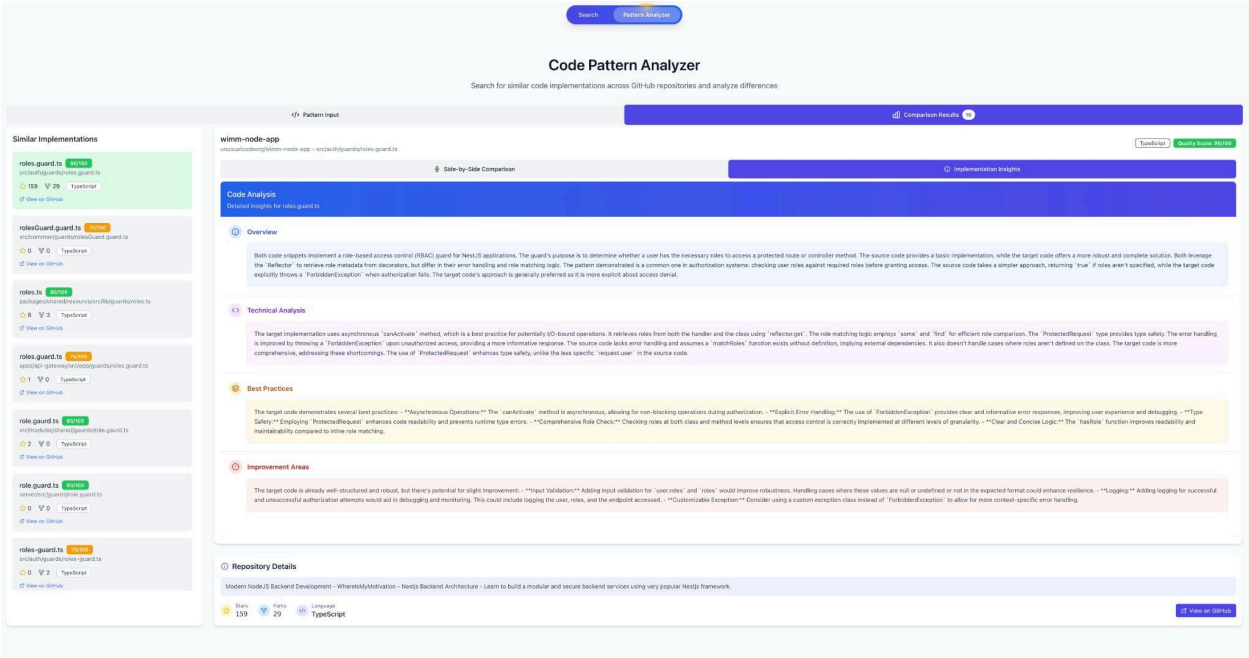


Figure 14: Implementation Insights Sub-tab

Prompt Engineering Refinements

Several optimization strategies and prompt refinement techniques were employed to improve LLM performance. Multiple prompt versions were tested to find the most optimal instructions. This was a tedious yet rewarding process.

Detailed Role Description

Positioning the LLM as a domain specific expert:

In Feature 1: Natural Language GitHub Search

"Objective: Transform a natural language query into an optimized GitHub Search API query string."

In Feature 2: Code Pattern Analyzer

"As a senior code analyst, analyze these code snippets in detail and provide comprehensive insights:"

Knowledge Injection

Providing GitHub-specific search syntax rules:

"Keywords first, then qualifiers (e.g., \"database connection" repo:expressjs/express\")"...

Few-Shot Learning

Including an example of expected transformation:

"Natural language query: "Find express database connection examples not in tests\". Expected JSON Output: {...}"

Output Formatting

Strict specification of response format:

"Output Format: Respond ONLY with a valid JSON object matching this exact structure: {...}"

LLM Selection: Gemini-1.5-flash

The application utilizes Google's Gemini-1.5-Flash model for both the search query translation and code pattern analysis features. The model choice was deliberate and strategic for several reasons.

Cost Efficiency

Free Tier Access: Gemini-1.5-Flash is available through Google's generous free tier allocation in the Generative AI API.
No Credit Card Requirement: Unlike some competing models (such as OpenAI and Anthropic), Gemini API models can be accessed with just a Google account.

Performance for Prompt-Driven Tasks

Sufficient Capability: For well-engineered prompts with clear instructions and structure, Gemini-1.5-Flash provides comparable quality to newer models like Gemini-2.0-Flash.
Prompt Engineering vs. Model Power: Our implementation demonstrates that sophisticated prompt design can compensate for using an older/smaller model.
Structured Output Control: The carefully crafted prompts with detailed JSON response formats ensure consistent, high-quality outputs despite using a less expensive model.

