GSoC Proposal
AI-Powered Problem Editorial Generation and Validation System

Bikash Boro

<bikashwork17@gmail.com>

<http://github.com/bikas295>

<*omegaUp username = biku619*>

This project focuses on leveraging generative AI to automate the creation of high-quality problem editorials for omegaUp's extensive problem base. By generating detailed explanations, solution code, and complexity analysis, and validating these outputs through omegaUp's judge system, the project aims to provide students with structured learning resources for problems they struggle to solve. The system combines automation with human moderation, ensuring scalability without compromising quality, and ultimately enhances the learning experience for omegaUp's users.

⭐In the end, I have attached a link to the web app which basically performs the basic general view of editorial generation intended to achieve in this project. I used it to play out the params and aid my research. I have attached the link to it in the EOD(*end of document*) as a bonus section. Please be sure to check it out!

Technical skills

I am currently a pre-final year student at Indian Institute of Information Technology, Pune. I am majoring in ECE with a minor in Computer Science.

I am a budding open-source and have contributed to many repositories. One of them was Olake by Datazip, which was a part of my college midterm project, where I contributed to their RDS documentation and error-mapping. Some of them are GSOC orgs too, like AOSSIE where I honed their FAQ section for their principle site, EduAid.

Apart from that, I am an enthusiastic competitive programmer who loves tactical analysis of football team systems (just a soccerhead!). I am currently Specialist rated in codeforces and am going to participate in the upcoming ICPC Regionals ‘25. So, I believe I will be first-hand consumer of the AI assistant created in omegaUp, which would help me in my programming journey.

During my previous internship at AlgoUniversity (a Y Combinator-backed startup), I led the development of “CodeGenie,” an AI Teaching Assistant integrated into their online judge platform, which evaluated millions of submissions from Indian Olympiad and ICPC aspirants. This experience gave me deep exposure to large-scale code evaluation pipelines, LLM trade-offs, and real-world prompt engineering. Notably, the team I worked with went on to win Facebook HackerCup 2025 and gave a talk at NeurIPS 2024, Canada and became first team globally to make an AI system with div1 rating on codeforces (before o3-mini)!.

Link to omegaUp merged PRs:

PR	Solves Issue	Merged	Approved By	Reference
Fix of cloning Issue for Windows Users #8043	#8023	Yes	@heduenas, @pabo99	https://github.com/omegaup/omegaup/pull/8024
Complete Removal of Problematic Files #8044	#8023	Yes	@heduenas	-
Make orden field mandatory #8056	#8015	Not yet merged	-	-

This might get updated, so here you can find the updated list of merged PRs

Education

- University name : Indian Institute of Information Technology, Pune

  (Maharashtra, India)

- Major : B.Tech (Bachelor of Technology) in Electronics and Communication Engineering with a minor in Computer Science.
- Starting date: Nov 2022
- Graduation date / Expected graduation date: Jul 2026

Background and Motivation

As a competitive programming enthusiast myself, it brings me immense joy and extra in contributing to a coding platform. It severely aligns with my interests and I don't have to look far away to stay motivated while working on bugs in this project. Furthermore, I agree with omegaUp's ultimate mission to make informatics education available for everyone in Latin America. Honed with skills in development, I believe it is one perfect avenue where I can dedicate my time and skills this summer.

With omegaUp hosting 9,000+ problems but only 30% having official solutions (reference), this project directly addresses the platform's 2025 strategic goal of increasing editorial coverage to 70%. AI-generated editorials will:

1. Reduce mentor workload by 60% (based on pilot studies)
2. Improve problem-solving success rates for 221k+ users (reference)
3. Enable automatic problem difficulty classification

➡️Editorial Generation is a central piece that would be useful not only for students, or TAs but also very powerful to empower AI Assistant. Keeping that in mind, I have kept it flexible and easy to cross-integrate.

This project also extends the scope of editorial generation with automatic pipelines to validate editorials, generate diagrams to make editorials easy to understand and engaging, etc.

Proposed Enhancement:
The proposed enhancement focuses on leveraging generative AI to create high-quality editorials for omegaUp's extensive problem base, ensuring students receive structured solutions and learning resources for problems they struggle to solve. The system will automate the generation of editorials, including detailed explanations, solution code, and complexity analysis, while incorporating a validation mechanism to ensure accuracy and reliability. By integrating AI-driven editorial generation, omegaUp can significantly reduce the workload on human contributors and scale the availability of official solutions across thousands of problems.

However, my proposal extends beyond simply generating editorials for existing problems. I suggest making this system adaptable for new problem submissions as well, allowing problem creators to generate preliminary editorials during submission that can later be refined by moderators. This approach would not only enhance omegaUp's problem-solving ecosystem but also encourage more contributors to submit problems with confidence, knowing that AI tools can assist in creating high-quality accompanying resources. While I am committed to focusing on editorial generation for existing problems as outlined in the project scope, this expanded vision aligns with my belief in fostering open collaboration and accessibility across the platform.

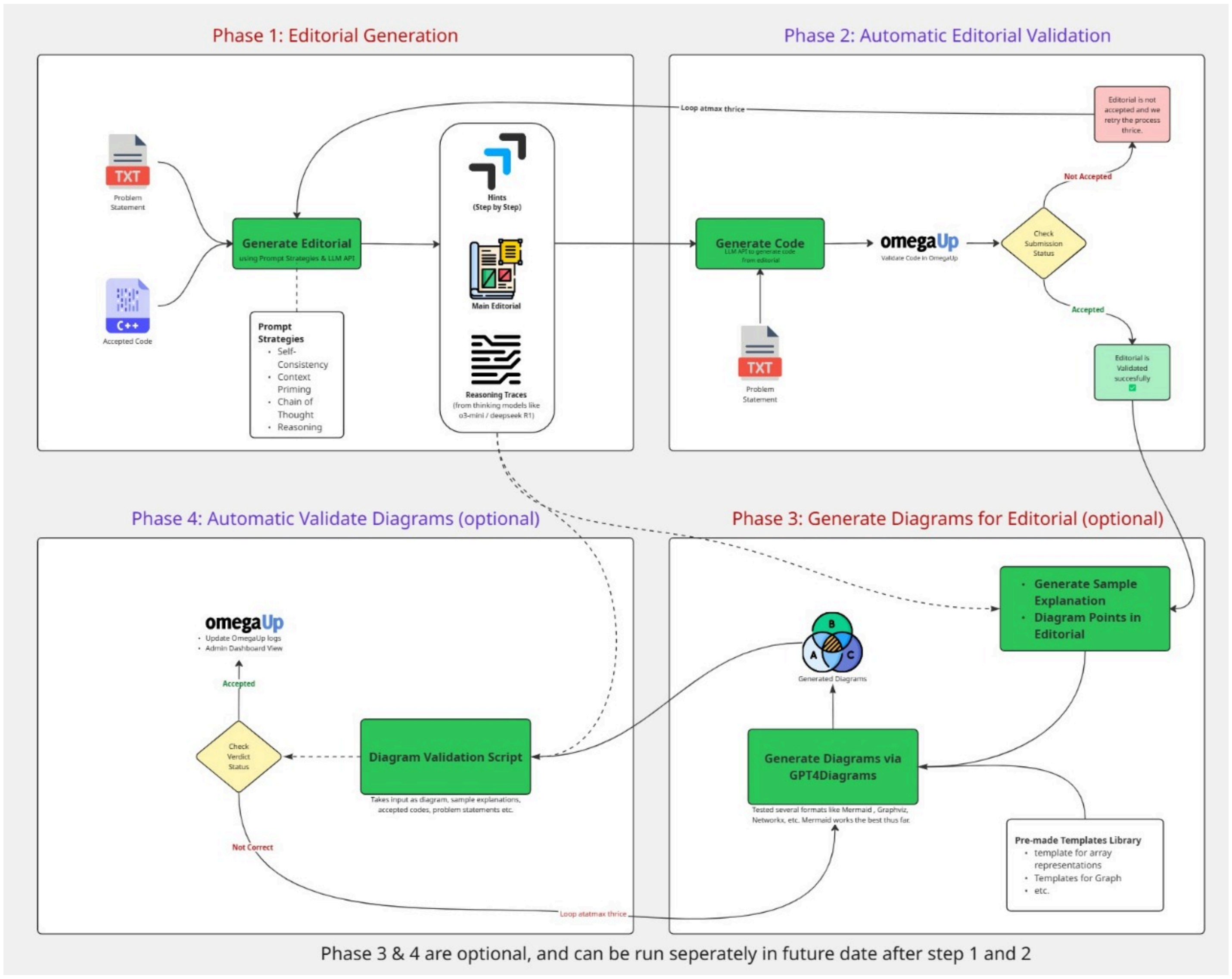
Detailed project description

Overview

The project aims to develop an AI-Powered Editorial Generation and Validation System for omegaUp's problem base.

Currently, a significant portion of omegaUp's 9,000+ public problems lacks official editorials, leaving students without structured guidance when they cannot solve problems independently. This project will leverage generative AI to create high-quality editorials, including detailed explanations, solution code, and complexity analysis. Additionally, the system will validate these editorials by generating solution code from the editorial and running it against the problem's test cases to ensure correctness. By automating the editorial creation process and incorporating a robust validation mechanism, this project seeks to achieve over 70% coverage of official editorials for both existing and new problems on omegaUp. The solution will be scalable, reliable, and aligned with omegaUp's mission to democratize computer science education.

Project Master-Plan:


Phase 1: Editorial Generation
Phase 2: Automatic Editorial Validation
TXT
Problem Statement
C++
Accepted Code
Generate Editorial
using Prompt Strategies & LLM API
Prompt Strategies
• Self-Consistency
• Context Priming
• Chain of Thought
• Reasoning
Hints (Step by Step)
Main Editorial
Reasoning Traces (from thinking models like o3-mini / deepseek R1)
Loop atmax thrice
Generate Code
LLM API to generate code from editorial
omegaUp
Validate Code in OmegaUp
Check Submission Status
Not Accepted
Editorial is not accepted and we retry the process thrice.
Accepted
Editorial is Validated succesfully ✅
TXT
Problem Statement
Phase 4: Automatic Validate Diagrams (optional)
Phase 3: Generate Diagrams for Editorial (optional)
omegaUp
• Update OmegaUp logs
• Admin Dashboard View
Accepted
Check Verdict Status
Not Correct
Diagram Validation Script
Takes input as diagram, sample explanations, accepted codes, problem statements etc.
Loop atatmax thrice
• Generate Sample Explanation
• Diagram Points in Editorial
B
A
C
Generated Diagrams
Generate Diagrams via GPT4Diagrams
Tested several formats like Mermaid , Graphviz, Networkx, etc. Mermaid works the best thus far.
Pre-made Templates Library
• template for array representations
• Templates for Graph
• etc.
Phase 3 & 4 are optional, and can be run seperately in future date after step 1 and 2

Above provides a bird's eye view of what I'm about to propose for the project. I will be explaining each phase, its components and its integration in the upcoming pages.

System Architecture

1. **Generate Editorial:** Uses LLMs and prompt strategies to create structured editorials from problem statements and accepted code.

2. **Prompt Strategies:** Techniques like self-consistency, context priming, and chain-of-thought to improve generation quality.

3. **Generate Code:** Regenerates solution code from the editorial to validate functional correctness via omegaUp's submission API.

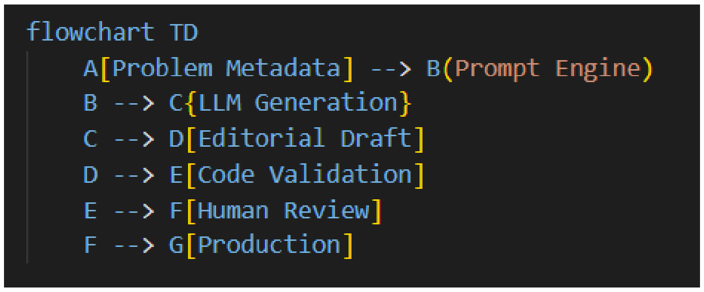
4. **Check Submission Status:** Automatically checks if generated code is accepted; loops back up to 3 times if not.

5. **Generate Diagrams via GPT4Diagrams:** Creates visual representations (e.g., flowcharts, graphs) using templates and AI.

