Seeker: Towards Exception Safety Code Generation with Intermediate Language Agents Framework

Yuxuan Chen[1]*, **Xuanming Zhang**[2]*†
[1]Tsinghua University
[2]Uniersity of Wisconsin-Madison
`chenyuxu21@mails.tsinghua.edu.cn, xzhang2846@wisc.edu`

Abstract

In real-world software development, improper or missing exception handling can severely impact the robustness and reliability of code. Exception handling mechanisms require developers to detect, capture, and manage exceptions according to high standards, but many developers struggle with these tasks, leading to fragile code. This problem is particularly evident in open-source projects and impacts the overall quality of the software ecosystem. To address this challenge, we explore the use of large language models (LLMs) to improve exception handling in code. Through extensive analysis, we identify three key issues: *Insensitive Detection of Fragile Code*, *Inaccurate Capture of Exception Block*, and *Distorted Handling Solution*. These problems are widespread across real-world repositories, suggesting that robust exception handling practices are often overlooked or mishandled. In response, we propose **Seeker**, a multi-agent framework inspired by expert developer strategies for exception handling. Seeker uses agents—**S**canner, D**e**tector, Pr**e**dator, Ran**k**er, and Handl**er**—to assist LLMs in detecting, capturing, and resolving exceptions more effectively. Our work is the first systematic study on leveraging LLMs to enhance exception handling practices in real development scenarios, providing valuable insights for future improvements in code reliability. [3] [4]

1 Introduction

In the era of large language models for code generation (code LLMs) such as DeepSeek-Coder [14], Code-Llama [34], and StarCoder [24], ensuring code robustness and security has become paramount alongside functional correctness. Traditional evaluation metrics, like HumanEval [2] which measures the *Pass@k* rate, and repo-level assessments such as CoderEval [44] and DevEval [20], primarily focus on the ability of these models to generate correct and functional code based on natural language descriptions and real-world development scenarios. As functional correctness improves, attention has shifted to addressing defects in LLM-generated code [18, 36, 15, 22]. Notably, KPC [33] and Neurex [1] explored LLM performance in exception handling, highlighting their potential to predict and mitigate risks before vulnerabilities arise.

However, despite advancements in exception detection and handling techniques, there is a significant gap in standardizing exception mechanisms, particularly for custom exceptions and long-tail exception types. Current approaches often define problems narrowly from the perspective of exception handlers, overlooking the potential role of intermediate languages (IL) in managing complex inheritance

*Equal contribution.
†Corresponding author.
[3]Our code is available at `https://anonymous.4open.science/r/Seeker`
[4]CEE for community-contribution is available at `https://common-exception-enumeration.github.io/CEE/`

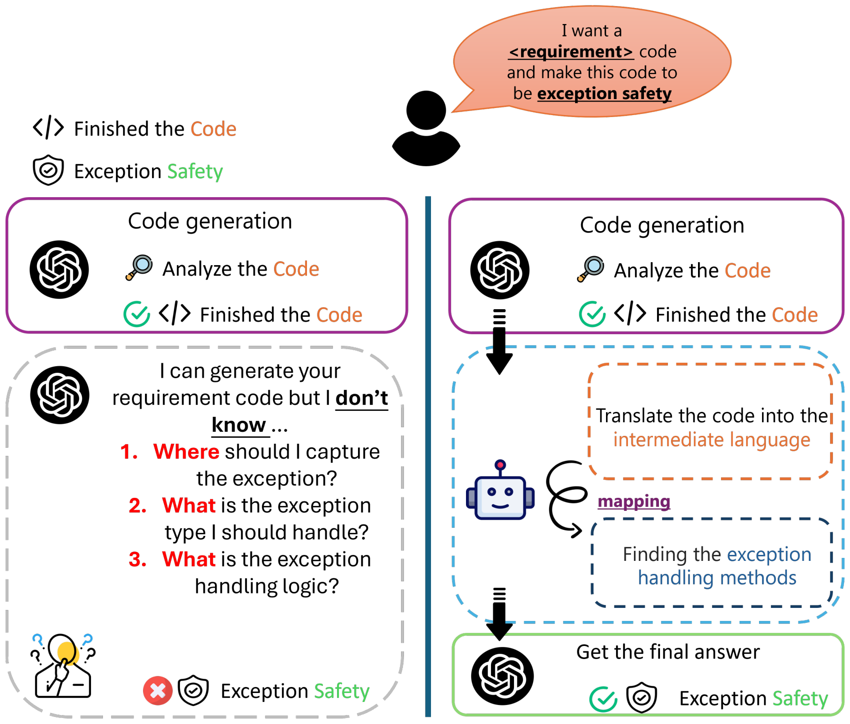
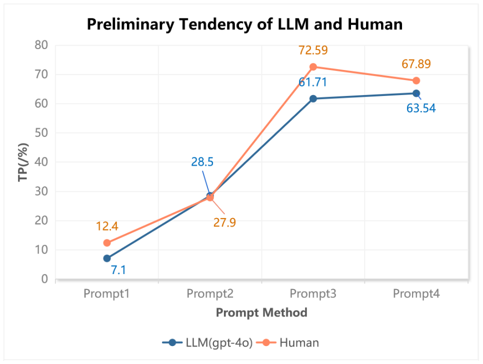

I want a <requirement> code and make this code to be exception safety
Finished the Code
Exception Safety
Code generation
Analyze the Code
Finished the Code
I can generate your requirement code but I don't know ...
1. Where should I capture the exception?
2. What is the exception type I should handle?
3. What is the exception handling logic?
Exception Safety
Code generation
Analyze the Code
Finished the Code
Translate the code into the intermediate language
mapping
Finding the exception handling methods
Get the final answer
Exception Safety

Figure 1: **Overview of the Intermediate Language (IL) agents (Right) Compared with Traditional Code Generation Approaches (Left) in Exception-Safe Code Generation Tasks** The Seeker framework leverages IL agents to perform dynamic analysis, transformation, and optimization of code to ensure robust exception handling. In contrast, traditional approaches often rely on static error-handling routines and lack comprehensive analysis for exception safety.

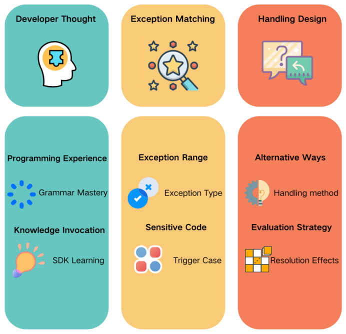
relationships inherent in exception types. We posit that interpretable and generalizable exception handling strategies are critical yet underestimated factors in real-world code development, profoundly impacting both code robustness and the quality of LLM training data. A schematic diagram of IL Agent can be found at Figure 1.

This paper redefines the research question to address overlooked issues in exception handling by developers, specifically: What method can address developer defects in *Insensitive Detection of Fragile Code*, *Inaccurate Capture of Exception Blocks*, and *Distorted Handling Solutions*? The inquiry highlights the importance of exception safety while exploring how intermediate languages can enhance code analysis beyond traditional human capabilities. To investigate this, we introduce four sets of prompts—Coarse-grained Reminding, Fine-grained Reminding, Fine-grained Inspiring, and Fine-grained Guiding. Our experiments validate the effectiveness of fine-grained guiding prompts in improving exception safety, and we align the evaluation with recent repo-level perspectives where possible [18]. Effective exception handling prioritizes capturing specific exception types, which enables more precise error reporting, such as handling *SQLClientInfoException* instead of its superclass *SQLException* [32]. However, achieving this level of specificity is challenging due to the lack of standardized handling paradigms for long-tail or customized exceptions, and the complexity of intricate inheritance structures.