Technical Considerations

Response Speed: Gemini Flash variants (compared to Gemini Pro variants) prioritize low-latency responses which is crucial for an interactive web application.
Model Context Window: Gemini-1.5-Flash comes with a 1-million-token context window [1], which far exceeds the market standard of competitors like OpenAI and Anthropic, and more than sufficient for our use cases.

Limitations and Challenges

Despite the effectiveness of the solution, several limitations and challenges affect the system:

GitHub API Limitations

1. **Query Length Restrictions:**
    - GitHub's search API limits query strings to approximately 256 characters.
    - For complex patterns, the system must truncate to first 200 characters, potentially losing important context.
2. **Search Precision:**
    - GitHub's code search is not optimized for semantic understanding.
    - Results may include false positives despite LLM query refinement.
3. **Content Access Restrictions:**
    - The system cannot access private repositories without specific user authorization.
    - Some content may be inaccessible due to repository permissions.

LLM Limitations

1. **Consistency Challenges:**
    - LLM responses may occasionally vary in structure despite strict output formatting.
    - Thus the system must handle potential inconsistencies in LLM output by introducing fallback mechanisms.
2. **Non-deterministic Output:**
    - LLMs can produce different responses to the same input across multiple runs, even when given strict instructions.
    - For Feature 2: Code Pattern Analyzer (Figure 14), the score rating shown in `Similar Implementations` column and implementation insights shown in `Implementation Insights` Sub-tab may vary for identical code samples across different sessions.

3. **Model Context Window:**
    - Gemini-1.5-Flash comes with a 1-million-token context window [1], this context window is more than sufficient for our use case.

Conclusion

GitLLM represents a significant advancement in how developers can interact with GitHub's vast code repositories. By leveraging LLMs to bridge the gap between natural language and API interfaces, the system provides capabilities beyond what either GitHub's native search or traditional code search tools can offer.

The natural language search function transforms how developers find code examples and repositories, making GitHub's extensive resources more accessible through intuitive queries. Meanwhile, the pattern analyzer function provides valuable insights into code implementation alternatives, enabling developers to learn from community practices and improve their own code.

The project demonstrates several important principles in LLM application development:

1. **Effective Prompt Engineering**: Carefully crafted prompts enable LLMs to consistently produce structured, useful outputs.
2. **API Integration Synergy**: Combining LLM capabilities with traditional APIs creates functionalities greater than either could provide individually.
3. **Structured Information Extraction**: Converting free-form LLM outputs into structured formats enables programmatic processing.
4. **Graceful Error Handling**: Managing the uncertainties of LLM responses requires robust error handling and fallback strategies.

Future development could address current limitations through:

1. **User Feedback Loop**: Incorporating user feedback to improve search and analysis quality, and storing conversation history to further improve the LLM feedback quality.

GitLLM demonstrates the potential of using LLMs as intelligent intermediaries between users and complex APIs, creating more intuitive and powerful developer tools. This approach could be extended to other development resources and tools, further improving developer productivity and learning through AI assistance.

Reference

[1] “Long context | Gemini API | google AI for developers,” Google, https://ai.google.dev/gemini-api/docs/long-context (accessed Apr. 17, 2025).

Appendix

Prompt for Feature 1: Natural Language GitHub Search

const prompt = `

Objective: Transform a natural language query into an optimized GitHub Search API query string.

Context: GitHub provides two primary search APIs relevant here:

* \`/search/code\`: Finds code snippets within files.

* \`/search/repositories\`: Finds repositories.

Your Task:

1. **Analyze Intent:** Determine if the user wants to find specific code ('code' target) or repositories ('repositories' target). Default to 'code' unless the query strongly implies searching for repositories (e.g., mentions "repository", "project", or is a single word likely a username).

2. **Construct Query String:** Build the GitHub search query string based on the analyzed intent.

    * **Keywords:** Extract the core search terms.

    * **Qualifiers:** Intelligently apply relevant GitHub search qualifiers based on the natural language. Key qualifiers include:

        * \`repo:owner/name\`: Use ONLY if a specific repository is clearly mentioned.

        * \`user:username\`: Use for queries targeting a specific user's code/repos.

        * \`language:lang\`: Use if a programming language is specified.

        * \`path:path/to/dir\`: Use if a specific directory or path is mentioned.

