CodeCommenter.AI: Context-Aware Comment Generation for Game Development Scripts

Github repo : https://github.com/AnthonyBibbo/csce689-final-proj-CodeCommenter.AI

1. Introduction

As having been an intern for gaming companies in the past I have learned that large scale game development projects greatly rely on well documented code. And I have seen the recurring issue where developers lack or deprioritize good documentation. This is where I got the inspiration to integrate LLMs into a solution that could solve this problem.

The main goal of this project is to build a tool, CodeCommenter.AI, that would automatically generate high quality, context rich comments using api calls to a Large Language Model. This tool would be beneficial for and tested with popular game engine scripts, from Unity and Unreal.

For my tool, I evaluate its performance and effectiveness with the help of automatic metrics and several different prompting strategies. I will explore all of these methods and variables deeper throughout my report. There is a setup and usage section at the end of this document to help with getting started and trying it out yourself.

---

2. Project Objectives

The project aims to:

- Develop a working prototype that generates descriptive comments for game scripting functions.
- Experiment with different prompting strategies for LLM inference.
- Evaluate generated comments against real developer-written comments using automatic metrics (BLEU, CodeBLEU).
- Analyze the impact of example-based prompting and domain-specific instructions.
- Produce a reusable dataset, experiments, and analysis demonstrating real-world applicability.

3. Methodology

The system is designed as a **code-to-comment generation pipeline** composed of:

1. **Input Processing**
   Raw Unity/Unreal source code files are parsed into function-level chunks.
2. **LLM Stage**
   A large language model generates a short descriptive comment for each function using one of four prompting strategies.
3. **Output Reconstruction**
   Generated comments are inserted directly above the function definitions and written to a new annotated file.
4. **Evaluation**
   The generated comments are compared to ground-truth comments using BLEU and CodeBLEU metrics.

3.1 Data Collection

To obtain real game-development comments for evaluation, I downloaded the open-source Unity project **Open Project: Chop Chop** by Unity Technologies.
From this project:

- All .cs files under Assets/Scripts/ were collected.
- A custom dataset extraction script parsed over **105 functions** with valid human-written comments.
- Each dataset entry contains:
    - Function name
    - Function code block
    - Original developer-written comment
    - File path and line numbers

This dataset served as both evaluation reference and source of few-shot prompt examples.

3.2 Parsing and Chunking

A function parser was implemented to:

- Identify function signatures using heuristics (e.g., presence of parentheses and braces)
- Collect the full function body using brace matching
- Extract preceding comment blocks (//, ///, or /* */)
- Store everything as structured JSON

This parsing step ensures LLM prompts focus only on meaningful, self-contained units of code.

3.3 Prompting Strategies

To understand how prompting affects LLM performance, four inference modes were tested:

1. **Zero-shot**
    - The model receives only raw function code.
    - Baseline with no examples or domain hints.
2. **Few-shot**
    - The model receives three real example code–comment pairs.
    - Intended to teach style and tone directly from the dataset.
3. **Domain-aware**
    - The model receives Unity-specific contextual instructions (e.g., explain Update(), physics callbacks).
    - No examples included.
4. **Few-shot + Domain-aware**
    - Combination of both examples and engine-specific guidance.

3.4 Experimental Design

- **Sample Size:** 25 randomly selected annotated functions from the dataset.
- **Model Used:** gpt-4o-mini via OpenAI API.
- **Metrics:**
    - **BLEU:** n-gram similarity between generated and ground-truth comments.
    - **CodeBLEU:** syntax-weighted variant designed for code summarization.

The experiments output JSONL files containing both the generated comments and ground-truth comments, allowing reproducible evaluation.

4. Results

4.1 Automatic Metrics

Mode	Avg BLEU	Avg CodeBLEU
Zero-shot	**1.86**	**0.297**
Few-shot	**8.37**	**0.305**
Domain-aware	**1.66**	**0.230**
Few-shot + Domain-aware	**5.37**	**0.266**

4.2 Discussion of Results

Few-shot prompting achieved the best results across both metrics, outperforming all other modes. BLEU increased from 1.86 to 8.37, and CodeBLEU increased from 0.297 to 0.305. This indicates that LLMs benefit significantly from seeing actual Unity code comments, which teach the preferred structure, tone, and level of detail.

Zero-shot and domain-aware prompting were least effective.
Zero-shot results demonstrate that without examples, the model produces comments that are correct but not stylistically aligned with human-written comments. Domain-aware prompting, surprisingly, performed worse than zero-shot; while domain hints help with correctness, they do not guide tone or phrasing, causing mismatch with ground-truth comments.

Few-shot + domain-aware performed better than zero-shot but worse than pure few-shot.
This suggests that domain instructions may conflict with stylistic examples, making comments more verbose or generic.