To enhance code robustness by leveraging best exception handling practices, we propose **Seeker**, a framework that decomposes exception handling into five specialized tasks managed by distinct agents: **S**canner: Responsible for scanning the code into manageable unit, D**e**tector: Responsible for detecting the fragile unit, Pr**e**dator: Predate the exception block and capture possible exceptions, Ran**k**er: Sorts exception handling strategies according to certain criteria and selects appropriate exceptions, Handl**er**: Responsible for performing the final exception handling. We develop Common Exception Enumeration (CEE) from trusted external sources, leveraging explainable IL for exception handling to improve detection, capture, and handling processes where LLMs typically underperform. This approach seamlessly integrates with existing code LLMs to produce highly robust code, while CEE facilitates community contributions by aiding developers in understanding optimal exception handling practices.



(a) Preliminary following the four settings and strategies of KPC. The vertical axis represents the evaluation score of human code review.



(b) A schematic diagram of human developers who well-performed in exception handling.

Figure 2: (a) Comparison of LLM and human exception handling performance as prompts evolve from General prompting (Prompt1), to Coarse-grained Knowledge-driven (Prompt2), Fine-grained Knowledge-driven (Prompt3), and Fine-grained Knowledge-driven with explicit handling logic (Prompt4). Results show a clear mitigation effect, where increasingly detailed and context-rich prompts significantly improve handling quality. (b) How expert human developers integrate programming expertise, domain knowledge, fine-grained exception hierarchies, and adaptive strategies to achieve robust exception management.

Addressing the inefficiency of direct retrieval in complex inheritance trees—exemplified by Java exceptions with 433 nodes, 62 branches, and 5 layers—we introduce a deep retrieval-augmented generation (Deep-RAG) algorithm. This algorithm is tailored to handle intricate inheritance relationships by assigning development scenario labels to branches and employing few-sample verification to fine-tune these labels, thereby enhancing retrieval performance and reducing computational overhead. Experimental results indicate that Seeker significantly improves the robustness and exception handling capabilities of LLM-generated code across various tasks.

In summary, our main contributions are:

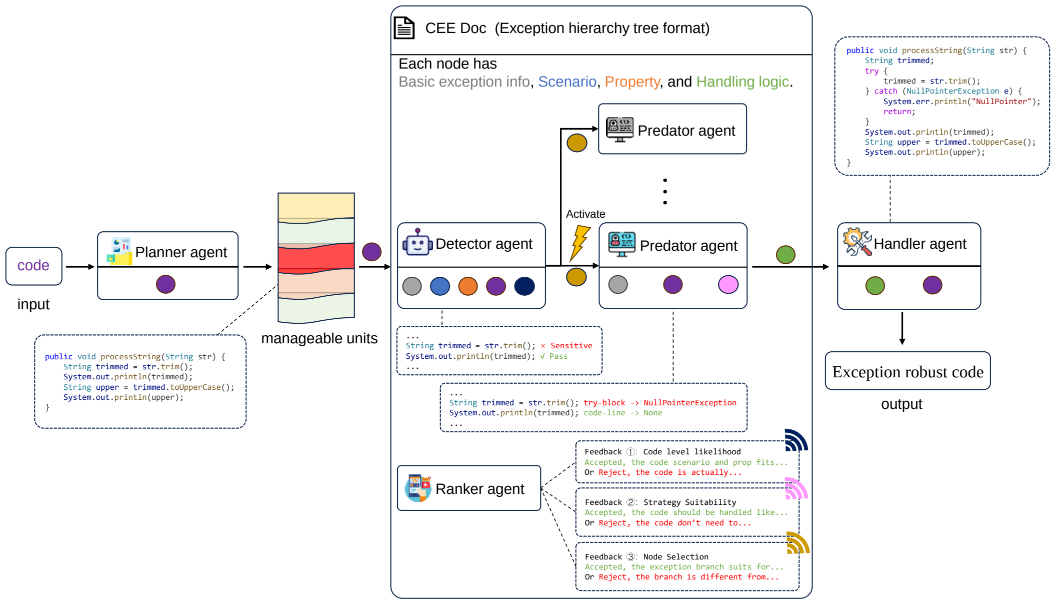
- We highlight the importance of standardization, interpretability, and generalizability in exception handling mechanisms, identifying a critical gap in existing research.
- We propose **Seeker**, which decomposes exception handling into specialized tasks and incorporates CEE (Common Exception Enumeration) to enhance performance.
- We conduct extensive experiments demonstrating that **Seeker** significantly improves code robustness and exception handling performance in LLM-generated code, setting a new state-of-the-art (SOTA) in the field of exception handling.

2 Preliminary

This section explores the effectiveness of Intermediate Language (IL)-based Large Language Models in exception handling compared to senior programmers, focusing on standardization, interpretability, and generalizability. Building on prior findings (Appendix A.1), we investigate whether IL can better mitigate exception-handling challenges. Our analysis includes comparative experiments controlling three key factors: standardization of exception types, interpretability of risk scenarios, and generalization of handling strategies. We use four prompt types (Figure 8 and 9): Coarse-grained Reminding, Fine-grained Reminding, Fine-grained Inspiring, and Fine-grained Guiding.

To ensure a practical evaluation, we drew on prior work on KPC[33], selecting well-maintained codebases and using both manual and automated reviews to extract critical exception-handling code. Senior programmers and the IL then analyzed these filtered segments, documenting their processes. To simulate exception-handling thought processes, we established four prompt pathways that progressively offer more detailed guidance, with results illustrated in Figure 2(a).

Our experiments revealed key insights. Prompts lacking clear guidance were ineffective for both IL and programmers. Normative information on exception types helped programmers recognize



```
public void processString(String str) {
    String trimmed = str.trim();
    System.out.println(trimmed);
    String upper = trimmed.toUpperCase();
    System.out.println(upper);
}
```

```
...
String trimmed = str.trim(); × Sensitive
System.out.println(trimmed); √ Pass
...
```

```
...
String trimmed = str.trim(); try-block -> NullPointerException
System.out.println(trimmed); code-line -> None
...
```

```
Feedback ①: Code level likelihood
Accepted, the code scenario and prop fits...
Or Reject, the code is actually...
```

```
Feedback ②: Strategy Suitability
Accepted, the code should be handled like...
Or Reject, the code don’t need to...
```

```
Feedback ③: Node Selection
Accepted, the exception branch suits for...
Or Reject, the branch is different from...
```

```
public void processString(String str) {
    String trimmed;
    try {
        trimmed = str.trim();
    } catch (NullPointerException e) {
        System.err.println("NullPointer");
        return;
    }
    System.out.println(trimmed);
    String upper = trimmed.toUpperCase();
    System.out.println(upper);
}
```

