Dear Reviewers,

In this attachment, we present detailed explanations and examples to support our responses in the response letter.

Reviewer#A-Q1: Manual Filtering

To further clarify our filtering process, two annotators independently filtered code samples based on three criteria, i.e., *Style Consistency*, *Functional Correctness*, and *Implementation Conciseness*. Below are detailed explanations of each criterion:

1. Style consistency refers to the code samples align with the ground truth in terms of naming conventions, commenting style, code structure, and other aspects related to code style.

2. Functional correctness refers to code samples that pass the initial test cases but are found to be functionally incorrect upon manual inspection. For instance, among the code samples we collected, there are cases containing only a "pass" statement or just a TODO comment.

3. Implementation conciseness refers to code that includes extra elements not relevant to the implementation requirements. For example, in some of the code samples we collected, certain variables are defined but remain unused throughout the entirety of the code sample.

The specific process we followed for filtering is as follows:

1. Two annotators independently applied the criteria of (1) Style Consistency, (2) Functional Correctness, and (3) Implementation Conciseness in sequence.
2. After completing the filtering for a specific criterion, the annotators cross-checked the filtered code samples to ensure consistency. Any discrepancies found were discussed to reach a consensus.
3. If the annotators could not resolve discrepancies through discussion, a third annotator was introduced to facilitate the resolution. Finally, disagreements were resolved through a majority voting mechanism.

Reviewer#B-C2: Examples in Sections

We use the following two examples to illustrate that "Understanding the differences in coding styles between LLMs and human developers is crucial, as it directly impacts code readability, maintainability". Each example consists of Code-A (human-written code) and Code-B (model-generated code). While both Code-A and Code-B are functionally correct and semantically similar, they differ in coding style.

Example 1 illustrates that "Understanding the differences in coding styles between LLMs and human developers is crucial, as it directly impacts code readability." The differences in coding style between Code-A and Code-B are reflected in the following aspects:

1. **Naming conventions**: Code-A uses semantically meaningful variable names, such as log_pattern and user_id, while Code-B employs vague names like results and m, which lack clear semantics.
2. **Commenting style**: Code-A includes comments for each step to explain specific functionalities, whereas Code-B lacks any comments.

3. **Code organization and formatting**: Code-A separates code blocks with blank lines, presenting a more structured and visually clear layout, while Code-B does not.

   In summary, Code-A demonstrates more readability compared to Code-B.

```
# Code-A(human-written code)
def parse_log_and_count_actions(log_file_path: str) -> dict[str, dict[str, int]]:
    """
    Parse a log file to count the number of actions performed by each user.

    Args:
        log_file_path (str): The path to the log file.

    Returns:
        dict[str, dict[str, int]]: A dictionary where the keys are user IDs and
        the values are dictionaries with action types as keys and their counts as values.
    """
    # Initialize a dictionary to store results
    user_actions = defaultdict(lambda: defaultdict(int))

    # Define a regex pattern to extract user IDs and actions
    log_pattern = re.compile(r"USER: (\w+).*ACTION: (\w+)")

    try:
        # Read the log file line by line
        with open(log_file_path, 'r') as log_file:
            for line in log_file:
                match = log_pattern.search(line)
                if match:
                    user_id, action = match.groups()
                    user_actions[user_id][action] += 1

    except FileNotFoundError:
        print(f"Error: The file '{log_file_path}' does not exist.")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")

    # Convert defaultdict to a regular dictionary for final output
    return {user: dict(actions) for user, actions in user_actions.items()}

# Code-B(CodeLlama 7B-generated code)
def parse_log_and_count_actions(log_file_path: str) -> dict[str, dict[str, int]]:
    """
    Parse a log file to count the number of actions performed by each user.

    Args:
```

```
        log_file_path (str): The path to the log file.

    Returns:
        dict[str, dict[str, int]]: A dictionary where the keys are user IDs and
        the values are dictionaries with action types as keys and their counts as values.
    """
    results = defaultdict(lambda: defaultdict(int))
    try:
        with open(log_file_path, 'r') as f:
            for line in f:
                m = re.search(r"USER: (\w+).*ACTION: (\w+)", line)
                if m:
                    results[m.group(1)][m.group(2)] += 1
    except:
        print("Error reading file or invalid input.")
    return {k: dict(v) for k, v in results.items()}
```

Example 2 illustrates that "Understanding the differences in coding styles between LLMs and human developers is crucial, as it directly impacts code readability, maintainability".