        * \`extension:ext\`: Use if a file extension is mentioned.

        * \`in:file,path\`: Use if the query specifies searching within file contents or paths.

        * Numeric/Date: \`stars:>\`, \`forks:>\`, \`size:>\`, \`created:>\`, \`pushed:>\`.

        * Boolean: \`fork:true/only\`, \`archived:false\`.

* **Logical Operators:** Interpret "and", "or", "not" (and similar terms) using GitHub's \`AND\`, \`OR\`, \`NOT\` operators. \`AND\` is often implicit.

* **Syntax Rules:**

  * Keywords first, then qualifiers (e.g., \`"database connection" repo:expressjs/express\`).

  * Qualifiers format: \`qualifier:value\`.

  * Separate all terms and qualifiers with spaces.

  * If \`language:\` is used, do not repeat the language in the keywords.

  * The final string MUST NOT include the \`q=\` prefix.

3. **Handle Ambiguity/Unsupported:**

* If the query is clearly unsupported (e.g., searching issues, users directly), set \`searchTarget\` to \`"none"\`.

* If the query is a single word likely a username, default to a repository search: \`searchTarget: "repositories"\`, \`githubQueryString: "user:username"\`.

Output Format: Respond ONLY with a valid JSON object matching this exact structure:

\`\`\`json

{

"searchPlan": {

"searchTarget": "repositories" | "code" | "none",

"githubQueryString": "The constructed query string (keywords + qualifiers)",

"queryAssessment": "A brief evaluation of how well the query maps to GitHub search capabilities.",

"constructionRationale": "Concise explanation of the chosen target, qualifiers, and any assumptions.",

"inferredIntent": "A short summary of what the user is likely trying to achieve."

}

}

\`\`\`

Example:

Natural language query: "Find express database connection examples not in tests"

Expected JSON Output:

\`\`\`json

{

"searchPlan": {

"searchTarget": "code",

"githubQueryString": "\\"express database connection examples\\" NOT path:test NOT path:tests",

"queryAssessment": "Good mapping to code search with exclusion.",

"constructionRationale": "Identified keywords 'express database connection examples'. Inferred exclusion of test directories using 'NOT path:'. Target is 'code' as user asks for examples.",

"inferredIntent": "Find code examples for Express database connections, excluding test files."

}

}

\`\`\`

User's Natural language query: ${query}

`;

Prompt for Feature 2: Code Pattern Analyzer

const prompt = `

As a senior code analyst, analyze these code snippets in detail and provide comprehensive insights:

SOURCE CODE (User's Code):

\`\`\`

${sourceCode}

\`\`\`

TARGET CODE (GitHub Implementation):

\`\`\`

${targetCode}

\`\`\`

Perform a deep professional analysis of the above code samples. Your analysis should be thorough, technically precise, and educational.

IMPORTANT: You must return your response as a raw JSON object without ANY markdown formatting, code blocks, or additional text.

DO NOT use \`\`\`json or any other markdown formatting in your response.

Return ONLY the JSON object itself starting with { and ending with } and nothing else.

Your response should follow this structure:

{

"insights": "Give a high-level overview of what this code does, its purpose, and key functionality. Explain the pattern demonstrated and how both implementations approach the same problem. Include any notable differences in philosophy or design approach.",

"technicalAnalysis": "Provide a comprehensive technical analysis of the target implementation. Explain how the implementation works including algorithms, data structures, language features and techniques used. Compare implementation approaches between source and target code, discussing patterns, paradigms, and potential tradeoffs. Evaluate whether certain approaches might work better in different contexts.",

"bestPractices": "Highlight best practices demonstrated in the target code. Note any performance optimizations, security considerations, maintainability improvements, or other quality aspects. Suggest what the user could learn from this implementation. If there are no notable best practices, it's okay to mention that.",

"improvementAreas": "Identify potential issues, anti-patterns, or areas that could be improved in the target code. Consider performance concerns, edge cases, security vulnerabilities, or maintainability issues. Provide specific suggestions for improvement where possible. If there are no obvious areas for improvement, it's perfectly fine to state that the implementation is solid and doesn't have clear issues to address.",

"overallScore": Give a numerical score from 1-100 that represents the overall quality of this implementation compared to the user's code pattern. Consider code quality, best practices, efficiency, maintainability, and how well it solves the intended problem. Higher scores (80-100) indicate exceptional implementations, medium scores (50-79) indicate solid implementations with some room for improvement, and lower scores (below 50) indicate implementations with significant issues or that poorly match the intended use case.

}

`;