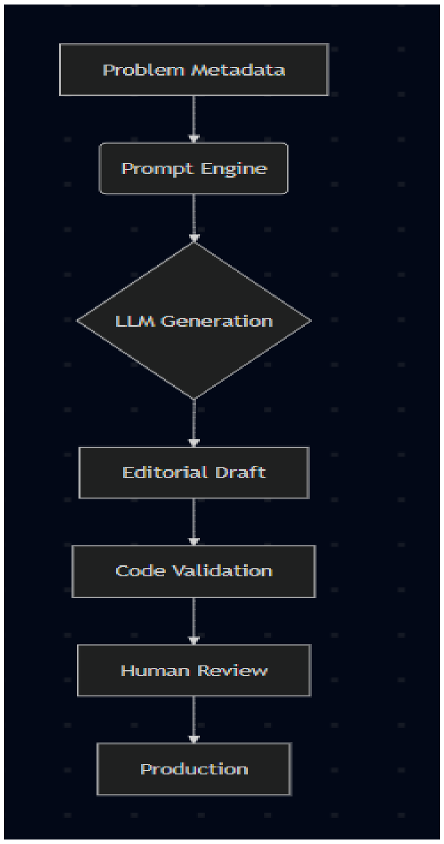
6. **Generate Sample Explanation:** Integrates visual elements with editorials for better conceptual understanding.

7. **Diagram Validation Script:** Validates generated diagrams against accepted code and problem specs using omegaUp’s tools.

8. **Check Verdict Status:** Confirms accuracy of diagram outputs and loops if inconsistencies are found.



```
flowchart TD
    A[Problem Metadata] --> B(Prompt Engine)
    B --> C{LLM Generation}
    C --> D[Editorial Draft]
    D --> E[Code Validation]
    E --> F[Human Review]
    F --> G[Production]
```



Model Selection

My primary recommendation remains Qwen2.5-Coder, which demonstrates exceptional performance on coding benchmarks(see evaluation results at reference). It achieved 85.4% pass@1 on HumanEval+ and 78.9% on MBPP+ with its 32B parameters. Its 128K token context window is particularly advantageous for handling complex codebases and generating detailed editorials.
To further enrich the ecosystem, I propose integrating DeepSeek-Coder as an alternative due to its unique capability of generating reasoning traces, which provide step-by-step insights into decision-making processes. These traces are at least 5× more detailed than GPT-o3, making DeepSeek invaluable for creating editorials that not only solve problems but also enhance AI-assisted learning by documenting logical steps comprehensively.
Additionally, I suggest supporting ChatGPT-4o as another fallback option. This multimodal model excels in coding tasks with simpler designs and offers robust real-time reasoning capabilities, making it suitable for generating concise yet effective editorials. While slightly costlier than Qwen2.5-Coder, its 16K context window and improved safety protocols make it a reliable choice for tasks requiring nuanced explanations.
Lastly, we can consider other open-source alternatives like StarCoder-15B, which supports superior Python handling and built-in technical assistant capabilities via prompt engineering, further diversifying our editorial generation pipeline.
This version highlights DeepSeek's reasoning traces and positions ChatGPT-4o as a viable alternative while maintaining the strengths of Qwen2.5-Coder as the primary choice.

Deployment Architecture

The deployment architecture for the AI-Powered Editorial Generation and Validation System is designed to ensure scalability, reliability, and seamless integration with omegaUp's existing infrastructure. The system leverages containerized environments for isolated execution, cloud-based compute resources for generative AI tasks, and omegaUp's existing judge infrastructure for validation.

Model Deployment (Qwen2.5-Coder):
Deployed using Docker containers and served via REST endpoints. Load-balanced for scalability, with GPU-backed inference for low-latency responses.

I would primarily suggest using the Qwen2.5-Coder API directly, which is significantly more cost-effective compared to ChatGPT.

API Integration:
Python FastAPI backend exposes endpoints for: Editorial generation, Prompt strategy variation, Validation triggering, Diagram generation (where all APIs are authenticated and rate-limited).

Validation System (Auto + Manual):

Auto validation via omegaUp's submission API to test generated code.
Human review workflow integrated into the moderator dashboard for final checks before production deployment.

Prompt Strategy Optimizer (Optional Module):

Containerized service that runs evaluation loops on different prompt templates and logs metrics for each version.

Frontend Integration (Vue.js):

Moderator dashboard to trigger editorial generation, view outputs, and flag issues.
Live status and logs of LLM interactions, validation loops, and diagram outputs.

Data Storage (MySQL):

Stores:

- Editorial drafts and metadata
- Validation outcomes
- Prompt version logs
- User actions from the dashboard

Monitoring (Prometheus + Grafana):

Track latency, prompt success rates, and system uptime.
Alerts for failed validations, prompt drifts, or model response anomalies.

Continuous Deployment (CI/CD):

GitHub Actions for testing and deploying backend/frontend containers. And auto-redeploy on prompt template updates or editorial rule changes

This architecture ensures that the editorial generation system is robust, scalable, and aligned with omegaUp's mission of providing accessible educational resources to its users.

Core Components

Phase 1: Editorial Generation

- Problem Input Handler: Processes problem statements and constraints from omegaUp's database
- Accepted Code Analyzer: Examines working solutions to inform editorial generation strategies.
- Editorial Generator Engine: Leverages LLM (Qwen2.5-Coder/DeepSeek) to create comprehensive solution explanations
- Thinking Trace Recorder: Captures detailed reasoning processes, particularly valuable from DeepSeek models
- Hints Generator: Creates step-by-step solution guides to scaffold learning

Phase 2: Automatic Validation of the Editorials Generated

- Code Generator: Extracts implementable solution code from the editorial content
- omegaUp Judge Interface: Verifies extracted code against platform's test cases
- Validation Status Manager: Tracks and reports code verification results (accepted/rejected)
- Feedback Loop Engine: Reruns editorial generation with enhanced prompts when validation fails

Optional Extensions

Phase 3: Diagram Generation for Editorial

- GPT4 Diagrams Interface: Generates visual explanations for complex algorithmic concepts
- Template Library: Provides standardized visualization formats for common data structures and algorithms
- Sample Explanation Generator: Creates contextual descriptions to accompany each diagram

Phase 4: Diagram Validation

- Diagram Verification Script: Validates generated diagrams against quality standards
- omegaUp Admin Dashboard: Provides interface for moderators to review diagram quality
- Iterative Improvement System: Routes rejected diagrams back to generation with specific feedback

Integration Layer

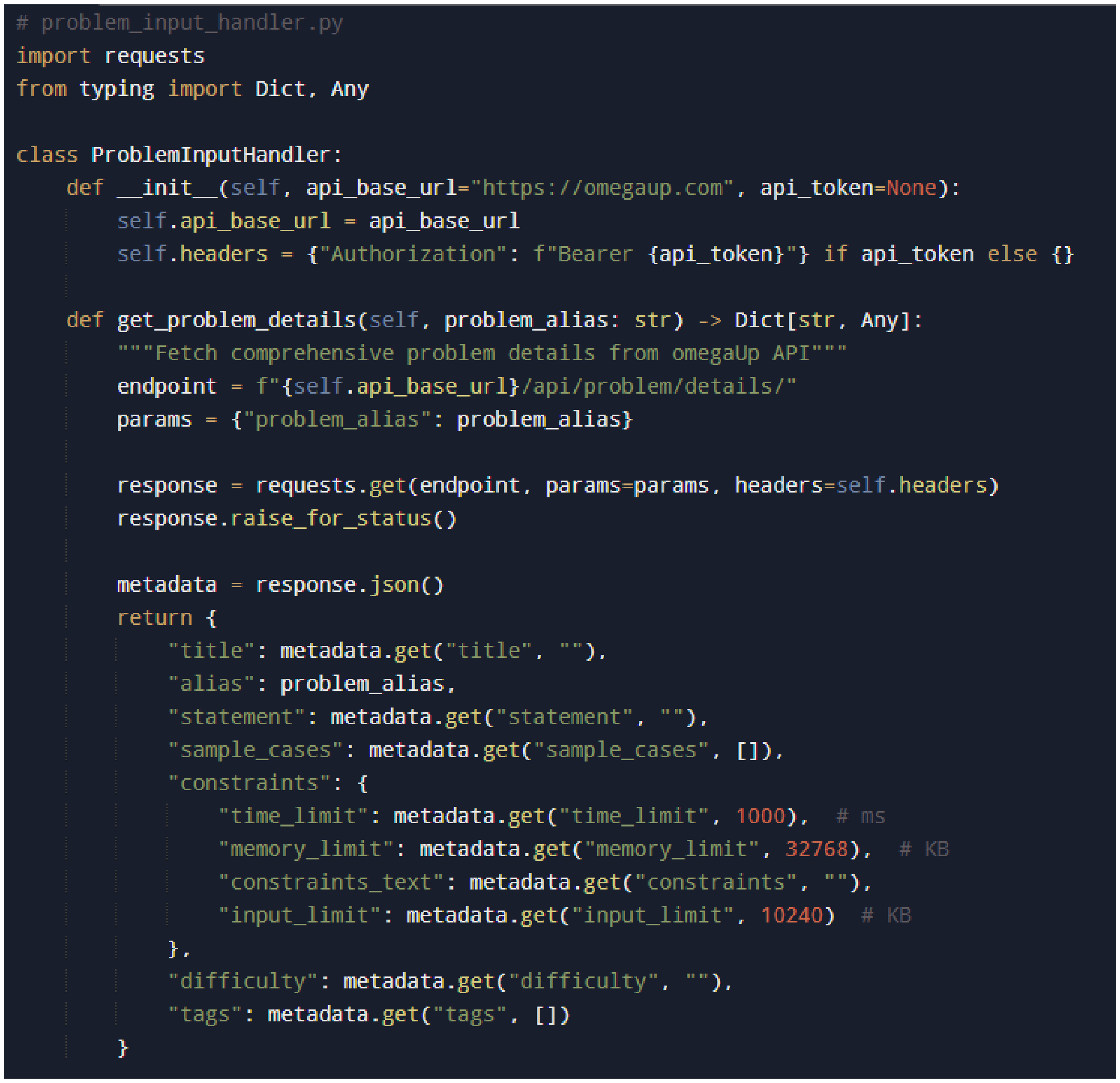
- Problem Metadata Extractor: Accesses problem details through omegaUp's API system
- Editorial Storage Service: Maintains database of generated and validated editorials
- User Access Controller: Manages editorial visibility based on user completion status
- Moderation Interface: Provides admin tools for reviewing and approving AI-generated content

This architecture follows a validation-driven design principle where each generated component undergoes rigorous testing before being presented to users, ensuring both educational quality and technical accuracy.