Compared to Code-B, Code-A demonstrates better readability through the use of meaningful function names and clear logical segment. It organizes distinct tasks, such as load_config, initialize_module, and generate_report, into separate functions. This modular design not only enhances clarity but also makes the code easier to extend and maintain in the future. In contrast, Code-B combines all logic within the main function. This approach lacks organization and uses less descriptive naming, which impacts readability and makes the code harder to maintain.

```
# Code-A(human-written code)
def initialize_system(config_path: str, report_path: str) -> Dict[str, Union[str, Dict]]:
    """
    Initialize the system based on a configuration file and generate a detailed status report.

    Args:
        config_path (str): Path to the configuration file in JSON format.
        report_path (str): Path to save the initialization report.

    Returns:
        Dict[str, Union[str, Dict]]: A dictionary containing the overall status and module-specific details.
    """

    def load_config(file_path: str) -> Dict:
        """Load and validate the configuration file."""
        if not os.path.exists(file_path):
            raise FileNotFoundError(f"Configuration file does not exist: {file_path}")
        try:
            with open(file_path, 'r') as file:
                config = json.load(file)
```

```
                if "modules" not in config or not isinstance(config["modules"], dict):
                    raise ValueError("Invalid configuration structure: 'modules' key is missing or invalid.")
                return config
            except json.JSONDecodeError:
                raise ValueError(f"Invalid JSON format in configuration file: {file_path}")

    def initialize_module(name: str, settings: Dict) -> Dict[str, Union[str, float]]:
        """Initialize a single module, tracking its status and execution time."""
        start_time = time.time()
        try:
            logging.info(f"Starting initialization for module '{name}' with settings: {settings}")

            # Check if the module is enabled
            if not settings.get("enabled", True):
                logging.info(f"Module '{name}' is disabled.")
                return {"status": "disabled", "time": 0.0}

            # Verify critical modules have dependencies
            if settings.get("critical", False):
                if "dependency" not in settings:
                    raise ValueError(f"Module '{name}' is critical but lacks a required dependency.")
                dependency = settings["dependency"]
                logging.info(f"Module '{name}' depends on '{dependency}'. Checking availability...")

            # Perform actual initialization logic
            if name == "database":
                db_settings = settings.get("config", {})
                connection = psycopg2.connect(
                    dbname=db_settings.get("dbname", "example_db"),
                    user=db_settings.get("user", "admin"),
                    password=db_settings.get("password", "password"),
                    host=db_settings.get("host", "localhost"),
                    port=db_settings.get("port", 5432)
                )
                logging.info(f"Connected to database '{db_settings.get('dbname', 'example_db')}'
successfully.")
                connection.close()

            elif name == "cache":
                cache_settings = settings.get("config", {})
                redis_client = Redis(
                    host=cache_settings.get("host", "localhost"),
                    port=cache_settings.get("port", 6379),
                    decode_responses=True
```

```
                    )
                    redis_client.ping()
                    logging.info(f"Connected to Redis cache at {cache_settings.get('host', 'localhost')}
successfully.")

                else:
                    logging.info(f"Module '{name}' does not require specialized setup.")

                return {"status": "initialized", "time": time.time() - start_time}
            except Exception as e:
                logging.error(f"Error initializing module '{name}': {e}")
                return {"status": "failed", "time": time.time() - start_time}

        def generate_report(module_status: Dict[str, Dict[str, Union[str, float]]], report_path: str) -> None:
            """Generate a detailed report summarizing the initialization process."""
            try:
                with open(report_path, 'w') as file:
                    file.write("System Initialization Report\n")
                    file.write("=" * 50 + "\n")
                    for module, status_info in module_status.items():
                        file.write(f"Module: {module}\n")
                        for key, value in status_info.items():
                            file.write(f"    {key}: {value}\n")
                        file.write("\n")
                logging.info(f"Initialization report saved to {report_path}")
            except Exception as e:
                logging.error(f"Failed to write the report: {e}")
                raise

        # Main process
        try:
            config = load_config(config_path)
        except Exception as e:

            logging.error(f"Failed to load configuration: {e}")
            return {"error": str(e)}

        module_status = {}
        for module_name, module_settings in config["modules"].items():
            module_status[module_name] = initialize_module(module_name, module_settings)

        try:
            generate_report(module_status, report_path)
        except Exception as e:
```

```
            logging.error(f"Failed to generate report: {e}")

        return {"overall_status": "completed", "module_details": module_status}

# Code-B(DeepSeekCoder 1.3B-generated code)
def initialize_system(config_path: str, report_path: str) -> Dict[str, Union[str, Dict]]:
    """
    Initialize the system based on a configuration file and generate a detailed status report.

    Args:
        config_path (str): Path to the configuration file in JSON format.
        report_path (str): Path to save the initialization report.

    Returns:
        Dict[str, Union[str, Dict]]: A dictionary containing the overall status and module-specific details.
    """
    module_status = {}
    try:
        if not os.path.exists(config_path):
            raise FileNotFoundError(f"Configuration file : {config_path}")
        with open(config_path, 'r') as f:
            config = json.load(f)
    except Exception as e:
        logging.error(f"Failed to load configuration: {e}")
        return {"error": str(e)}

    for name, settings in config["modules"].items():
        start_time = time.time()
        try:
            if not settings.get("enabled", True):
                module_status[name] = {"status": "disabled", "time": 0.0}
                continue
            if settings.get("critical", False):
                if "dependency" not in settings:
                    raise ValueError(f"Module '{name}' lacks a required dependency.")
                dependency = settings["dependency"]
                logging.info(f"Module '{name}' depends on '{dependency}'. ")
            if name == "database":
                db_settings = settings.get("config", {})
                try:
                    connection = psycopg2.connect(
                        dbname=db_settings.get("dbname", "example_db"),
                        user=db_settings.get("user", "admin"),
                        password=db_settings.get("password", "password"),
```

```
                        host=db_settings.get("host", "localhost"),
                        port=db_settings.get("port", 5432)
                    )
                    connection.close()
                except Exception as e:
                    logging.error(f"Database connection failed: {e}")
                    raise
            elif name == "cache":
                cache_settings = settings.get("config", {})
                try:
                    redis_client = Redis(
                        host=cache_settings.get("host", "localhost"),
                        port=cache_settings.get("port", 6379),
                        decode_responses=True
                    )
                    redis_client.ping()
                except Exception as e:
                    logging.error(f"Redis connection failed: {e}")
                    raise
            else:
                logging.info(f"Module '{name}' does not require setup.")
            module_status[name] = {"status": "initialized", "time": time.time() - start_time}
        except Exception as e:
            logging.error(f"Error initializing   '{name}': {e}")
            module_status[name] = {"status": "failed", "time": time.time() - start_time}

    try:
        with open(report_path, 'w') as f:
            f.write("System_Initialization_Report\n")
            for module, status_info in module_status.items():
                f.write(f"Module: {module}\n")
                for k, v in status_info.items():
                    f.write(f"{k}: {v}\n")
                f.write("\n")
        logging.info(f"Initialization report saved to {report_path}")
    except Exception as report_e:
        raise

    return {"overall_status": "completed", "module_details": module_status}
```