I am happy with my findings as they resemble results seen in other code-summarization studies and literature where it was found that
- Short comments = low absolute BLEU scores
- Example-based prompting = largest improvements
- Domain hints help semantic accuracy but not stylistic similarity

5. Qualitative Examples

Below is a real qualitative comparison from the output:

Ground Truth:

Initializes the character's rigidbody reference if it hasn't been assigned.

Zero-shot:

Sets up the rigidbody component for the object.

Few-shot:

Ensures the player's Rigidbody component is assigned before gameplay begins.

Domain-aware:

Prepares the Rigidbody required for Unity physics interactions.

Few-shot + domain-aware:

Makes sure the character has a Rigidbody so physics work properly.

Observation:
Few-shot is the closest match in tone, length, and purpose — explaining why it scored highest.

---

6. Limitations

- **BLEU is harsh on short, single-sentence comments**, often underestimating semantic similarity.
- Dataset derived from a single Unity project; additional games would improve diversity.
- Domain-aware prompting was handcrafted and could be improved with better engine-specific knowledge.
- The system currently supports Unity and Unreal but was only tested on Unity for this project.

7. Conclusion

The results demonstrate that LLM-based comment generation is feasible and effective for Unity game development scripts. The system successfully generates concise, descriptive comments aligned with developer expectations.

The most important finding is that few-shot prompting dramatically improves comment quality, outperforming both baseline and domain-aware models. This shows that lightweight, example-driven prompting can serve as a practical alternative to full-scale fine-tuning for game-specific code summarization.

The CodeCommenter.AI prototype shows strong potential as a workflow tool to reduce technical debt and improve code readability in game development teams.

---

8. Future Work

Several promising extensions remain:

- Fine-tuning a small model (e.g., StarCoder2) on Unity/Unreal-specific datasets.
- Expanding dataset diversity across multiple open-source Unity and Unreal games.
- Developing a VS Code extension to integrate CodeCommenter.AI into real workflows.
- Adding semantic similarity metrics (e.g., BERTScore).

---

9. Setup and Usage Guide

This section provides detailed instructions for setting up the CodeCommenter.AI project and reproducing all experiments included in the report. The steps assume that the user has already cloned the project's GitHub repository.

Requirements: Python Dependencies

All required Python packages are listed in `requirements.txt`.

Install them using:

```
pip3 install -r requirements.txt
```

Setting Your OpenAI API Key

The tool relies on an LLM accessed through the OpenAI API.
Set your API key as an environment variable:

macOS / Linux:

```
export OPENAI_API_KEY="your-api-key-here"
```

Windows PowerShell:

```
$env:OPENAI_API_KEY="your-api-key-here"
```

This must be set before running any scripts that interact with the model.

Preparing the Dataset

To reproduce the dataset used in the project:

1. Place your Unity C# files into:

```
data/raw_unity/
```

```
NOTE: I left the game Chop Chop scripts in
data/raw_unity/ChopChop_scripts for you to use if you'd like
```

2. Run the dataset builder:

```
python3 build_dataset.py --input-dir data/raw_unity --language unity
--output data/datasets/unity_dataset.jsonl
```

This will extract function-level code-comment pairs into a JSONL dataset.

Running the Comment Generator

To test the core comment-generation pipeline on a single script:

```
python3 code_commenter.py data/raw_unity/MyScript.cs --model
gpt-4o-mini
```

This creates an annotated version:

```
MyScript.commented.cs
```

Running Experiments (Zero-shot, Few-shot, Domain)

To reproduce the experimental results described in this report, run each experiment mode below. These commands generate predictions stored as JSONL files.

Zero-shot

```
python3 run_experiments.py --dataset data/datasets/unity_dataset.jsonl
--model gpt-4o-mini --mode zero-shot --output results_zero_shot.jsonl
--max-samples 25
```

Few-shot

```
python3 run_experiments.py --dataset data/datasets/unity_dataset.jsonl
--model gpt-4o-mini --mode few-shot --output results_few_shot.jsonl
--max-samples 25
```

Domain-aware

```
python3 run_experiments.py --dataset data/datasets/unity_dataset.jsonl
--model gpt-4o-mini --mode domain --output results_domain.jsonl
--max-samples 25
```

Few-shot + Domain-aware

```
python3 run_experiments.py --dataset data/datasets/unity_dataset.jsonl
--model gpt-4o-mini --mode few-shot-domain --output
results_few_shot_domain.jsonl --max-samples 25
```

All generated files will be placed in the project directory unless you specify a different output folder.

Running Automatic Evaluation

Once the experiment files are created, evaluate all modes using BLEU and CodeBLEU:

```
python3 evaluate_results.py --results results_zero_shot.jsonl
results_few_shot.jsonl results_domain.jsonl
results_few_shot_domain.jsonl --output evaluation_summary.json
```

This produces a summary file with average BLEU and CodeBLEU scores for each experiment type.