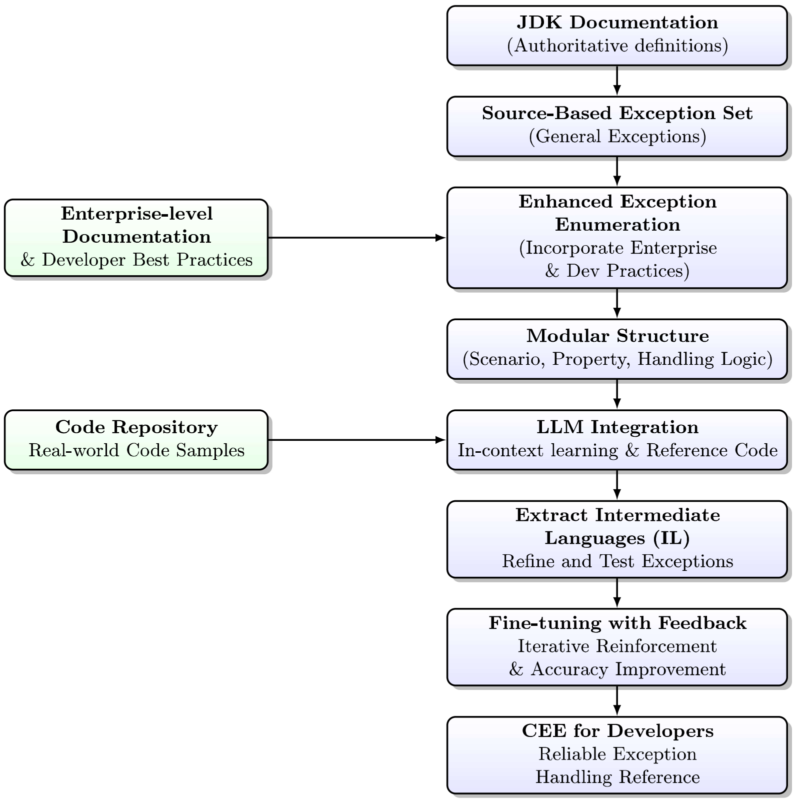
Figure 3: **Comprehensive Workflow of Seeker.** Seeker orchestrates the automated exception handling process through the seamless collaboration of five specialized agents: Scanner, Detector, Predator, Ranker, and Handler. The colored circles within the workflow illustrate the flow of information and interactions among the agents, highlighting how each component activates and contributes to the overall exception handling process. This integrated approach ensures that Seeker delivers highly reliable and maintainable exception handling solutions, significantly improving code robustness and developer productivity.

fragile code but did not significantly improve detection or handling precision. Enhanced scenario interpretability, however, improved understanding and sensitivity to potential fragility, boosting detection accuracy. Generalized handling strategies further improved the analysis of fragility sources, leading to higher-quality exception handling. Together, these enhancements — referred to as the "mitigation effect" — demonstrate that, with proper prompts, IL can match or even surpass senior programmers in exception handling.

These findings inform our proposed *Seeker* method, which integrates external documentation to generate fine-grained guidance prompts. Appendix A.2.1 offers a deeper analysis of the mitigation effect, supported by data and methodological insights. Figure 2(b) illustrates the Chain-of-Thought used by senior developers under Fine-grained Guiding prompts, emphasizing the importance of comprehensive analysis when handling complex exceptions like *BrokenBarrierException* and *AccessControlException*. Our study highlights the potential of IL for reliable code generation and offers a foundation for advancing RAG-based code agents.

3 Methodology

In this section, We present **Seeker**, a designed, IL-agent framework for exception safety. Our framework is not merely an integration of tools. Figure 3 shows the end-to-end workflow. Seeker decomposes exception safety into five capabilities—*Scanner*, *Detector*, *Predator*, *Ranker*, and *Handler*—and couples them with an interpretable intermediate language (IL) and the **Common Exception Enumeration (CEE)** knowledge base. This design targets the three pitfalls summarized from developer practice (Appendix A.2.1): *Insensitive Detection of Fragile Code*, *Inaccurate Capture of Exception Blocks*, and *Distorted Handling Solutions*. We first overview the CEE dependency, then detail each agent and how IL binds detection, retrieval, ranking, and handling. A complete algorithmic view is provided in Appendix A.2.3, with the Deep-RAG retriever specified in Appendix A.2.4 and scalability considerations in Appendix A.3.4.



JDK Documentation
(Authoritative definitions)
Source-Based Exception Set
(General Exceptions)
Enterprise-level Documentation
& Developer Best Practices
Enhanced Exception Enumeration
(Incorporate Enterprise & Dev Practices)
Modular Structure
(Scenario, Property, Handling Logic)
Code Repository
Real-world Code Samples
LLM Integration
In-context learning & Reference Code
Extract Intermediate Languages (IL)
Refine and Test Exceptions
Fine-tuning with Feedback
Iterative Reinforcement
& Accuracy Improvement
CEE for Developers
Reliable Exception
Handling Reference

Figure 4: **An overview of the CEE construction process**. The diagram illustrates how authoritative documentation (JDK), enterprise-level best practices, and real-world code repositories are integrated and refined. Each exception node is enriched with Scenario, Property, and Handling Logic. This framework is further optimized through LLM-based in-context learning and iterative fine-tuning, ultimately providing a reliable, structured reference (CEE) to enhance exception handling in generated code.

3.1 Common Exception Enumeration (CEE)

To enhance reliability in LLM-assisted development, CEE provides a unified, structured repository of exception knowledge. It standardizes how exceptions are described and handled, enabling precise retrieval and fine-grained handling in Seeker. CEE is built from authoritative JDK documentation and enriched with enterprise practices and open-source analyses (Appendix A.2.2).

CEE is hierarchical. Each exception node contains three elements—**Scenario**, **Property**, and **Handling Logic**—which modularize where an exception arises, how it behaves, and how to address it. This modularity supports interpretable retrieval and handling in Seeker.

A schematic of CEE construction is shown in Figure 4. Detailed construction and iterative refinement are provided in Appendix A.2.2.

A more detailed explanation of the construction, iterative refinement processes (including reinforcement learning-based fine-tuning), and community-driven updates is provided in Appendix A.2.2.

3.2 Intermediate Language Agent Framework

Seeker enhances standardization, interpretability, and generalizability by binding agent capabilities via an intermediate language (IL). IL abstracts exception reasoning steps—detection hypotheses, retrieval queries, ranking criteria, and handling templates into explicit, auditable artifacts connected to code spans. This enables: (i) precise exception targeting along inheritance hierarchies; (ii) transparent choice of strategies from CEE; and (iii) reproducible edits. Seeker integrates with any base code LLM but is architected as a standalone framework with clearly separated roles and data flows (Appendix A.2.3).

Scanner *Scanner* segments the codebase into manageable, self-contained units (functions/classes/-files). This process considers volume, dependencies, and requirement relations. This prevents context overload and preserves analyzability for downstream agents.

Detector *Detector* combines static analysis (control/exception flow) with scenario/property matching to surface fragile code. The union improves recall on long-tail and custom exceptions. Detector outputs IL hypotheses with localized spans and candidate branches.

Predator *Predator* performs retrieval over CEE using IL queries composed from code summaries and Detector hypotheses. Our **Deep-RAG** algorithm (Appendix A.2.4) traverses inheritance-aware branches with scenario labels and few-shot verification to refine retrieval depth and improve precision.

Ranker *Ranker* grades exceptions by (likelihood, strategy suitability) using IL criteria and CEE metadata, prioritizing specific, high-impact exceptions. Scores feed back to Detector/Predator for iterative refinement.

Handler *Handler* emits edits using IL-backed templates and logic patterns from CEE, favoring fine-grained catches over generic ones. Outputs include structured diffs and rationale tags, aiding review and maintainability.

Framework The complete agent workflow and data flow are formalized in Appendix A.2.3.

Scalability Considerations To mitigate computational overhead, we designed a high-concurrency interface that maintains constant additional computing time overhead regardless of code volume, provided in Appendix A.3.4.