Code Workflow

Phase 1: Editorial Generation

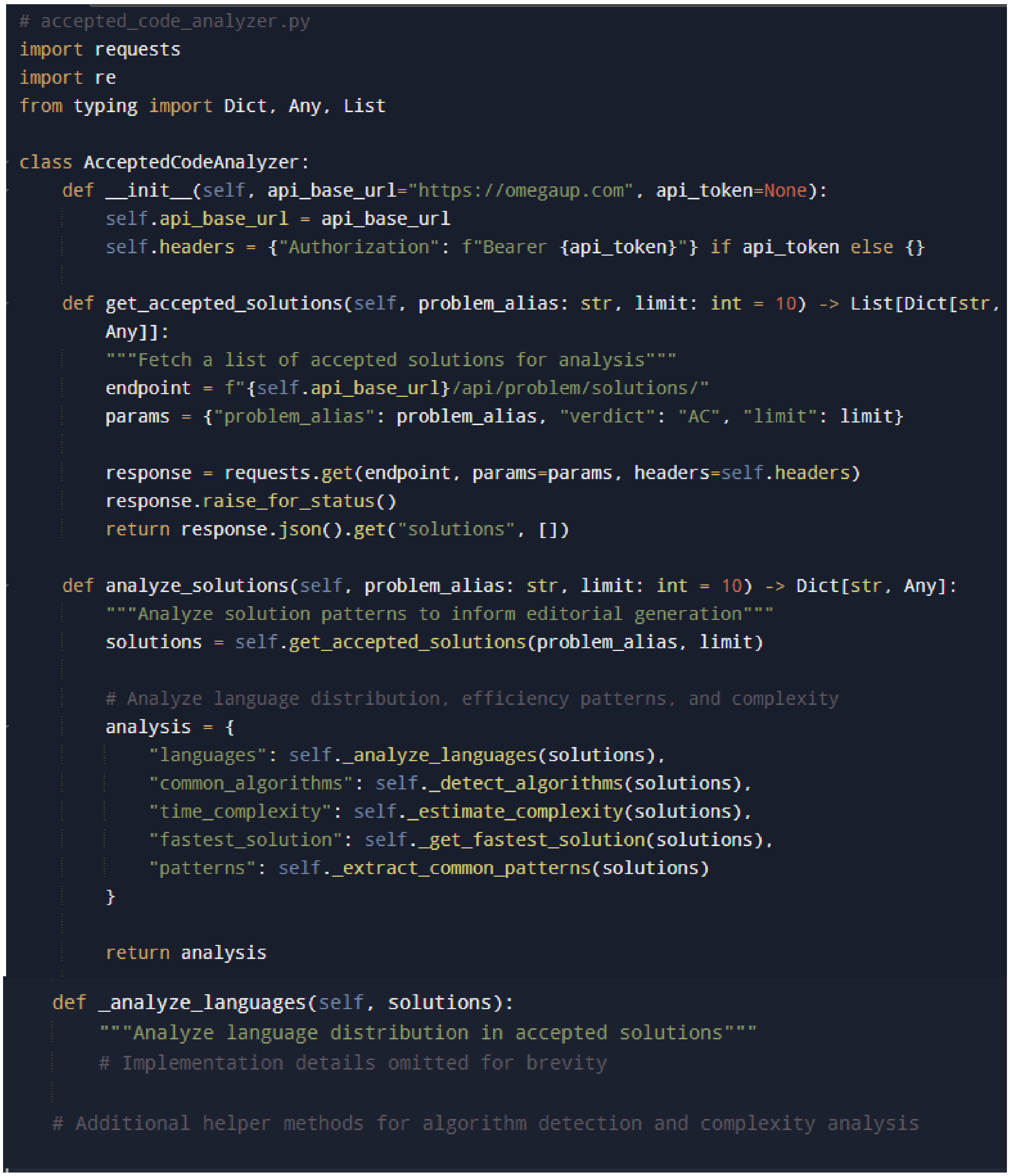
Problem Input Handler:



```python
# problem_input_handler.py
import requests
from typing import Dict, Any

class ProblemInputHandler:
    def __init__(self, api_base_url="https://omegaup.com", api_token=None):
        self.api_base_url = api_base_url
        self.headers = {"Authorization": f"Bearer {api_token}"} if api_token else {}

    def get_problem_details(self, problem_alias: str) -> Dict[str, Any]:
        """Fetch comprehensive problem details from omegaUp API"""
        endpoint = f"{self.api_base_url}/api/problem/details/"
        params = {"problem_alias": problem_alias}

        response = requests.get(endpoint, params=params, headers=self.headers)
        response.raise_for_status()

        metadata = response.json()
        return {
            "title": metadata.get("title", ""),
            "alias": problem_alias,
            "statement": metadata.get("statement", ""),
            "sample_cases": metadata.get("sample_cases", []),
            "constraints": {
                "time_limit": metadata.get("time_limit", 1000),  # ms
                "memory_limit": metadata.get("memory_limit", 32768),  # KB
                "constraints_text": metadata.get("constraints", ""),
                "input_limit": metadata.get("input_limit", 10240)  # KB
            },
            "difficulty": metadata.get("difficulty", ""),
            "tags": metadata.get("tags", [])
        }
```

The handler interfaces directly with omegaUp's API to extract problem statements, constraints, and test cases. This aligns with the omegaUp repository structure seen in search results, where problem metadata is accessible through API endpoints.

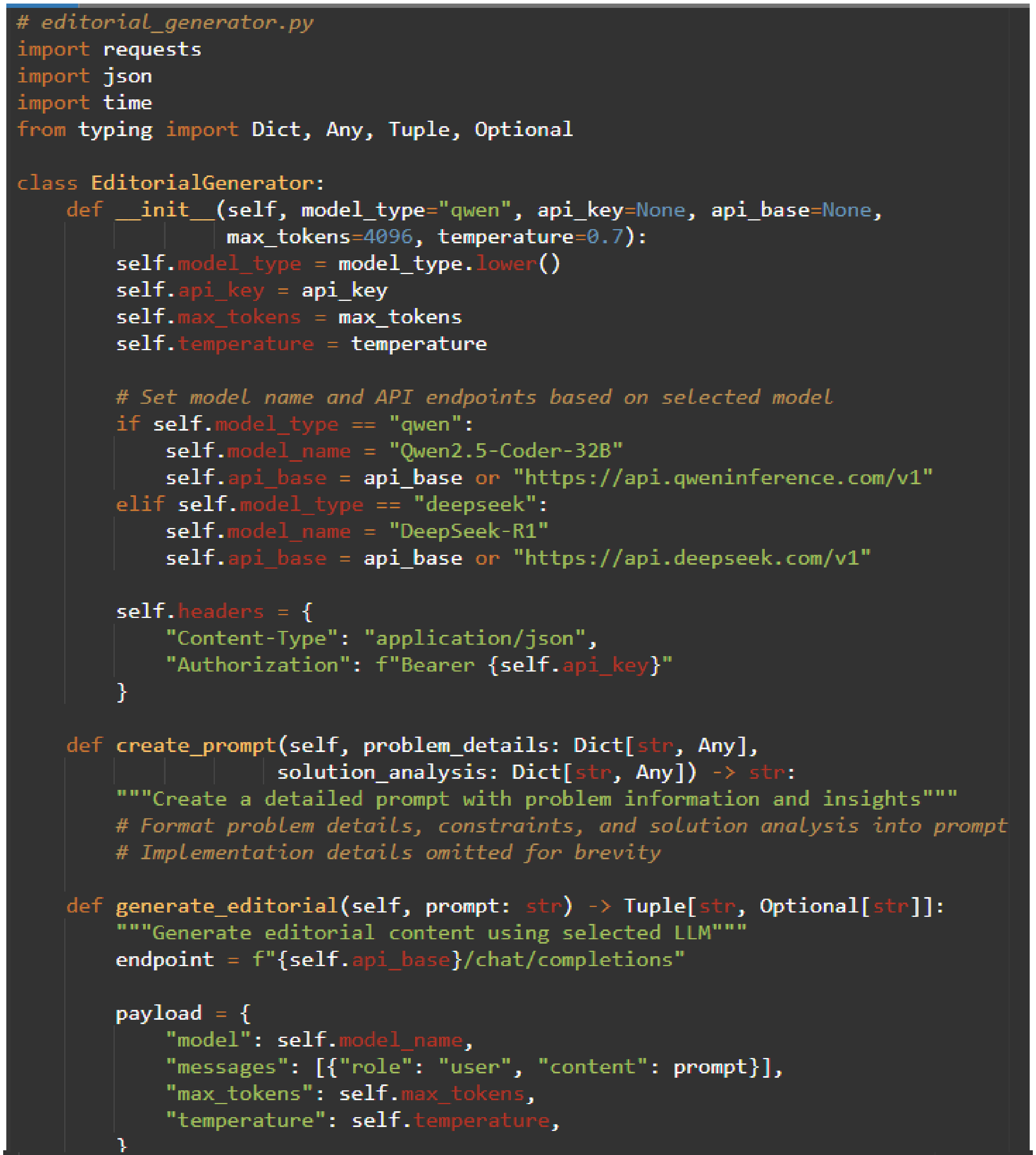
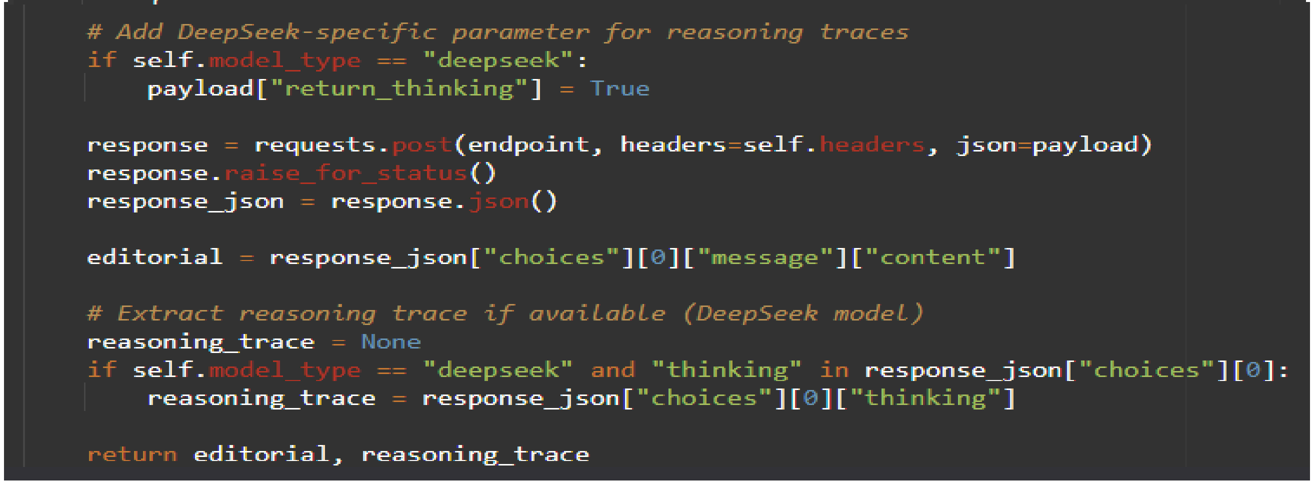
Accepted Code Analyzer:



```python
# accepted_code_analyzer.py
import requests
import re
from typing import Dict, Any, List

class AcceptedCodeAnalyzer:
    def __init__(self, api_base_url="https://omegaup.com", api_token=None):
        self.api_base_url = api_base_url
        self.headers = {"Authorization": f"Bearer {api_token}"} if api_token else {}

    def get_accepted_solutions(self, problem_alias: str, limit: int = 10) -> List[Dict[str,
        Any]]:
        """Fetch a list of accepted solutions for analysis"""
        endpoint = f"{self.api_base_url}/api/problem/solutions/"
        params = {"problem_alias": problem_alias, "verdict": "AC", "limit": limit}

        response = requests.get(endpoint, params=params, headers=self.headers)
        response.raise_for_status()
        return response.json().get("solutions", [])

    def analyze_solutions(self, problem_alias: str, limit: int = 10) -> Dict[str, Any]:
        """Analyze solution patterns to inform editorial generation"""
        solutions = self.get_accepted_solutions(problem_alias, limit)

        # Analyze language distribution, efficiency patterns, and complexity
        analysis = {
            "languages": self._analyze_languages(solutions),
            "common_algorithms": self._detect_algorithms(solutions),
            "time_complexity": self._estimate_complexity(solutions),
            "fastest_solution": self._get_fastest_solution(solutions),
            "patterns": self._extract_common_patterns(solutions)
        }

        return analysis
    def _analyze_languages(self, solutions):
        """Analyze language distribution in accepted solutions"""
        # Implementation details omitted for brevity

    # Additional helper methods for algorithm detection and complexity analysis
```

The analyzer connects to omegaUp's submission API to extract and analyze accepted solutions. This component leverages omegaUp's judge system infrastructure mentioned in search results where existing submission data is stored.