Reviewer#C-Q2: Taxonomy Across Programming Languages

We used DeepSeekCoder-6.7B to execute 230 Java tasks in CoderEval, generated 10 code samples for each task, and obtained a total of 2,300 code samples. Among them, 604 code

samples passed all test cases for each task. We then merged identical code samples to reduce the analysis effort, resulting in 396 unique samples. Next, we filtered out results that showed no inconsistency in coding style, removing 64 code samples. We also filtered out results that implemented the task incorrectly despite passing unit tests, removing an additional 12 code samples. Furthermore, we filtered out results containing extra code that did not contribute to the function's implementation requirements, removing 1 code sample. Finally, we annotated the remaining 319 code samples.

Reviewer#C-Q4,Reviewer#B-C1: Systematic Criteria for Assessing Readability, Conciseness, and Robustness

We provide a detailed explanation of the scoring criteria for readability, conciseness, and robustness. Drawing inspiration from prior studies[1, 2], we selected some aspects related to readability, conciseness, and robustness for quantitative analysis. The rationale behind selecting these features is their relative simplicity and their intuitive impact on readability, conciseness, and robustness.

Readability:

We evaluated readability through four key aspects:

1. **Naming convention**:
   - Definition:The proportion of semantically meaningful variable names defined within code segments out of the total number of defined variable names.
   - Indicator: A higher proportion indicates better readability.
2. **Comment style**:
   - Definition: The proportion of semantically meaningful block comment and inline comment lines out of the total lines of code.
   - Indicator: A higher proportion indicates better readability.
3. **Code structure and Formatting**:

3.1 Blank lines:

- Definition: The proportion of blank lines to the total lines of code. Using blank lines to separate semantically meaningful code blocks is considered good coding style.
- Indicator: A higher proportion indicates better readability.