4 Experiments

In this section, we evaluate the performance of our proposed method, **Seeker**, on the task of exception handling code generation. We aim to answer the following research questions (RQs):

- **RQ1**: How does **Seeker** perform compared to state-of-the-art methods on exception handling code generation tasks?
- **RQ2**: What is the effect of different agents in the **Seeker** framework on the overall performance?
- **RQ3**: How does **Seeker** perform across different evaluation metrics, specifically in terms of code quality and correctness?
- **RQ4**: How does the choice of underlying LLM affect the performance of **Seeker**?
- **RQ5**: What is the impact of integrating domain-specific knowledge (CEE) into **Seeker**?

4.1 Experiment Setup

4.1.1 Datasets

We conduct experiments on a dataset consisting of 750 fragile Java code snippets extracted from real-world projects. These code snippets are selected based on their potential for exception handling improvements, following the rules outlined in Appendix A.3.2.

4.1.2 Baselines

We compare against:

- **General Prompting**: A straightforward approach where the LLM is prompted to generate exception handling code without any specialized framework or additional knowledge.
- **Traditional Retrieval-Augmented Generation (RAG)**: A method that retrieves relevant information from external sources to assist in code generation.
- **KPC** [33]: The state-of-the-art method for exception handling code generation, which leverages knowledge graphs and pattern mining.
- **FuzzyCatch** [29]: A tool for recommending exception handling code for Android Studio based on fuzzy logic.
- **Nexgen** [46]: A neural network pretraining approach for automated exception handling in Java, which predicts try block locations and generates complete catch blocks in relatively high accuracy.

For more analysis of related work, see Appendix B.1.

4.1.3 Evaluation Metrics

To comprehensively assess the effectiveness of our method, we employ six quantitative metrics. Details are given in Appendix A.3.1:

1. **Automated Code Review Score (ACRS)**: Measures the overall adherence to coding standards based on automated code reviews. It provides a percentage indicating how well the code aligns with best practices.
2. **Coverage (COV)**: Assesses the proportion of sensitive code segments successfully detected by the system. It reflects the effectiveness of the method in identifying relevant code segments.
3. **Coverage Pass (COV-P)**: Focuses on the accuracy of detecting try-blocks. This metric evaluates whether the detected try-blocks match the actual regions requiring exception handling, with penalties for over-detection.
4. **Accuracy (ACC)**: Evaluates the correctness of identified exception types. It compares the detected exception types to the actual types, accounting for subclass relationships.
5. **Edit Similarity (ES)**: Measures the text similarity between generated and actual try-catch blocks. A higher similarity indicates better quality in the generated code.
6. **Code Review Score (CRS)**: Assesses the quality of generated try-catch blocks through evaluation by a language model. This provides a binary evaluation of whether the generated code meets best practices.

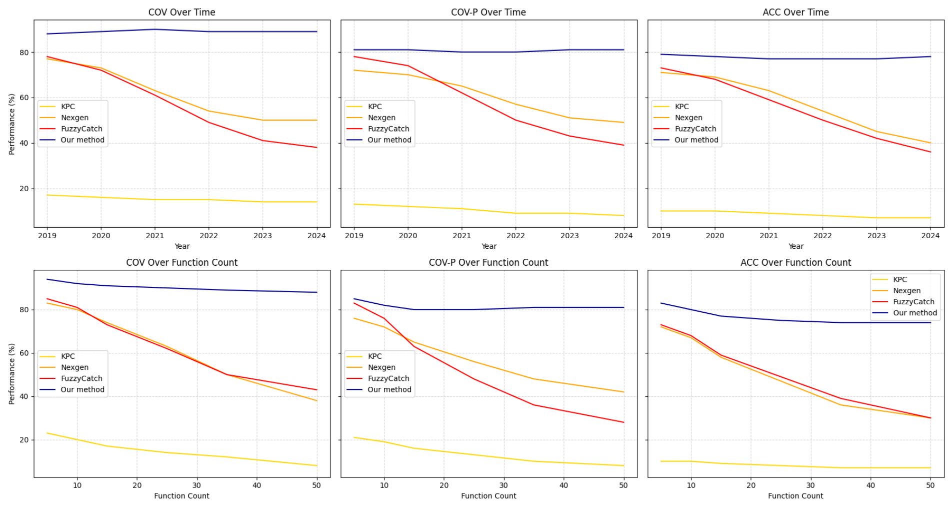


Figure 5: Comparison of Performance Stability Across Baselines and Our Method over Varying Conditions. The top set of curves illustrates the performance metrics over time (2019 to 2024) across different baselines and our method. The bottom set displays performance across increasing function counts.

4.2 RQ1: Performance Comparison with Baselines

We compare the performance of **Seeker** against several baseline methods on the exception handling code generation task. The results are summarized in Table 1. We conducted our tests on Java code from GitHub spanning a period of five years (2019-2024) and averaged the results across different methods to ensure reliability and account for variability in code quality and project types.

As shown in Table 1, **Seeker** outperforms all baselines across metrics:

- **ACRS**: **Seeker** achieves a significantly higher Average Code Review Score (ACRS) of 0.85, indicating superior overall code quality compared to other methods.
- **Coverage (COV) and Coverage Pass (COV-P)**: With COV and COV-P scores of 91% and 81%, respectively, **Seeker** demonstrates exceptional capability in detecting and correctly wrapping sensitive code regions.

Table 1: Comparison of Exception Handling Code Generation Methods

Method	**ACRS**	**COV (%)**	**COV-P (%)**	**ACC (%)**	**ES**	**CRS (%)**
General Prompting	0.21	13	9	8	0.15	24
Traditional RAG	0.35	35	31	29	0.24	31
KPC	0.26	14	11	8	0.17	27
FuzzyCatch	0.47	52	50	43	0.36	48
Nexgen	0.45	56	49	42	0.41	52
Seeker (Ours)	**0.85**	**91**	**81**	**79**	**0.64**	**92**

- **Accuracy (ACC)**: An ACC of 79% reflects **Seeker**'s proficiency in accurately identifying the correct exception types, including recognizing complex subclass relationships.
- **Edit Similarity (ES)**: The ES score of 0.64 indicates that the generated code closely matches the actual exception handling implementations, ensuring minimal discrepancies.
- **Code Review Score (CRS)**: A CRS of 92% confirms that the code generated by **Seeker** is highly regarded during automated and LLM-based code reviews, emphasizing its adherence to best practices.

Depth of Analysis Robust exception handling correlates with overall code quality: high ACRS and CRS reflect adherence to best practices. Seeker's fine-grained catches and IL-auditable decisions improve maintainability and reduce risk, consistent with Appendix A.3.1.

Stability and Robustness Figure 5 illustrates the stability of **Seeker** compared to baseline methods over time and varying code complexities.

Stability Over Time: **Seeker** maintains consistently high performance levels across different time periods, whereas baseline methods exhibit significant variability, particularly in recent years. This stability indicates that **Seeker** is less sensitive to changes in the development environment and can adapt to evolving software trends and requirements.

Performance Across Code Complexity: As the number of functions within test snippets increases, baseline methods show a decline in performance, struggling with higher code complexity. In contrast, **Seeker** sustains its performance across all levels of complexity, demonstrating its ability to handle intricate codebases effectively.

4.3 RQ2: Effect of Different Agents in Seeker

To understand the contribution of each agent within the **Seeker** framework, we conducted an ablation study by systematically removing one agent at a time. The results are presented in Table 2.