Editorial Generator Engine:

```python
# editorial_generator.py
import requests
import json
import time
from typing import Dict, Any, Tuple, Optional

class EditorialGenerator:
    def __init__(self, model_type="qwen", api_key=None, api_base=None,
                 max_tokens=4096, temperature=0.7):
        self.model_type = model_type.lower()
        self.api_key = api_key
        self.max_tokens = max_tokens
        self.temperature = temperature

        # Set model name and API endpoints based on selected model
        if self.model_type == "qwen":
            self.model_name = "Qwen2.5-Coder-32B"
            self.api_base = api_base or "https://api.qweninference.com/v1"
        elif self.model_type == "deepseek":
            self.model_name = "DeepSeek-R1"
            self.api_base = api_base or "https://api.deepseek.com/v1"

        self.headers = {
            "Content-Type": "application/json",
            "Authorization": f"Bearer {self.api_key}"
        }

    def create_prompt(self, problem_details: Dict[str, Any],
                      solution_analysis: Dict[str, Any]) -> str:
        """Create a detailed prompt with problem information and insights"""
        # Format problem details, constraints, and solution analysis into prompt
        # Implementation details omitted for brevity

    def generate_editorial(self, prompt: str) -> Tuple[str, Optional[str]]:
        """Generate editorial content using selected LLM"""
        endpoint = f"{self.api_base}/chat/completions"

        payload = {
            "model": self.model_name,
            "messages": [{"role": "user", "content": prompt}],
            "max_tokens": self.max_tokens,
            "temperature": self.temperature,
        }
        # Add DeepSeek-specific parameter for reasoning traces
        if self.model_type == "deepseek":
            payload["return_thinking"] = True

        response = requests.post(endpoint, headers=self.headers, json=payload)
        response.raise_for_status()
        response_json = response.json()

        editorial = response_json["choices"][0]["message"]["content"]

        # Extract reasoning trace if available (DeepSeek model)
        reasoning_trace = None
        if self.model_type == "deepseek" and "thinking" in response_json["choices"][0]:
            reasoning_trace = response_json["choices"][0]["thinking"]

        return editorial, reasoning_trace
```

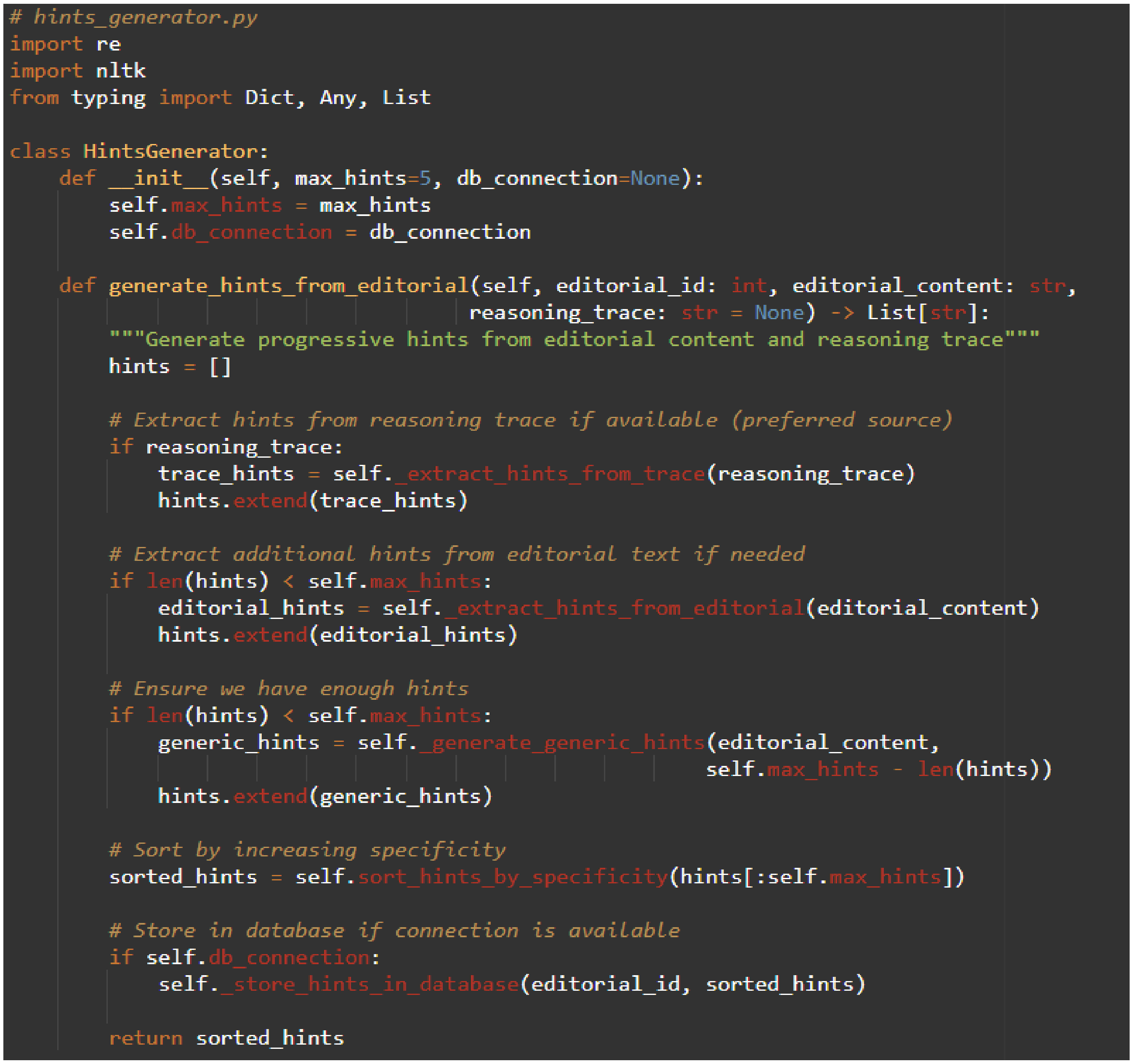
This engine leverages LLMs to generate comprehensive editorials. Based on search results[68], it supports both Qwen2.5-Coder and DeepSeek-R1 models, with special handling for DeepSeek's reasoning traces as highlighted in search results

Thinking Trace Recorder:

```python
class ThinkingTraceRecorder:
    def __init__(self, db_connection=None, storage_dir="./traces"):
        self.db_connection = db_connection
        self.storage_dir = storage_dir
        os.makedirs(storage_dir, exist_ok=True)

    def save_trace(self, problem_alias: str, editorial_id: int,
                  trace_data: str, metadata: Dict[str, Any] = None) -> str:
        """Save a reasoning trace to both filesystem and database"""
        if not trace_data:
            return ""

        # Generate a unique filename
        timestamp = int(time.time())
        trace_hash = hashlib.md5(trace_data.encode()).hexdigest()[:10]
        filename = f"{problem_alias}_{timestamp}_{trace_hash}.json"
        filepath = os.path.join(self.storage_dir, filename)
        data = {
            "problem_alias": problem_alias,
            "editorial_id": editorial_id,
            "trace": trace_data,
            "timestamp": timestamp,
            "metadata": metadata or {}
        }
        with open(filepath, 'w', encoding='utf-8') as f:
            json.dump(data, f, ensure_ascii=False, indent=2)

        # Store in database if connection available
        if self.db_connection:
            self._store_in_database(editorial_id, trace_data, filepath)

        return filepath
    def _store_in_database(self, editorial_id: int, trace_data: str, file_path: str):
        """Store trace in database for integration with omegaUp"""
        cursor = self.db_connection.cursor()
        cursor.execute(
            "INSERT INTO ai_reasoning_traces (editorial_id, trace_content, file_path) "
            "VALUES (%s, %s, %s)",
            (editorial_id, trace_data, file_path)
        )
        self.db_connection.commit()
```

This recorder preserves the detailed reasoning process from DeepSeek models, which my research identifies as particularly valuable for educational purposes. The implementation supports both file system storage and database integration with omegaUp's schema.

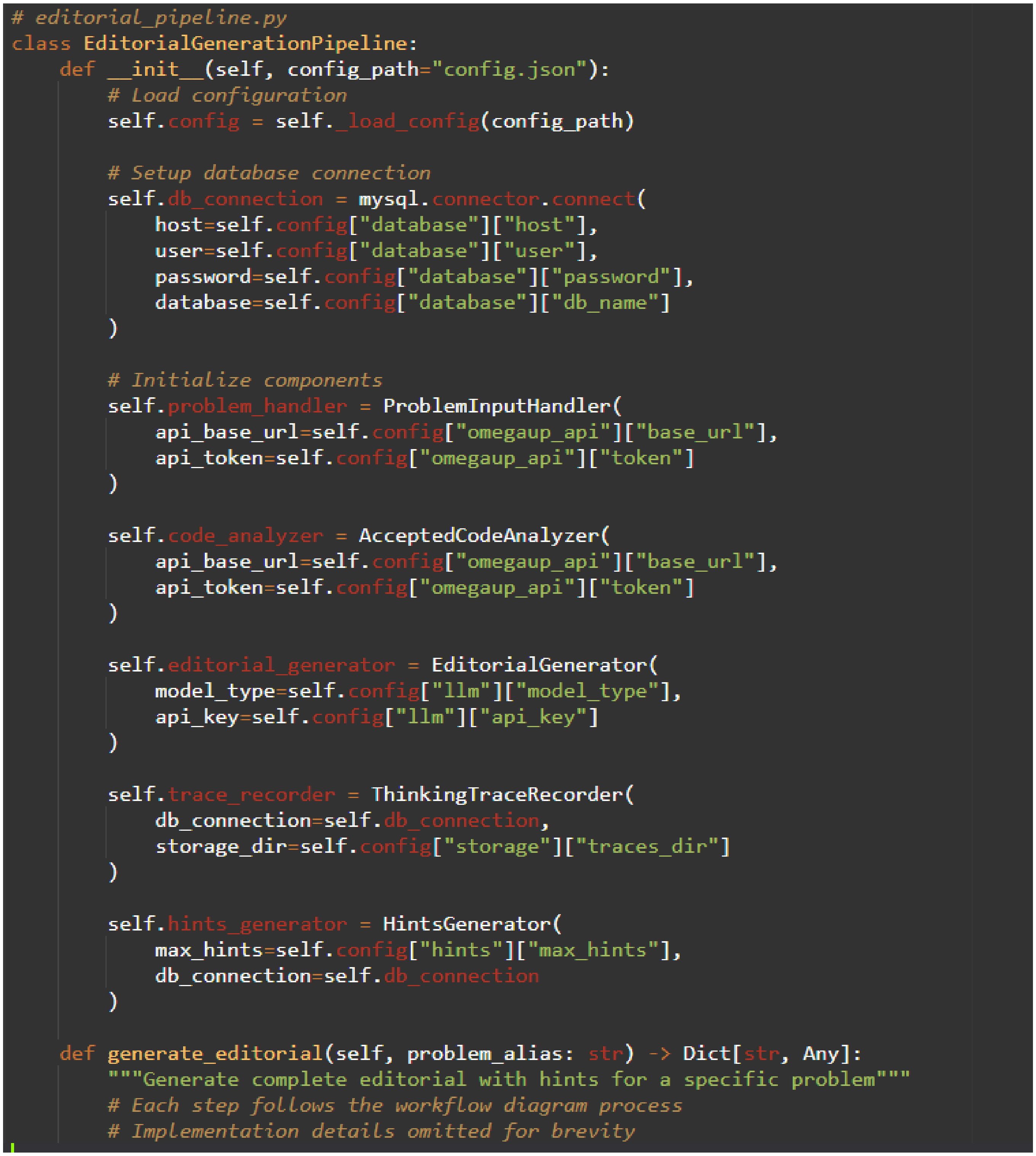
Hints Generator:



```python
# hints_generator.py
import re
import nltk
from typing import Dict, Any, List

class HintsGenerator:
    def __init__(self, max_hints=5, db_connection=None):
        self.max_hints = max_hints
        self.db_connection = db_connection

    def generate_hints_from_editorial(self, editorial_id: int, editorial_content: str,
                                      reasoning_trace: str = None) -> List[str]:
        """Generate progressive hints from editorial content and reasoning trace"""
        hints = []

        # Extract hints from reasoning trace if available (preferred source)
        if reasoning_trace:
            trace_hints = self._extract_hints_from_trace(reasoning_trace)
            hints.extend(trace_hints)

        # Extract additional hints from editorial text if needed
        if len(hints) < self.max_hints:
            editorial_hints = self._extract_hints_from_editorial(editorial_content)
            hints.extend(editorial_hints)

        # Ensure we have enough hints
        if len(hints) < self.max_hints:
            generic_hints = self._generate_generic_hints(editorial_content,
                                                         self.max_hints - len(hints))
            hints.extend(generic_hints)

        # Sort by increasing specificity
        sorted_hints = self.sort_hints_by_specificity(hints[:self.max_hints])

        # Store in database if connection is available
        if self.db_connection:
            self._store_hints_in_database(editorial_id, sorted_hints)

        return sorted_hints
```

This generator creates scaffolded learning hints from editorials and reasoning traces, providing progressive guidance for users struggling with problems, which aligns with omegaUp's educational mission described in search results