3.2 spaces and indentations:

- Definition:The proportion of lines with improper use of spaces or indentations to the total lines of code.
- Indicator: A lower proportion indicates better readability.

4. **Code Complexity**:
   - Definition: Assessment the cyclomatic complexity of the code.
   - Indicator: Lower complexity scores indicates better readability.

Scoring Methodology for Readability: Assume two code samples: Code A (human-written) and Code B (model-generated). Both start with a readability score of 0. For each of the four selected aspects:

- If Code A outperforms Code B, Code A is awarded one point, and Code B receives no points.
- After evaluating all aspects: If Code A's total score is higher than Code B's, readability is recorded as “human better”. If both scores are equal, it is recorded as a “tie”. If Code A's score is lower than Code B's, readability is recorded as “model better”.

Conciseness:

We evaluated conciseness based on one aspect:

1. Lines of Code**:**
   - Definition: The total number of lines of code generated.
   - Rationale**:** Each task in CoderEval involves generating a function. Therefore, if both code samples implement the functionality of this function, a lower number of lines of code indicates greater conciseness.

Scoring Methodology for Conciseness: Both Code A and Code B start with a conciseness score of 0. For the selected aspect:

- If Code A outperforms Code B, Code A is awarded one point.
- After evaluation:If Code A's final score is higher than Code B's, conciseness is recorded as “human better.” If both scores are equal, it is recorded as a “tie.”

  If Code A's score is lower than Code B's, conciseness is recorded as “model better.”

Robustness:

We evaluated robustness based on one aspect:

1. Tolerance mechanisms:
   - Definition: Use if statements or try-except blocks for input validation, runtime validation, etc.

Scoring Methodology for Robustness: Both Code A and Code B start with a robustness score of 0. For each code block:

- If Code A implements tolerance mechanisms (using if statements or try-except blocks) within a specific code block and the corresponding block in Code B does not, Code A is awarded one point, while Code B receives no points.
- After evaluation:If Code A's final score is higher than Code B's, robustness is recorded as “human better.” If both scores are equal, it is recorded as a “tie.” If Code A's score is lower than Code B's, robustness is recorded as “model better.”

[1] Buse R P L, Weimer W R. Learning a metric for code readability[J]. IEEE Transactions on software engineering, 2009, 36(4): 546-558.

[2] Lavazza L, Morasca S, Gatto M. An empirical study on software understandability and its dependence on code characteristics[J]. Empirical Software Engineering, 2023, 28(6): 155.

Reviewer#C-C1: Training datasets

-StarCoder2-7B is trained on an open-source, high-quality code dataset called The Stack v2. The dataset includes source code from GitHub repositories, GitHub Issues, pull requests, etc[1].
-CodeLlama-7B is based on the Llama 2-7B foundation model and is trained on a dataset of 500 billion tokens[2]. These training data include:（1）85% code data, sourced from publicly

available code. (2) 8% natural language related to code, derived from natural language datasets related to code. This dataset contains many discussions about code and code snippets included in natural language questions or answers. (3) 7% natural language, sourced from a general natural language dataset.

-The training dataset of DeepSeek-Coder is composed of 87% source code, 10% English coderelated natural language corpus, and 3% code-unrelated Chinese natural language corpus. The source code originates from GitHub, etc. The English corpus consists of materials from GitHub's Markdown and StackExchange. Chinese corpus consists of high-quality articles[3].

We consider that the training datasets of these Code LLMs share many common components, as their code data is primarily sourced from platforms like GitHub. However, the selection of code data may differ across models, and the proportion of code data to natural language datasets in their training data also varies. These factors could contribute to discrepancies in the coding styles of these Code LLMs.

[1] Lozhkov A, Li R, Allal L B, et al. Starcoder 2 and the stack v2: The next generation[J]. arXiv preprint arXiv:2402.19173, 2024.

[2] Roziere B, Gehring J, Gloeckle F, et al. Code llama: Open foundation models for code[J]. arXiv preprint arXiv:2308.12950, 2023.

[3] Guo D, Zhu Q, Yang D, et al. DeepSeek-Coder: When the Large Language Model Meets Programming--The Rise of Code Intelligence[J]. arXiv preprint arXiv:2401.14196, 2024.