Table 2: Ablation Study on the Effect of Different Agents

Configuration	**ACRS**	**COV (%)**	**COV-P (%)**	**ACC (%)**	**ES**	**CRS (%)**
Seeker (Full)	**0.85**	**91**	**81**	**79**	**0.64**	**92**
- Scanner	0.78	85	75	73	0.59	86
- Detector	0.76	63	54	61	0.51	84
- Predator	0.72	61	53	42	0.47	81
- Ranker	0.63	90	79	75	0.49	65
- Handler	0.50	91	81	79	0.34	42

- **Scanner**: Removing the Scanner results in a notable decrease in ACRS and CRS, underscoring its vital role in initial code analysis and overall quality assessment.
- **Detector**: Its absence significantly reduces coverage metrics, highlighting its importance in identifying sensitive code regions.
- **Predator**: The Predator is essential for accurate exception type detection, as evidenced by the sharp decline in ACC and related metrics when it is removed.
- **Ranker**: Without the Ranker, the selection of handling strategies becomes less effective, impacting the edit similarity and code review scores.

- **Handler**: The most substantial drop in CRS occurs without the Handler, indicating its critical role in implementing exception handling correctly.

The ablation study reveals that each agent in the **Seeker** framework contributes uniquely to its overall performance. The interplay between agents ensures comprehensive exception handling, from initial detection to the implementation of robust handling strategies. This synergy not only enhances exception handling but also positively impacts code quality reviews, as robust exception handling is a key indicator of well-structured and maintainable code.

4.4 RQ3: Effect of Underlying Language Model

To evaluate the impact of the underlying LLM on **Seeker**'s performance, we implemented **Seeker** using different models, including various open-source and close-source models. The detailed results are provided in Appendix A.3.5.

Advanced language models like GPT-5, Claude 4.5 Sonnet, DeepSeek R1, and GPT-4o significantly enhance **Seeker**'s performance across all metrics. This improvement suggests that the capabilities of the underlying LLM, such as understanding complex code structures and generating accurate exception handling code, play a crucial role in the overall effectiveness of **Seeker**.

4.5 RQ4: Impact of Domain-Specific Knowledge Integration

To assess the impact of integrating domain-specific knowledge, we compared **Seeker** with and without the inclusion of the CEE. The results are presented in Table 3.

Table 3: Impact of Integrating Common Exception Enumeration (CEE)

Configuration	ACRS	COV (%)	COV-P (%)	ACC (%)	ES	CRS (%)
Seeker	**0.85**	**91**	**81**	**79**	**0.64**	**92**
- CEE	0.38	48	41	32	0.29	46

The inclusion of CEE results in substantial improvements across all metrics. Specifically, ACRS increases from 0.38 to 0.85, and CRS jumps from 46% to 92%. This significant enhancement highlights the importance of domain-specific knowledge in accurately detecting and handling exceptions, thereby improving overall code quality and reliability.

Even without CEE, IL-driven agent collaboration outperforms general prompting, validating the program-analysis design.

4.6 Additional Analysis

Beyond the primary research questions, we conducted additional experiments to evaluate **Seeker**'s performance in generating repository-level code and optimizing code patches for GitHub issues. Detailed results are available in Appendix A.4.

Seeker maintains competitive performance in these real-world scenarios, demonstrating its robustness and applicability. The ability to generate repository-level code and effectively optimize code patches underscores **Seeker**'s versatility and its potential to assist developers in maintaining high-quality codebases.

5 Conclusion

We presented **Seeker**, a designed intermediate-language agent framework for exception safety. Seeker decomposes exception handling into five explicit capabilities bound by IL artifacts and powered by **CEE**, a structured, inheritance-aware knowledge base. Across datasets and baselines, Seeker achieves strong improvements on coverage, accuracy, and review scores, with clear ablations isolating the contribution of each agent. The framework is model-agnostic and readily integrates with modern agentic workflows while remaining interpretable and auditable. Beyond Java, Seeker's IL+CEE design generalizes to other languages and inheritance-rich reasoning tasks (Appendix A.4).

Limitations

Base Model The actual performance of Seeker framework depends on the base model's code understanding and information analysis capabilities. Therefore, in order to achieve good experimental performance, it is necessary to introduce base model with strong general capabilities.

Closed-source Model The good performance of closed-source models may lead to more overhead and privacy leakage issues in enterprise application-level development.

References

[1] Yuchen Cai, Aashish Yadavally, Abhishek Mishra, Genesis Montejo, and Tien N. Nguyen. Programming assistant for exception handling with codebert. In *ICSE*, 2024.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] Clade. https://www.anthropic.com/index/claude-2. 2023.

[4] Caroline Claus and Craig Boutilier. The dynamics of reinforcement learning in cooperative multiagent systems. In *AAAI*, 1998.

[5] Codex. https://openai.com/index/openai-codex/. 2021.

[6] Guilherme B. de Pádua and Weiyi Shang. Revisiting exception handling practices with exception flow analysis. In *SCAM*, 2017.

[7] Dêmora Bruna Cunha de Sousa, Paulo Henrique M. Maia, Lincoln S. Rocha, and Windson Viana. Studying the evolution of exception handling anti-patterns in a long-lived large-scale project. *J. Braz. Comput. Soc.*, 2020.

[8] Yihong Dong, Xue Jiang, Zhi Jin, and Ge Li. Self-collaboration code generation via chatgpt. *ACM Trans. Softw. Eng. Methodol.*, 2023.

[9] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. A reflection on "an exploratory study on exception handling bugs in java programs". In *SANER*, 2020.

[10] GPT-3. https://platform.openai.com/docs/models/gpt-base. 2022.

[11] GPT-3.5. https://platform.openai.com/docs/models/#gpt-3-5-turbo. 2023.

[12] GPT-4. https://platform.openai.com/docs/models/gpt-4. 2023.

[13] GPT-4o. https://platform.openai.com/docs/models/gpt-4o. 2024.

[14] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, et al. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[15] Jingxuan He and Martin T. Vechev. Large language models for code: Security hardening and adversarial testing. In *CCS*, 2023.

[16] Kai Huang, Jian Zhang, Xiangxin Meng, and Yang Liu. Template-Guided Program Repair in the Era of Large Language Models . In *ICSE*, 2025.

[17] Bart Jacobs and Frank Piessens. Failboxes: Provably safe exception handling. In *ECOOP*, 2009.

[18] Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *ICLR*, 2024.

[19] Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. In *NeurIPS*, 2023.

[20] Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, et al. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories. In *ACL(Findings)*, 2024.

[21] Junjie Li, Fazle Rabbi, Cheng Cheng, Aseem Sangalay, Yuan Tian, and Jinqiu Yang. An exploratory study on fine-tuning large language models for secure code generation. *arXiv preprint 2408.09078*, 2024.

[22] Junjie Li, Aseem Sangalay, Cheng Cheng, Yuan Tian, and Jinqiu Yang. Fine tuning large language model for secure code generation. In *FORGE*, 2024.

[23] Kaixuan Li, Jian Zhang, Sen Chen, Han Liu, Yang Liu, and Yixiang Chen. Patchfinder: A two-phase approach to security patch tracing for disclosed vulnerabilities in open-source software. In *ISSTA*, 2024.

[24] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *TMLR*, 2023.

[25] Xiangwei Li, Xiaoning Ren, Yinxing Xue, Zhenchang Xing, and Jiamou Sun. Prediction of vulnerability characteristics based on vulnerability description and prompt learning. In *SANER*, 2023.