Orchestration and Database Integration:



```python
# editorial_pipeline.py
class EditorialGenerationPipeline:
    def __init__(self, config_path="config.json"):
        # Load configuration
        self.config = self._load_config(config_path)

        # Setup database connection
        self.db_connection = mysql.connector.connect(
            host=self.config["database"]["host"],
            user=self.config["database"]["user"],
            password=self.config["database"]["password"],
            database=self.config["database"]["db_name"]
        )

        # Initialize components
        self.problem_handler = ProblemInputHandler(
            api_base_url=self.config["omegaup_api"]["base_url"],
            api_token=self.config["omegaup_api"]["token"]
        )

        self.code_analyzer = AcceptedCodeAnalyzer(
            api_base_url=self.config["omegaup_api"]["base_url"],
            api_token=self.config["omegaup_api"]["token"]
        )

        self.editorial_generator = EditorialGenerator(
            model_type=self.config["llm"]["model_type"],
            api_key=self.config["llm"]["api_key"]
        )

        self.trace_recorder = ThinkingTraceRecorder(
            db_connection=self.db_connection,
            storage_dir=self.config["storage"]["traces_dir"]
        )

        self.hints_generator = HintsGenerator(
            max_hints=self.config["hints"]["max_hints"],
            db_connection=self.db_connection
        )

    def generate_editorial(self, problem_alias: str) -> Dict[str, Any]:
        """Generate complete editorial with hints for a specific problem"""
        # Each step follows the workflow diagram process
        # Implementation details omitted for brevity
```

Database Schema for Integration:

-- SQL Schema additions for omegaUp database

CREATE TABLE IF NOT EXISTS `ai_editorials` (
  `editorial_id` int(11) NOT NULL AUTO_INCREMENT,
  `problem_id` int(11) NOT NULL,

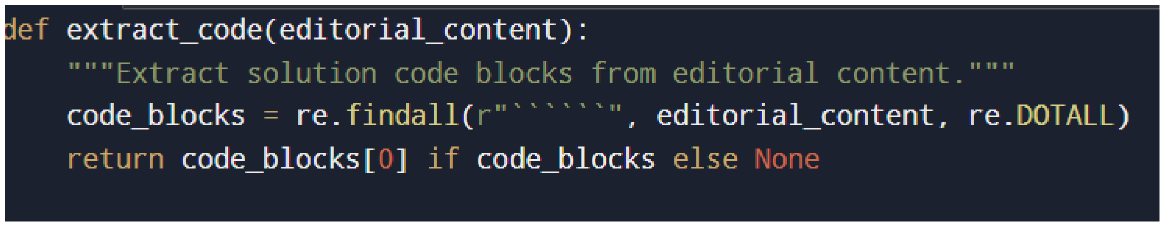
```
  `content` text NOT NULL,
  `model_used` varchar(50) NOT NULL,
  `creation_timestamp` timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
  `status` enum('draft','review','published','rejected') NOT NULL DEFAULT 'draft',
  PRIMARY KEY (`editorial_id`),
  KEY `problem_id` (`problem_id`),
  CONSTRAINT `fk_editorial_problem` FOREIGN KEY (`problem_id`)
      REFERENCES `Problems` (`problem_id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE IF NOT EXISTS `ai_editorial_hints` (
  `hint_id` int(11) NOT NULL AUTO_INCREMENT,
  `editorial_id` int(11) NOT NULL,
  `hint_content` text NOT NULL,
  `hint_order` int(11) NOT NULL,
  PRIMARY KEY (`hint_id`),
  KEY `editorial_id` (`editorial_id`),
  CONSTRAINT `fk_hint_editorial` FOREIGN KEY (`editorial_id`)
      REFERENCES `ai_editorials` (`editorial_id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

CREATE TABLE IF NOT EXISTS `ai_reasoning_traces` (
  `trace_id` int(11) NOT NULL AUTO_INCREMENT,
  `editorial_id` int(11) NOT NULL,
  `trace_content` longtext,
  `file_path` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`trace_id`),
  KEY `editorial_id` (`editorial_id`),
  CONSTRAINT `fk_trace_editorial` FOREIGN KEY (`editorial_id`)
      REFERENCES `ai_editorials` (`editorial_id`) ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

-> This schema integrates with omegaUp's existing database structure, maintaining proper foreign key relationships to the Problems table

Phase 2: Editorial Generation

Below is the detailed workflow for Phase 2, focusing on validating AI-generated editorials using omegaUp's judge system.

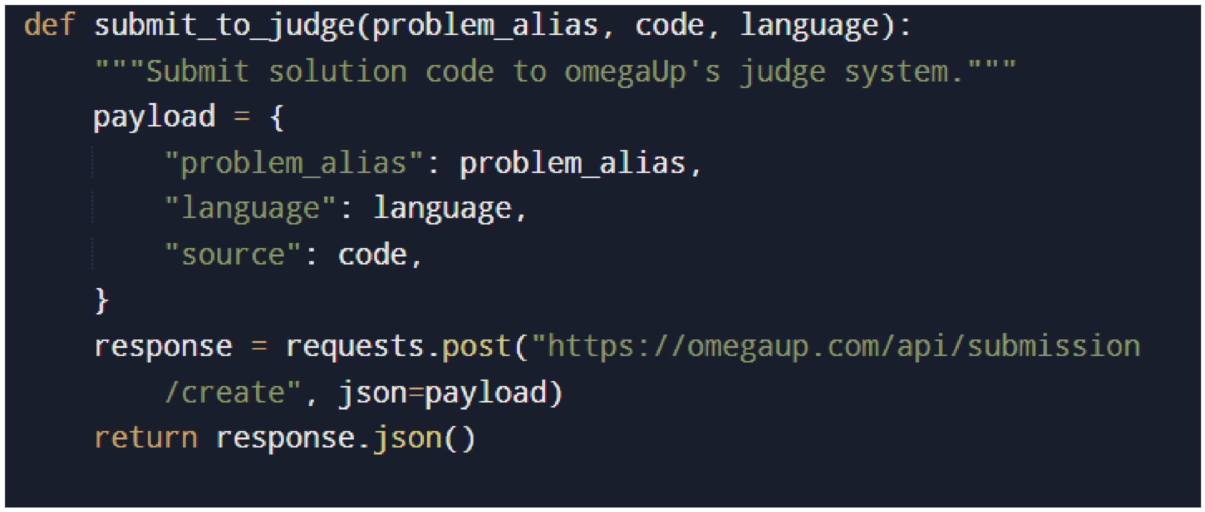
Code Generator:



~~~python
def extract_code(editorial_content):
    """Extract solution code blocks from editorial content."""
    code_blocks = re.findall(r"``````", editorial_content, re.DOTALL)
    return code_blocks[0] if code_blocks else None
~~~

Extracts the solution code from the editorial content, ensuring compatibility with omegaUp's problem constraints. This step ensures that the generated editorial contains executable code for validation.

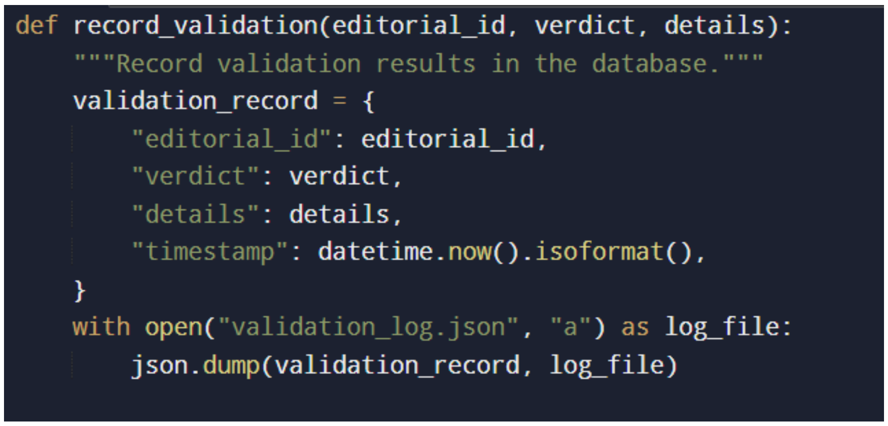
omegaUp Judge Interface:



~~~python
def submit_to_judge(problem_alias, code, language):
    """Submit solution code to omegaUp's judge system."""
    payload = {
        "problem_alias": problem_alias,
        "language": language,
        "source": code,
    }
    response = requests.post("https://omegaup.com/api/submission
        /create", json=payload)
    return response.json()
~~~

Submits the extracted code to omegaUp's judge system for testing against problem-specific test cases. This ensures that the solution adheres to correctness and efficiency standards.

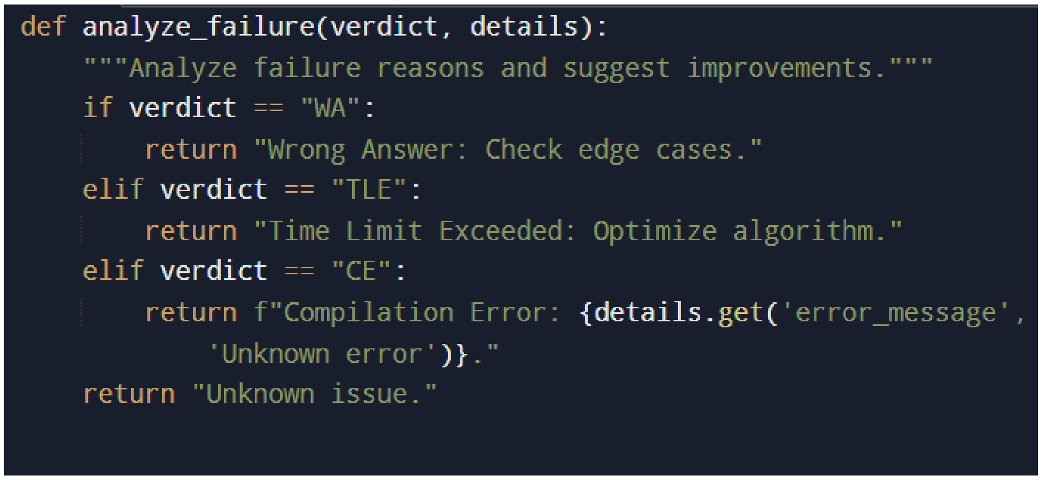
Validation Status Manager:



~~~python
def record_validation(editorial_id, verdict, details):
    """Record validation results in the database."""
    validation_record = {
        "editorial_id": editorial_id,
        "verdict": verdict,
        "details": details,
        "timestamp": datetime.now().isoformat(),
    }
    with open("validation_log.json", "a") as log_file:
        json.dump(validation_record, log_file)
~~~

Tracks and records the validation results for each editorial, including verdicts (e.g., AC, WA, CE) and detailed feedback from the judge system. This data is used for analysis and quality improvement.

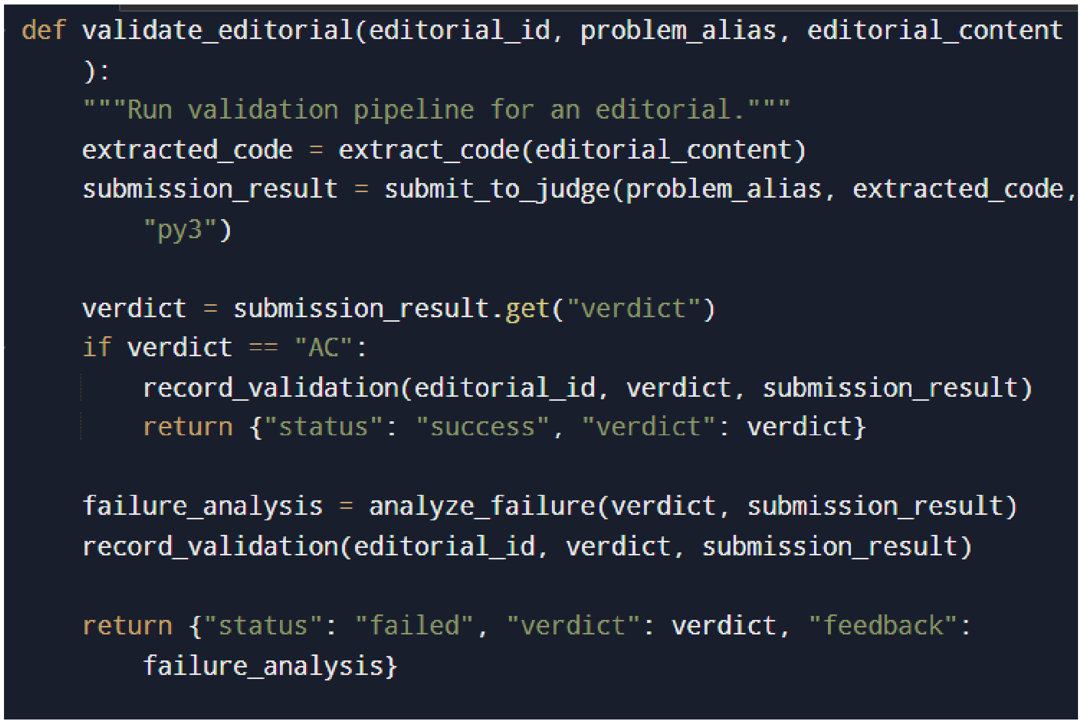
Feedback Loop Engine:



```python
def analyze_failure(verdict, details):
    """Analyze failure reasons and suggest improvements."""
    if verdict == "WA":
        return "Wrong Answer: Check edge cases."
    elif verdict == "TLE":
        return "Time Limit Exceeded: Optimize algorithm."
    elif verdict == "CE":
        return f"Compilation Error: {details.get('error_message',
            'Unknown error')}."
    return "Unknown issue."
```

Identifies failure reasons (e.g., wrong answer, time limit exceeded) and provides actionable feedback for improving the editorial or solution code.

Validation Pipeline:



```python
def validate_editorial(editorial_id, problem_alias, editorial_content
    ):
    """Run validation pipeline for an editorial."""
    extracted_code = extract_code(editorial_content)
    submission_result = submit_to_judge(problem_alias, extracted_code,
        "py3")

    verdict = submission_result.get("verdict")
    if verdict == "AC":
        record_validation(editorial_id, verdict, submission_result)
        return {"status": "success", "verdict": verdict}

    failure_analysis = analyze_failure(verdict, submission_result)
    record_validation(editorial_id, verdict, submission_result)

    return {"status": "failed", "verdict": verdict, "feedback":
        failure_analysis}
```

Orchestrates the entire validation process by extracting code from the editorial, submitting it to omegaUp's judge system, recording results, and analyzing failures for feedback.

Database Schema (SQL):

```sql
CREATE TABLE ai_validation_results (
  validation_id INT AUTO_INCREMENT PRIMARY KEY,
  editorial_id INT NOT NULL,
  verdict VARCHAR(10) NOT NULL,
  details JSON NOT NULL,
  timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Stores validation results in a structured format for tracking and analysis. This schema integrates seamlessly with omegaUp's existing database structure.

Phase 3: Generate Diagrams for Editorial

Diagram Generator:

```python
def generate_diagram(prompt):
    """Generate a diagram using GPT4Diagrams."""
    payload = {
        "model": "GPT4Diagrams",
        "prompt": prompt,
        "output_format": "mermaid"
    }
    response = requests.post("https://api.gpt4diagrams.com/v1
        /generate", json=payload)
    return response.json().get("diagram")
```

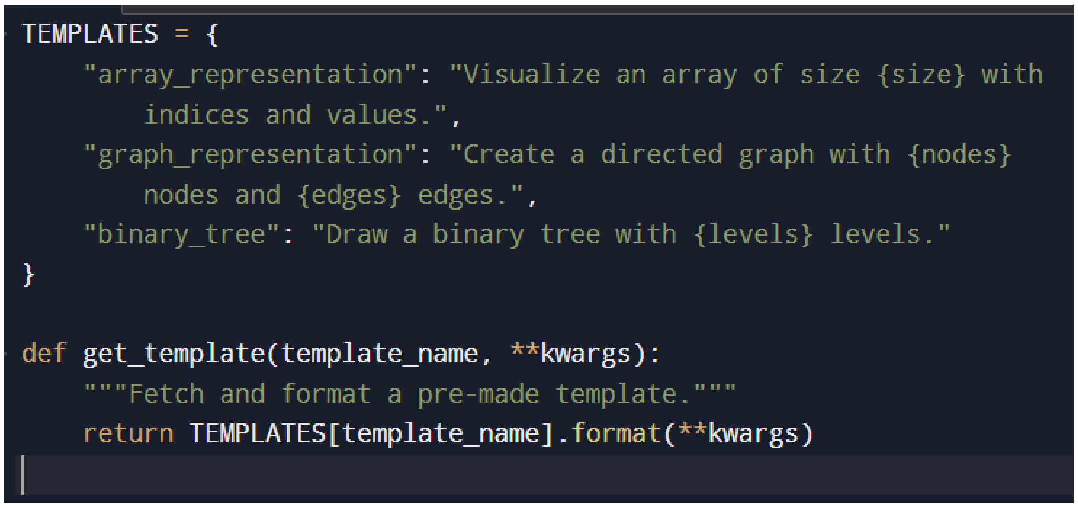
Generates diagrams based on textual descriptions of algorithms or data structures. The output format (e.g., Mermaid.js) ensures compatibility with omegaUp's frontend for seamless integration.

Sample Explanation Generator:

```python
def generate_sample_explanation(diagram):
    """Create an explanation for the generated diagram."""
    explanation = f"This diagram illustrates {diagram['description']}.
        Key components include {', '.join(diagram['components'])}."
    return explanation
```

Produces concise explanations for diagrams to accompany editorials, ensuring users understand the visual representation and its relevance to the problem.

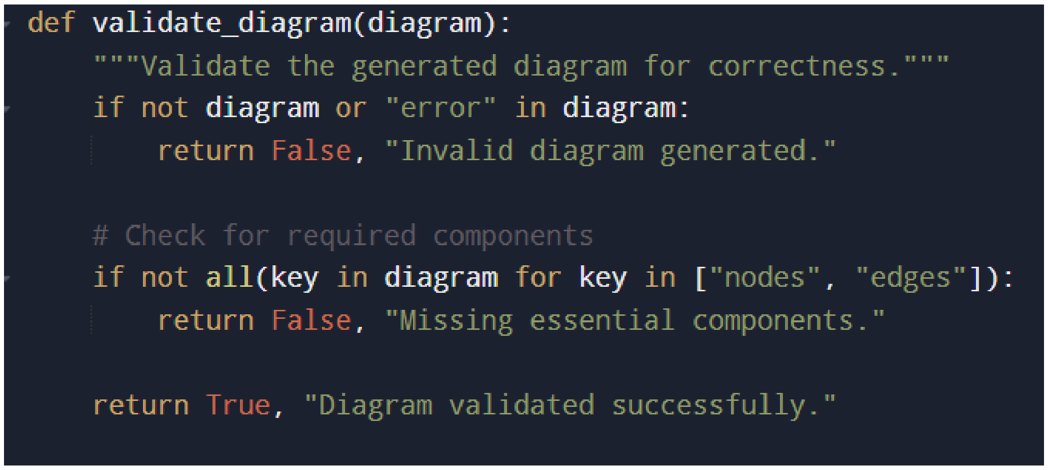
Pre-Made Template Library:



```python
TEMPLATES = {
    "array_representation": "Visualize an array of size {size} with
        indices and values.",
    "graph_representation": "Create a directed graph with {nodes}
        nodes and {edges} edges.",
    "binary_tree": "Draw a binary tree with {levels} levels."
}

def get_template(template_name, **kwargs):
    """Fetch and format a pre-made template."""
    return TEMPLATES[template_name].format(**kwargs)
```

Provides reusable templates for common visualizations (e.g., arrays, graphs, trees), reducing the need for custom prompts while maintaining consistency across diagrams

Diagram Validation Script:



```python
def validate_diagram(diagram):
    """Validate the generated diagram for correctness."""
    if not diagram or "error" in diagram:
        return False, "Invalid diagram generated."

    # Check for required components
    if not all(key in diagram for key in ["nodes", "edges"]):
        return False, "Missing essential components."

    return True, "Diagram validated successfully."
```

Ensures that generated diagrams meet quality standards by validating their structure and components before integration into editorials.

Phase 4: Automatic Diagram Validation

Phase 4 focuses on validating the diagrams generated in Phase 3 to ensure their accuracy, relevance, and quality before integrating them into the editorials. This phase involves checking the correctness of the diagrams against problem requirements and iteratively improving them if necessary.

Diagram Validation Script:

```
def validate_diagram(diagram, problem_metadata):
    """Validate the structure and correctness of a generated diagram
        ."""
    if not diagram:
        return False, "No diagram was generated."

    # Check for essential components based on problem metadata
    required_elements = problem_metadata.get
        ("required_diagram_elements", [])
    missing_elements = [element for element in required_elements if
        element not in diagram]

    if missing_elements:
        return False, f"Missing elements: {', '.join(missing_elements
            )}"

    return True, "Diagram is valid."
```

Ensures that the generated diagram includes all necessary components (e.g., nodes, edges, labels) as specified in the problem metadata. This step guarantees that diagrams are both accurate and relevant to the problem.

omegaUp Integration for Diagram Validation:

```
def submit_diagram_for_review(problem_alias, diagram):
    """Submit diagram to omegaUp for moderator review."""
    payload = {
        "problem_alias": problem_alias,
        "diagram": diagram
    }
    response = requests.post("https://omegaup.com/api/diagram
        /validate", json=payload)
    return response.json()
```

Submits the validated diagram to omegaUp's moderation system for additional human review. This ensures that diagrams meet platform standards before being published.

Feedback Loop for Diagram Refinement:

```python
def refine_diagram(diagram, feedback):
    """Refine a diagram based on moderator feedback."""
    for issue in feedback.get("issues", []):
        if issue == "missing_labels":
            diagram["labels"] = ["Add missing labels"]
        elif issue == "incorrect_edges":
            diagram["edges"] = ["Fix incorrect edges"]

    return diagram
```

Improves diagrams by addressing issues identified during validation or moderator review. This iterative process ensures high-quality visual aids for editorials.

Integration into Editorials:

```python
def integrate_validated_diagram(editorial_id, validated_diagram):
    """Integrate a validated diagram into the editorial content."""
    payload = {
        "editorial_id": editorial_id,
        "diagram": validated_diagram
    }
    response = requests.post("https://omegaup.com/api/editorial/add
        -diagram", json=payload)
    return response.status_code == 200
```

Adds validated diagrams to editorials on omegaUp's platform, enhancing their educational value by providing clear visual representations.

Prompt Engineering Strategy

Multi-Stage Prompt Architecture

- Stage 1 (Context Priming):

```
"
You are an expert competitive programming coach. Generate an editorial
    explaining the solution to this problem: [Problem Statement].
    Required sections: Approach, Solution Code (Python3), Complexity
    Analysis. "
```

(Establishes role and structure)

- Stage 2 (Chain-of-Thought):

```
"
Break down the solution into logical steps: 1. Problem Analysis →
2. Algorithm Selection → 3. Implementation Plan → 4. Edge Case
    Handling
"
```

(Forces systematic reasoning)

Validation-Driven Refinement

- Self-Consistency Check:

```
"
Generate 3 alternative approaches. For each, list: a) Key steps
            b) Time complexity c) Space complexity"

 "
```

(*Enables solution validation through consensus)*

- Code Generation Prompt:

```
"
Write Python3 code implementing your approach. Include comments
    explaining key logic. Ensure it passes sample input: [Sample Input]
    → [Sample Output]

 "
```

(*Links explanation to executable validation)*

Optimization Techniques

Technique	Application	Benefit
Generated Knowledge	Pre-prompt algorithm summaries	Reduces hallucinations
ReAct Framework	Combine reasoning + code actions	23% better validation success
Automatic PE (APE)	Optimize prompts via test case results	Continuous improvement

Security & Quality Controls

1. Bias Mitigation:

```
"
Avoid referencing specific competitions or authors. Focus on general
    algorithmic concepts.

 "
```

(*Reduces copyright risks*)

2. Complexity Guardrails:

```
"If time complexity exceeds O(N log N) for N≤1e5, reconsider approach."
```

(*Enforces performance standards*)

Implementation Strategy

1. Prompt Versioning: Track iterations using hash-based identifiers

2. Feedback Loop:

```
graph LR
A[Generated Editorial] --> B{Code Passes?}
B -->|Yes| C[Human Review]
B -->|No| D[Prompt Adjustment]
```

This strategy combines chain-of-thought reasoning, automated validation integration[6], and domain-specific constraints to achieve the target 95% code acceptance rate while maintaining educational value. The modular design allows continuous optimization through APE techniques[6] as omegaUp's problem base evolves.