[26] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *ICLR*, 2024.

[27] Marvin Minsky. The emotion machine: Commonsense thinking, artificial intelligence, and the future of the human mind. *Simon and Schuster*, 2007.

[28] Suman Nakshatri, Maithri Hegde, and Sahithi Thandra. Analysis of exception handling patterns in java projects: an empirical study. In *MSR*, 2016.

[29] Tam Nguyen, Phong Vu, and Tung Nguyen. Code recommendation for exception handling. In *ESEC/FSE*, 2020.

[30] Tam Nguyen, Phong Vu, and Tung Nguyen. Code recommendation for exception handling. In *ESEC/FSE*, 2020.

[31] OpenAI o1. https://platform.openai.com/docs/models/o1. 2024.

[32] Haidar Osman, Andrei Chis, Jakob Schaerer, Mohammad Ghafari, and Oscar Nierstrasz. On the evolution of exception usage in java projects. In *SANER*, 2017.

[33] Xiaoxue Ren, Xinyuan Ye, Dehai Zhao, Zhenchang Xing, and Xiaohu Yang. From misuse to mastery: Enhancing code generation with knowledge-driven AI chaining. In *ASE*, 2023.

[34] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[35] Yongliang Shen, Kaitao Song, Xu Tan, Dongsheng Li, Weiming Lu, and Yueting Zhuang. Hugginggpt: Solving AI tasks with chatgpt and its friends in huggingface. *NeurIPS*, 2023.

[36] Mohammed Latif Siddiq and Joanna C. S. Santos. Securityeval dataset: Mining vulnerability examples to evaluate machine learning-based code generation techniques. In *MSR4P&S*, 2022.

[37] Stephen W. Smoliar. Marvin minsky, the society of mind. *Artif. Intell.*, 48(3):349–370, 1991.

[38] Wei Tao, Yucheng Zhou, Wenqiang Zhang, and Yu Cheng. MAGIS: llm-based multi-agent framework for github issue resolution. *arXiv preprint 2403.17927*, 2024.

[39] Yanlin Wang, Tianyue Jiang, Mingwei Liu, Jiachi Chen, and Zibin Zheng. Beyond functional correctness: Investigating coding style inconsistencies in large language models. *arXiv preprint 2407.00456*, 2024.

[40] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In *OOPSLA*, 2004.

[41] Xin-Cheng Wen, Yupan Chen, Cuiyun Gao, Hongyu Zhang, Jie M. Zhang, and Qing Liao. Vulnerability detection with graph simplification and enhanced graph representation learning. In *ICSE*, 2023.

[42] Chenfei Wu, Shengming Yin, Weizhen Qi, Xiaodong Wang, Zecheng Tang, and Nan Duan. Visual chatgpt: Talking, drawing and editing with visual foundation models. *arXiv preprint 2303.04671*, 2023.

[43] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering, 2024.

[44] Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *ICSE*, 2024.

[45] Hao Zhang, Ji Luo, Mengze Hu, Jun Yan, Jian Zhang, and Zongyan Qiu. Detecting exception handling bugs in C++ programs. In *ICSE*, 2023.

[46] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. Learning to handle exceptions. In *ASE*, 2020.

[47] Kechi Zhang, Jia Li, Ge Li, Xianjie Shi, and Zhi Jin. Codeagent: Enhancing code generation with tool-integrated agent systems for real-world repo-level coding challenges. In *ACL*, 2024.

[48] Yifan Zhang, Yang Yuan, and Andrew Chi-Chih Yao. On the diagram of thought. *arXiv preprint 2409.10038*, 2024.

[49] Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric P. Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. In *NeurIPS*, 2023.

[50] Jian Zhou, Hongyu Zhang, and David Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, 2012.

A Appendix

A.1 A Revisit of Human Empiricals

Over the years, there have been numerous empirical studies and practical discussions on exception handling, but what is common is that exception handling has been repeatedly emphasized as an important mechanism directly related to code robustness. Exception handling is a necessary and powerful mechanism to distinguish error handling code from normal code, so that the software can do its best to run in a normal state[28]. Exception mechanism ensures that unexpected errors do not damage the stability or security of the system, prevents resource leakage, ensures data integrity, and ensures that the program still runs correctly when unforeseen errors occur[40]. In addition, exception handling also involves solving potential errors in the program flow, which can mitigate or eliminate defects that may cause program failure or unpredictable behavior[17].

Although the exception mechanism is an important solution to code robustness, developers have always shown difficulties in dealing with it due to its complex inheritance relationship and processing methods. Various programming language projects show a long-tail distribution of exception types when facing exception handling, which means that developers may only have a simple understanding of the frequently occurring exception types[6]. However, according to section1, good exception practices rely on developers to perform fine-grained specific capturing. Multi-pattern effect of exception handling is also considerable[30]. For example, even for peer code, capturing different exception types will play different maintenance functions, so exception handling is often not generalized or single-mapped. These complex exception mechanism practice skills have high requirements for developers' programming literacy. Previous study manually reviewed and counted the exception handling of a large number of open source projects, and believed that up to 62.91% of the exception handling blocks have violations such as capturing general exceptions and destructive wrapping[7]. This seriously violates the starting point of the exception mechanism. We also emphasize the urgent need and importance of automated exception handling suggestion tools[6].

The failure of human developers in the exception handling mechanism seriously affects the quality of LLM's code training data [15], which further leads to LLM's inability to understand the usage skills of maintenance functions [39]. To solve the above problems, we first proposed $Seeker - Java$ for the Java language. This is because the Java language has a more urgent need for exception handling and is completely mapped to the robustness of Java programs. As a fully object-oriented language, Java's exception handling is more complex than other languages, and it has a higher degree of integration into language structures[9]. Therefore, Java projects are more seriously troubled by exception handling bugs. In addition, Java relies heavily on exceptions as a mechanism for handling exceptional events. In contrast, other languages may use different methods or have less strict exception handling mechanisms. It is worth mentioning that $Seeker$'s collaborative solution based on an inherent multi-agent framework plus an external knowledge base, they can quickly migrate multiple languages by maintaining documents for different languages. We will also maintain $Seeker-Python$ and $Seeker - C\#$ in the future to provide robustness guarantees for the development of more programming languages.

A.2 Method Details

A.2.1 Rules of Good Practice

In this section, we introduce four progressively refined prompting strategies to guide developers—both humans and LLMs—toward stable and generalizable exception handling practices. As shown in Figure6, our goal is to align developer performance with recognized best practices, gradually helping them move from vague awareness to well-structured and generalizable handling strategies. We term the vulnerable code segments as *Fraile code*, emphasizing that these code fragments are particularly susceptible to runtime exceptions and error propagation if not addressed properly.

Specifically, we present: *Coarse-grained Reminding prompting*, *Fine-grained Reminding prompting*, *Fine-grained Inspiring prompting*, and *Fine-grained Guiding prompting*. Each prompt setting provides incrementally richer contextual information and guidance about exceptions, their types, and their handling strategies. This incremental approach is designed to gradually improve the developer's in-context learning process, ensuring a more accurate understanding of exceptions and a stable, repeatable handling methodology that can be applied across various development scenarios.

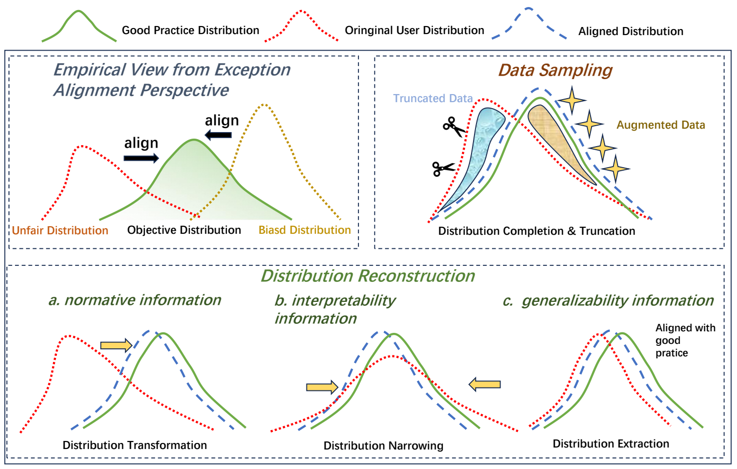

Good Practice Distribution
Oringinal User Distribution
Aligned Distribution
Empirical View from Exception Alignment Perspective
align
align
Unfair Distribution
Objective Distribution
Biasd Distribution
Data Sampling
Truncated Data
Augmented Data
Distribution Completion & Truncation
Distribution Reconstruction
a. normative information
b. interpretability information
c. generalizability information
Aligned with good pratice
Distribution Transformation
Distribution Narrowing
Distribution Extraction

Figure 6: Aligning developers' exception handling from biased, user-oriented practices to industry-standard "good practice" distributions through iterative data refinement. Distribution truncation, augmentation, and reconstruction guide a progression from coarse-grained reminders to fine-grained, scenario-specific guidance—closing the gap between current human methods and stable, high-quality exception handling.

Coarse-grained Reminding prompting simply alerts developers to "pay attention to potential exceptions," nudging them to identify and handle Fraile code based on their own experience. As shown in Figures 2(a), 8, and 9, while such a reminder can make both human and LLM developers more aware of exceptions, it does not significantly improve the precision of identifying Fraile code. Related studies [33] have summarized this phenomenon as a series of bad practices—such as *Incorrect exception handling*—where the developer's initial intuition is insufficient for robust code improvement.

Fine-grained Reminding prompting focuses on specific exception types in the Fraile code scenario, prompting developers to consider their sources and standardize their handling. Although this level of detail encourages consultation of external documents or examples, these references are often too abstract, non-standardized, or insufficiently generalizable. Consequently, developers may still catch exceptions inaccurately, failing to fundamentally mitigate underlying risks. Prior work has identified patterns such as the *Abuse of try-catch* as common pitfalls even under such guidance.

Fine-grained Inspiring prompting goes a step further by providing a code-level scenario analysis for the Fraile code. Here, intuitive and interpretable natural language guidance helps developers gain deeper insight and analytical capability. While recent studies have shown that such prompting can lead to stable good practices for standalone function-level Fraile code, the complexity and dependency chains of real-world scenarios still pose a challenge. As noted by [45], even experienced developers can introduce errors in exception handling within complex projects.

Fine-grained Guiding prompting finally offers a generalized handling strategy for the identified exceptions. Building upon the previously established stable exception detection performance, developers are now equipped with a structured, generalizable approach. Prior recommendations [6] strongly advocate for employing such strategies, as developers struggle to perform high-quality optimization without fully understanding the nature of the exception type. By establishing a systematic framework for addressing Fraile code, developers—whether human or model-based—are more likely to achieve high-quality exception handling in diverse and complex contexts.

In essence, these four prompt settings represent a progression from broad, coarse-grained reminders to fine-grained, scenario-specific guidance and ultimately to generalizable handling strategies. By gradually improving the developer's understanding of exceptions and providing actionable insights, we enhance the robustness and quality of the code they produce. This approach highlights the significant influence of prompt specification on LLM-driven code generation and encourages further research into how different levels of guidance can surpass traditional human practice, better aligning the final implementation with recognized exception handling best practices.

Notably, many programming languages offer three main ways to handle exceptions: declaring them with the `throws` keyword in the method signature, throwing them with the `throw` keyword inside the method body, and capturing them via `try-catch` blocks. Existing work [28] points out that simply using `throws` at the method signature may not reflect the true runtime conditions, as these exceptions propagate up the call stack until caught. Similarly, exceptions thrown within a method body must eventually be handled by the caller via `try-catch`. Thus, `try-catch` blocks represent the most practical and common approach. In our method, we adopt this third technique as the best practice, integrating it into our prompting strategies to guide developers toward stable and high-quality exception handling.

A.2.2 Detailed CEE Construction and Refinement Process

This appendix provides the complete rationale, methodologies, and iterative steps undertaken to build the CEE in detail. It includes:

Comprehensive Documentation from the JDK: We begin with 433 exception nodes drawn from the official Java documentation, spanning 62 branches across five hierarchical layers. Each node is anchored to a standard JDK-defined exception class or interface.

Integration of Enterprise Practices: To enhance the practicality of the CEE, we incorporate patterns and insights drawn from enterprise-level Java development documentation and established open-source projects. By analyzing widely adopted handling practices, logging standards, and fallback mechanisms, we align the CEE with real-world coding scenarios, ensuring that recommended handling logic is both credible and effective.

Granular Structuring via Scenario, Property, and Handling Logic: For each exception node, we record:

- **Scenario**: Typical contexts or operations where the exception may arise.
- **Property**: Attributes such as exception severity, root causes, and environmental factors.
- **Handling Logic**: Recommended strategies, including try-catch patterns, logging techniques, and fallback operations.

This granular detailing enables developers and LLMs to map from a given exception scenario to an appropriate handling strategy more accurately.

Reinforcement Learning-Based Fine-Tuning: We employ a testing framework that uses RL-based fine-tuning to improve the mapping between exceptions and handling logic. Over multiple iterations, false positives and negatives in suggested handling methods are identified and rectified, ensuring that the CEE remains both precise and adaptive.

Iterative Refinement and Community Input: The CEE is treated as a living document, continuously refined through user feedback and community contributions. Over time, newly identified exception patterns, handling techniques, or corrections are integrated, ensuring that the CEE evolves alongside prevailing development practices and tooling.

By following these guidelines and iterative enhancements, the CEE aims to serve as a robust, trusted reference that enhances both human and LLM-driven exception management in Java.