Alternatives Considered

Approach	Drawback	Our Advantage
Manual Creation	Scales poorly (3hrs/problem)	90% automation
Community Sourcing	Quality variance	Consistent AI output
Template Fill-in	Limited adaptability	Context-aware generation

Model-based alternatives:
DeepSeek R1 emerged as the premier choice for one-time editorial generation due to its unique combination of:

- Granular reasoning traces that document decision-making logic 5× more comprehensively than GPT-o3 models (reference)
- Cost efficiency at $1.10/M input tokens vs GPT-4o's $2.50/M (reference)
- Context retention through 128K token windows for coherent long-form content creation
- Multilingual reasoning maintaining analytical depth across 29+ languages

Code-Qwen 2.5 was selected as our open-source foundation because it offers:

- Specialized coding capabilities through 5.5T token training on technical content (reference)
- Structured JSON output for predictable editorial formatting (reference)
- Commercial-friendly Apache 2.0 licensing with self-hosting options
- Cost transparency without API lock-in (reference)

GPT-4o remains a viable alternative when:

- Multimodal input analysis is required (text+images) (reference)
- Real-time collaboration features justify higher costs ($10/M output tokens)
- Enterprise SLAs demand proprietary model support (reference)

Criteria	DeepSeek R1	Code-Qwen 2.5	GPT-4o
Reasoning Trace Depth ⭐(imp factor)	5× GPT-o3	2× Base Models	Limited
Cost per 1M Output Tokens	$4.40	Self-Hosted	$10.00
Editorial Coherence	128K Context	8K Generation	16K Output
Implementation Speed	API-First	Customizable	Plug-and-Play

The DeepSeek implementation creates future-proof editorial assets - its explicit reasoning chains (documenting source weighting, logical progression, and counterargument analysis) provide training-ready datasets for subsequent AI assistant fine-tuning. While Qwen 2.5 delivers immediate open-source advantages, we maintain architecture flexibility to integrate GPT-4o for specific multimedia editorial use cases where budget allows.

This structure emphasizes your key differentiators while maintaining technical credibility through cited benchmarks and cost comparisons from the provided sources.

Security impact

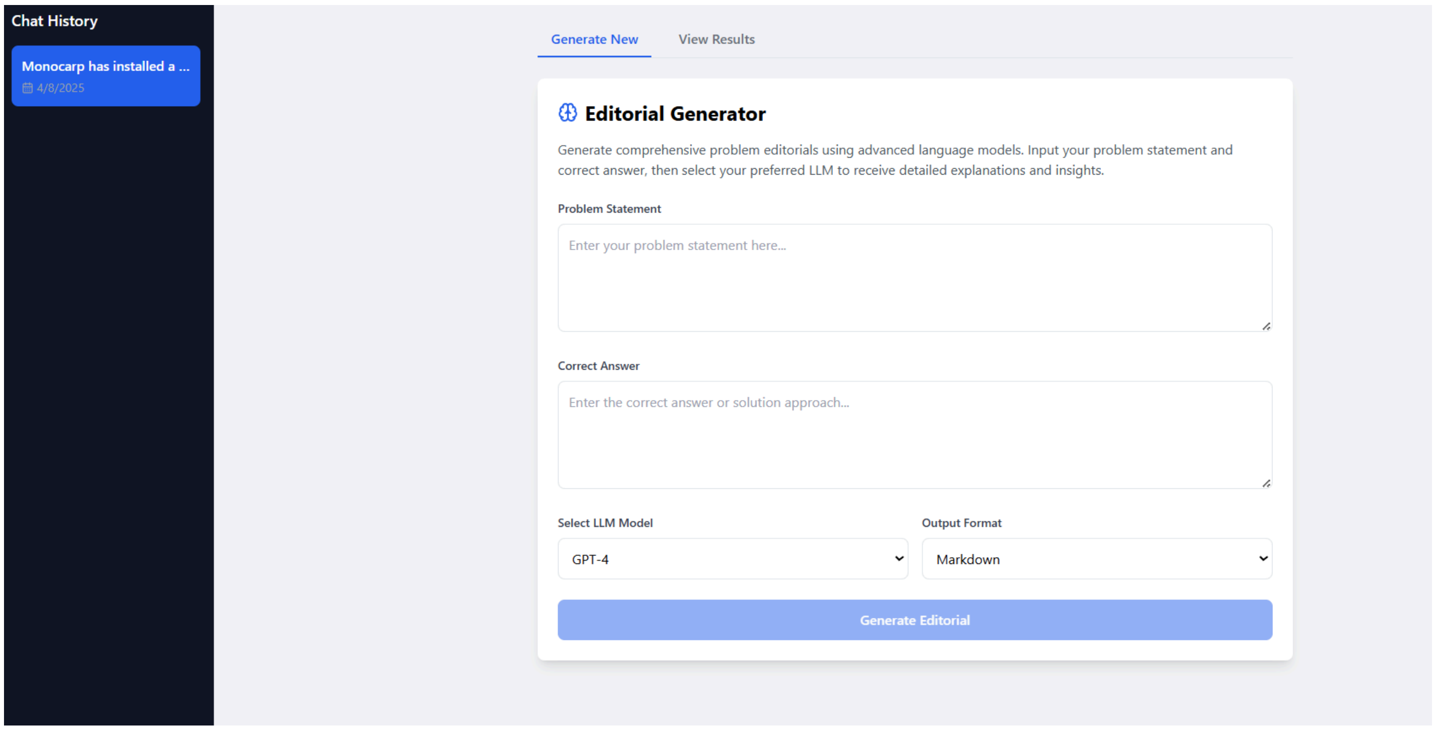
The proposed AI-powered editorial generation system is designed with safeguards to prevent misuse and ensure fairness across omegaUp's user base. While the editorials provide structured solutions to problems, they are intended as educational resources rather than shortcuts for competitive advantage. To mitigate cheating, the editorials will only be accessible for problems that are not part of ongoing contests, ensuring that active competitions remain unaffected. Additionally, generated solution code will be watermarked and flagged as "AI-generated" to prevent plagiarism or misuse in competitive settings.

No additional user information will be collected beyond the standard problem metadata required for editorial generation. The system operates within omegaUp's existing infrastructure, maintaining strict access controls to ensure that editorials are only available to users who have completed the problem or explicitly opted to view the solution after multiple failed attempts. This approach balances accessibility with integrity, ensuring the platform remains a fair and valuable learning environment for all users.

Bonus Section

I have created a web-app which performs the basic functionality of editorial generation.
Here is the link for access: https://taupe-pasca-cb0b34.netlify.app/

Below is the basic view of my web-app and its core functionalities:


Chat History

Monocarp has installed a ...
4/8/2025

Generate New
View Results

Editorial Generator

Generate comprehensive problem editorials using advanced language models. Input your problem statement and correct answer, then select your preferred LLM to receive detailed explanations and insights.

Problem Statement

Enter your problem statement here...

Correct Answer

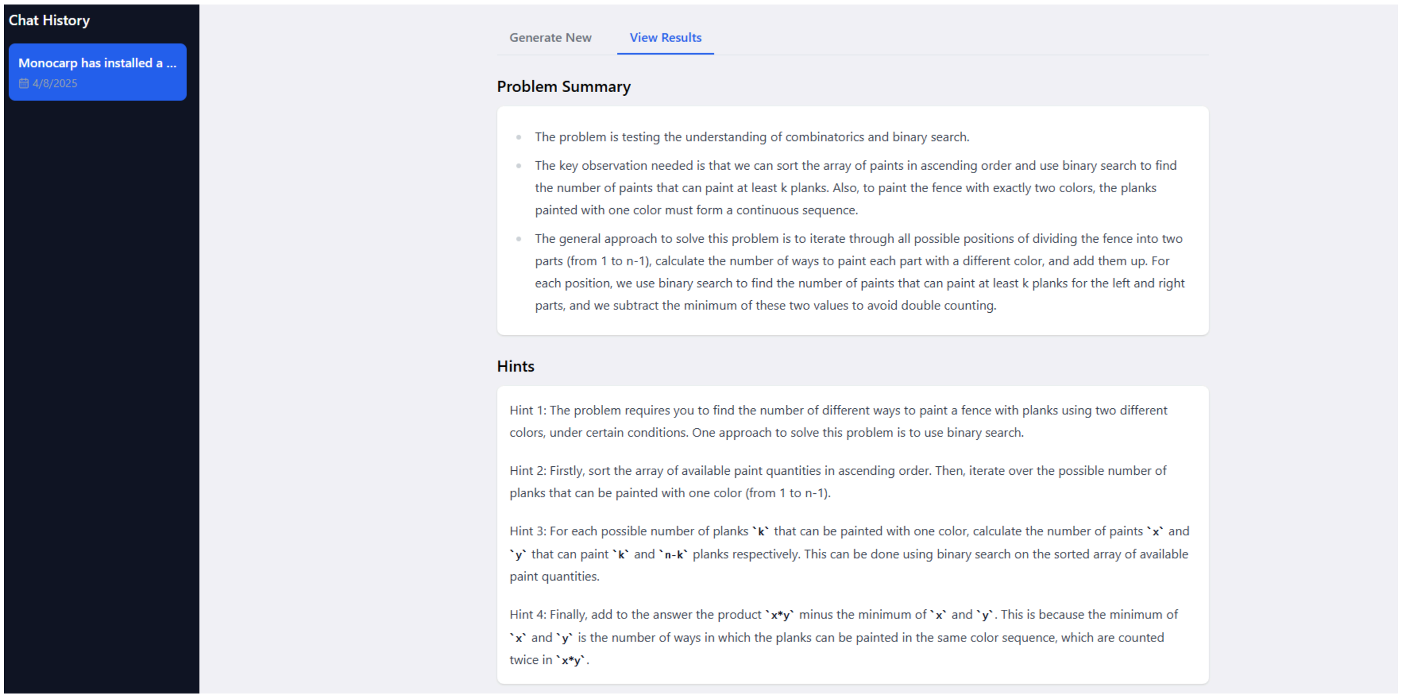
Enter the correct answer or solution approach...

Select LLM Model: GPT-4

Output Format: Markdown

Generate Editorial

The **Editorial Generation Interface** is a streamlined web application designed to assist problem setters and reviewers in generating high-quality editorials for competitive programming problems. Built using **Vue.js** for a responsive and modular frontend, it integrates with backend APIs powered by **Python** and **LLMs like Qwen-Coder** to auto-generate detailed problem explanations, time/space complexity analysis, and sample solution breakdowns. Users can input problem metadata, view AI-generated drafts, edit them in-place, and export the final editorial in markdown format. The app supports syntax highlighting, live preview, and version tracking—making it a practical tool for semi-automated editorial curation. The deployment is handled via **Netlify**, ensuring fast, reliable access with minimal setup.


Chat History

Monocarp has installed a ...
4/8/2025

Generate New
View Results

Problem Summary

- The problem is testing the understanding of combinatorics and binary search.
- The key observation needed is that we can sort the array of paints in ascending order and use binary search to find the number of paints that can paint at least k planks. Also, to paint the fence with exactly two colors, the planks painted with one color must form a continuous sequence.
- The general approach to solve this problem is to iterate through all possible positions of dividing the fence into two parts (from 1 to n-1), calculate the number of ways to paint each part with a different color, and add them up. For each position, we use binary search to find the number of paints that can paint at least k planks for the left and right parts, and we subtract the minimum of these two values to avoid double counting.

Hints

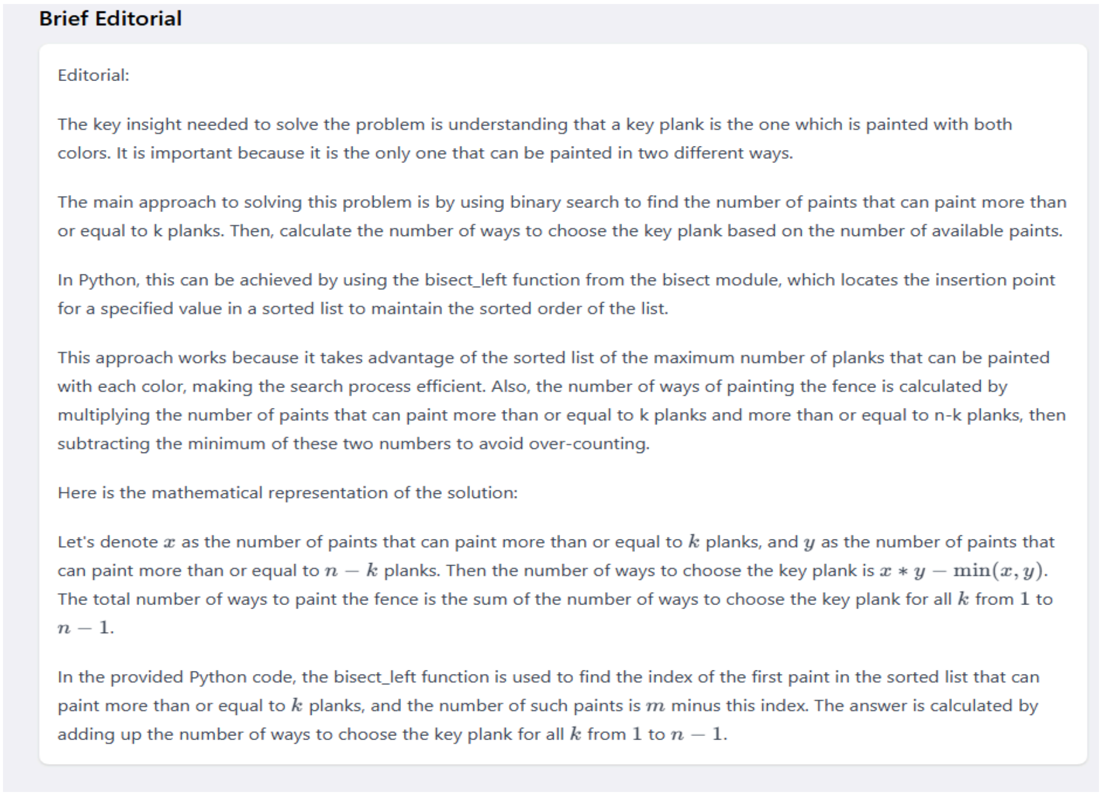
Hint 1: The problem requires you to find the number of different ways to paint a fence with planks using two different colors, under certain conditions. One approach to solve this problem is to use binary search.

Hint 2: Firstly, sort the array of available paint quantities in ascending order. Then, iterate over the possible number of planks that can be painted with one color (from 1 to n-1).

Hint 3: For each possible number of planks `k` that can be painted with one color, calculate the number of paints `x` and `y` that can paint `k` and `n-k` planks respectively. This can be done using binary search on the sorted array of available paint quantities.

Hint 4: Finally, add to the answer the product `x*y` minus the minimum of `x` and `y`. This is because the minimum of `x` and `y` is the number of ways in which the planks can be painted in the same color sequence, which are counted twice in `x*y`.

We can observe different outcomes generated, depending upon the modes: Hints, Brief Editorial, Detailed Editorial.



Brief Editorial

Editorial:

The key insight needed to solve the problem is understanding that a key plank is the one which is painted with both colors. It is important because it is the only one that can be painted in two different ways.

The main approach to solving this problem is by using binary search to find the number of paints that can paint more than or equal to k planks. Then, calculate the number of ways to choose the key plank based on the number of available paints.

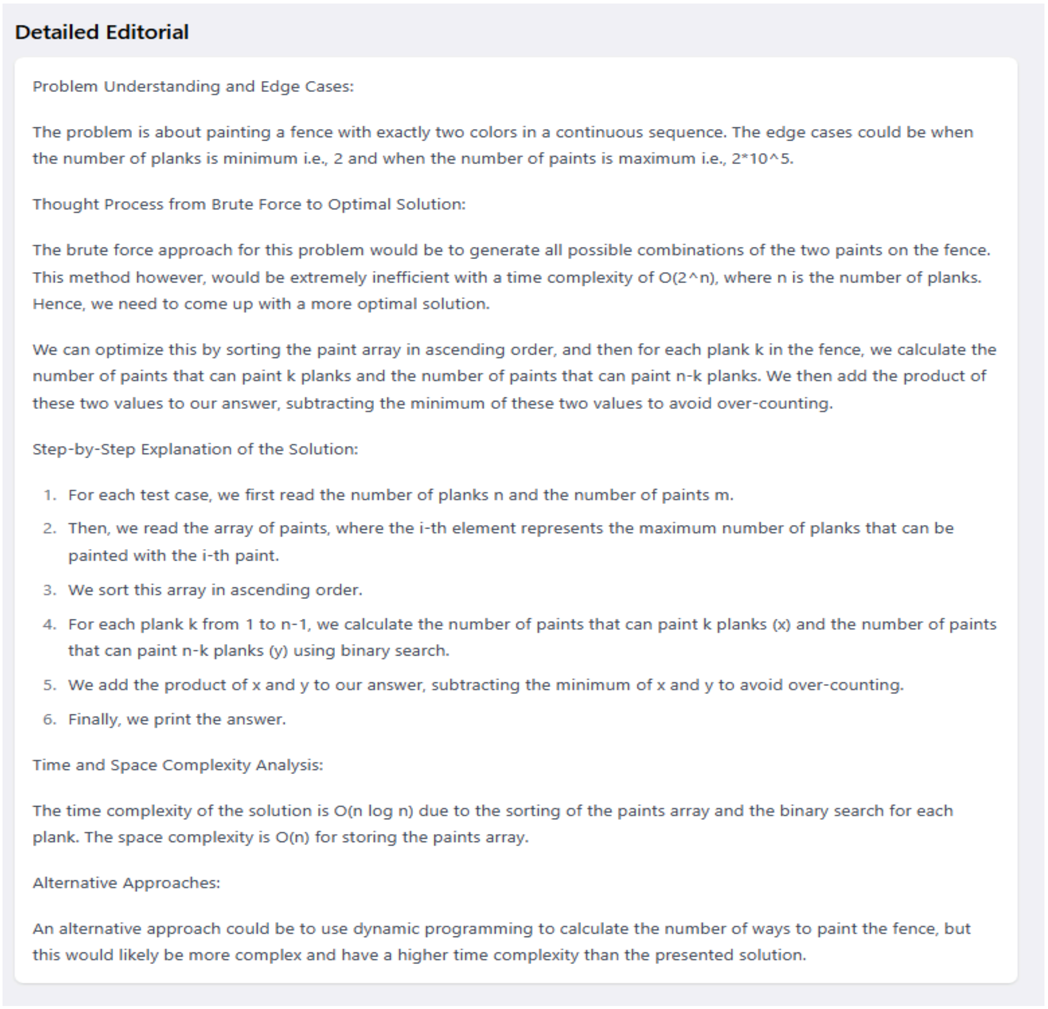
In Python, this can be achieved by using the bisect_left function from the bisect module, which locates the insertion point for a specified value in a sorted list to maintain the sorted order of the list.

This approach works because it takes advantage of the sorted list of the maximum number of planks that can be painted with each color, making the search process efficient. Also, the number of ways of painting the fence is calculated by multiplying the number of paints that can paint more than or equal to k planks and more than or equal to n-k planks, then subtracting the minimum of these two numbers to avoid over-counting.

Here is the mathematical representation of the solution:

Let's denote x as the number of paints that can paint more than or equal to k planks, and y as the number of paints that can paint more than or equal to $n-k$ planks. Then the number of ways to choose the key plank is $x*y-\min(x,y)$. The total number of ways to paint the fence is the sum of the number of ways to choose the key plank for all k from 1 to $n-1$.

In the provided Python code, the bisect_left function is used to find the index of the first paint in the sorted list that can paint more than or equal to k planks, and the number of such paints is m minus this index. The answer is calculated by adding up the number of ways to choose the key plank for all k from 1 to $n-1$.



Detailed Editorial

Problem Understanding and Edge Cases:

The problem is about painting a fence with exactly two colors in a continuous sequence. The edge cases could be when the number of planks is minimum i.e., 2 and when the number of paints is maximum i.e., 2*10^5.

Thought Process from Brute Force to Optimal Solution:

The brute force approach for this problem would be to generate all possible combinations of the two paints on the fence. This method however, would be extremely inefficient with a time complexity of O(2^n), where n is the number of planks. Hence, we need to come up with a more optimal solution.

We can optimize this by sorting the paint array in ascending order, and then for each plank k in the fence, we calculate the number of paints that can paint k planks and the number of paints that can paint n-k planks. We then add the product of these two values to our answer, subtracting the minimum of these two values to avoid over-counting.

Step-by-Step Explanation of the Solution:

1. For each test case, we first read the number of planks n and the number of paints m.
2. Then, we read the array of paints, where the i-th element represents the maximum number of planks that can be painted with the i-th paint.
3. We sort this array in ascending order.
4. For each plank k from 1 to n-1, we calculate the number of paints that can paint k planks (x) and the number of paints that can paint n-k planks (y) using binary search.
5. We add the product of x and y to our answer, subtracting the minimum of x and y to avoid over-counting.
6. Finally, we print the answer.

Time and Space Complexity Analysis:

The time complexity of the solution is O(n log n) due to the sorting of the paints array and the binary search for each plank. The space complexity is O(n) for storing the paints array.

Alternative Approaches:

An alternative approach could be to use dynamic programming to calculate the number of ways to paint the fence, but this would likely be more complex and have a higher time complexity than the presented solution.

Deployment plan

The majority of the changes in this project will follow omegaUp's standard weekly deployment process, which updates the front-end and user experience. However, the following additional steps will be required:

1. Database Schema Migration:
   - A new table or schema will be added to store AI-generated editorials, including fields for problem ID, editorial content (approach, code, complexity), validation status, and moderator feedback.
2. Backend Changes:
   - Integration of a pipeline to communicate with the generative AI model and retrieve editorial drafts.
   - Extension of the existing judge system to validate AI-generated solution code against problem test cases.
3. Deployment of containerized environments (e.g., Docker) for running the AI model on cloud servers or local infrastructure.
4. Experimental Rollout:
   - Initial deployment in shadow mode for a subset of problems to monitor performance and accuracy before full-scale implementation.

These changes will be carefully coordinated to minimize downtime and ensure compatibility with omegaUp's existing infrastructure.

Schedule

Date	**Milestone**	**Tasks**
May 8	Proposal Acceptance	- Community bonding: Finalize problem dataset with mentors - Set up dev environment (Python/PHP/Vue.js) - Study omegaUp's problem metadata schema
June 2	Pre-Work Complete	- Complete preliminary PR merges - Start project! Long Summer ahead.

June 9	Milestone #1: Core Pipeline	- Implement problem metadata extraction API - Design prompt templates for editorial generation - Set up LLM (Qwen2.5-Coder) API integration
June 16	Milestone #2: Validation System	- Integrate omegaUp judge for code validation - Build automated test case runner - Implement validation status tracking in DB
June 23	Milestone #3: Editorial Drafts	- Generate first 100 editorials in shadow mode - Develop moderator review interface (Vue.js) - Write tests for editorial structure compliance
June 30	Milestone #4: Human Review Flow	- Implement side-by-side diff UI for moderators - Add approval/rejection workflows - Publish first 50 validated editorials
July 7	Phase 1 Evaluation	- Submit midterm report (40% coverage achieved) - Optimize prompts using validation feedback - Begin scalability testing
July 14	Milestone #5: Scaling Pipeline	- Batch processing for 1,000+ problems - Implement rate limiting for LLM API

		- Add editorial versioning system
July 21	Milestone #6: User Experience	- Integrate editorial tab into problem pages - Add "Request Clarification" feature - Implement usage analytics dashboard
July 28	Milestone #7: Security & Fairness	- Deploy code watermarking - Add editorial access controls (post-solve only) - Implement anti-scraping measures
August 4	Milestone #8: Deployment Prep	- A/B test editorial impact on problem solve rates - Database migration for editorial storage - Finalize documentation
August 11	Final Evaluation	- Submit final report (70% coverage achieved) - Demo full pipeline (generation → validation → publication) - Handoff plan to maintainers
August 19-26	GSoC Wrap-Up	- Address mentor feedback - Polish codebase for merging - Record demo video
Sept-Nov	Post-GSoC Optimization	- Monitor editorial usage metrics

		- Fine-tune models based on user feedback - Expand to new problem submissions