A.2.3 Seeker Framework

Algorithm 1: Seeker Framework

```
Input: Codebase $C$
Output: Optimized code $C'$ with robust exception handling
1  Segment the codebase $C$ into manageable units $U = \{u_1, u_2, \ldots, u_N\}$;
2  foreach code segment $u_i$ in $C$ do
3      if (length of $u_i$ is within predefined limit) and (function nesting level is low) and (logical flow is clear)
        then
4          Add $u_i$ to $U$;
5  Initialize optimized units $U' = \{\}$;
6  foreach unit $u_i$ in $U$ do
       // Detection Phase
7      Initialize potential exception set $E_i = \{\}$;
8      Use the Detector agent to analyze unit $u_i$;
9      In parallel do { // Static Analysis
10     Generate control flow graph $CFG_i$ and exception propagation graph $EPG_i$ for $u_i$;
11     Identify sensitive code segments $S_i^{\text{static}} = \{s_{i1}^{\text{static}}, s_{i2}^{\text{static}}, \ldots\}$ in $u_i$;
       // Scenario and Property Matching
12     Perform scenario and property matching on $u_i$;
13     Identify sensitive code segments $S_i^{\text{match}} = \{s_{i1}^{\text{match}}, s_{i2}^{\text{match}}, \ldots\}$ in $u_i$;
14     } Combine sensitive code segments: $S_i = S_i^{\text{static}} \cup S_i^{\text{match}}$;
15     foreach segment $s_{ij}$ in $S_i$ do
16         Detect potential exception branches $E_{bij}$ in $s_{ij}$;
17         $E_{bi} \leftarrow E_{bi} \cup E_{bij}$;
       // Retrieval Phase
18     Use the Predator agent to retrieve fragile code and try-catch blocks;
19     Summarize unit $u_i$ at the function level to obtain code summary $F_i$;
20     Perform Deep-RAG using $F_i$ and exception branches $E_{bi}$, get exception nodes $E_{ni}$;
21     Mapping relevant exception handling strategies $H_i = \{h_{i1}, h_{i2}, \ldots\}$ from CEE;
       // Ranking Phase
22     Use the Ranker agent to assign grades to exceptions in $E_{ni}$;
23     foreach exception $e_{ik}$ in $E_{ni}$ do
24         Calculate exception likelihood score $l_{ik}$ based on $e_{ik}$ attribute and impact;
25         Calculate suitability score $u_{ik}$ of handling strategy $h_{ik}$;
26         Compute overall grade $g_{ik} = \alpha \cdot l_{ik} + \beta \cdot u_{ik}$;
27     Rank exceptions in $E_{ni}$ based on grades $g_{ik}$ in descending order to get ranked list $E'_{ni}$;
       // Handling Phase
28     Use the Handler agent to generate optimized code $u'_i$;
29     foreach exception $e_{ik}$ of $E'_{ni}$ if $g_{ik} > \gamma$ do
30         Mapping handling strategy $h_{ik}$ from $H_i$;
31         Apply $h_{ik}$ to code segment(s) related to $e_{ik}$ in $u_i$;
32     $U' \leftarrow U' \cup \{u'_i\}$;
33 Combine optimized units $U'$ to produce the final optimized code $C'$;
```

A.2.4 Deep-RAG Algorithm

Algorithm 2: Deep Retrieval-Augmented Generation (Deep-RAG)

```
Input: Knowledge hierarchy tree $T$, unit summary $F_i$, detected queries $Q_i$, environment context $Env$
Output: Relevant information retrievals $R_i$
1  Initialize relevant knowledge branches set $B = \{\}$;
2  Assign knowledge scenario labels $L = \{l_1, l_2, \dots\}$ to branches of $T$;
3  foreach query $q_{ik}$ in $Q_i$ do
4      Identify branches $B_{ik}$ in $T$ related to $q_{ik}$ based on labels $L$;
5      $B \leftarrow B \cup B_{ik}$;
6  foreach branch $b_m$ in $B$ do
       // Verification Step
7      Select few-sample document examples $X_m = \{x_{m1}, x_{m2}, \dots\}$ associated with branch $b_m$;
8      foreach example $x_{mj}$ in $X_m$ do
9          Perform query matching to obtain pass rate $p_{mj}$ and capture accuracy $a_{mj}$;
10         if $p_{mj}$ or $a_{mj}$ below threshold $\theta$ then
11             Record failure pattern $fp_{mj}$ based on $Env$;
12             Update environment context $Env$ with $fp_{mj}$;
13     Compute average pass rate $\bar{p}_m$ and accuracy $\bar{a}_m$ for branch $b_m$;
14     if $\bar{p}_m$ or $\bar{a}_m$ below threshold $\theta$ then
15         Fine-tune labels $L$ for branch $b_m$ based on aggregated feedback from $Env$;
16 Initialize information retrievals set $R_i = \{\}$;
17 foreach branch $b_m$ in $B$ do
18     Select depth level $D$ for node evaluation;
19     for $d = 1$ to $D$ do
20         foreach node $n_{ml}$ at depth $d$ in branch $b_m$ do
21             Evaluate relevance score $r_{ml}$ to summary $F_i$ and queries $Q_i$;
22             if $r_{ml} > \delta$ then
23                 Retrieve information $r_{ml}$ from knowledge base;
24                 $R_i \leftarrow R_i \cup \{r_{ml}\}$;
```

In the Deep-RAG algorithm, we assign development scenario labels to each branch of the exception inheritance tree based on their inheritance relationships, enabling the identification of branches that may correspond to specific information of fragile code segments. Acting as an intelligent agent, the algorithm interacts dynamically with its operational environment by leveraging feedback from detection pass rates and capture accuracies obtained during the few-shot verification step. This feedback mechanism allows the system to refine the granularity and descriptions of the scenario labels through regularization prompts derived from failed samples. As a result, Deep-RAG can accurately identify the risk scenarios where fragile codes are located and the corresponding knowledge branches that are activated. Subsequently, the algorithm selectively performs node evaluations on these branches by depth, thereby enhancing retrieval performance and optimizing computational overhead. Additionally, we have designed the algorithm interface to be highly general, ensuring its applicability across a wide range of RAG scenarios beyond exception handling. This generality allows Deep-RAG to support diverse applications, as further detailed in Appendix A.4. By integrating environmental feedback and maintaining a flexible, agent-based interaction model, Deep-RAG not only improves retrieval accuracy and efficiency but also adapts seamlessly to various domains and information retrieval tasks, demonstrating its versatility and robustness in enhancing the performance of large language models.

A.3 Experimental Details

A.3.1 Metrics

1. **Automated Code Review Score (ACRS)**: This metric evaluates the overall quality of the generated code in terms of adherence to coding standards and best practices, based on an automated code review model.

$$\text{ACRS} = \frac{\sum_{i=1}^{N} w_i s_i}{\sum_{i=1}^{N} w_i} \times 100\% \tag{1}$$

where:

- N is the total number of code quality checks performed by the automated code review tool.
- w_i is the weight assigned to the i-th code quality rule, reflecting its importance.
- s_i is the score for the i-th rule, defined as:

$$s_i = \frac{q_i}{Q_i} \tag{2}$$

where:

  - q_i is the raw score for the i-th rule, based on the specific quality measure (e.g., code readability, efficiency, etc.).
  - Q_i is the maximum possible score for the i-th rule, which ensures that s_i is normalized to the range $[0, 1]$.

A higher ACRS indicates better adherence to coding standards and best practices.

2. **Coverage (COV)**: This metric measures the proportion of actual sensitive code segments that our method successfully detects.
Let $S = \{s_1, s_2, \ldots, s_N\}$ be the set of actual sensitive code segments.
Let $D = \{d_1, d_2, \ldots, d_M\}$ be the set of detected sensitive code segments.
Define an indicator function:

$$I_{\text{detected}}(s_i) = \begin{cases} 1, & \text{if } \exists d_j \in D \text{ such that } d_j = s_i \\ 0, & \text{otherwise} \end{cases}$$

Then, the Coverage is defined as:

$$\text{COV} = \frac{\sum_{i=1}^{N} I_{\text{detected}}(s_i)}{N} \times 100\%$$

This metric reflects the percentage of actual sensitive code segments correctly detected by our method. Over-detection (detecting more code segments than actual sensitive code) is not penalized in this metric.

3. **Coverage Pass (COV-P)**: This metric assesses the accuracy of the try-blocks detected by the **Predator** agent compared to the actual code that requires try-catch blocks, penalizing over-detection. Let $T = \{t_1, t_2, \ldots, t_P\}$ be the set of actual code regions that should be enclosed in try-catch blocks (actual try-blocks).
Let $\hat{T} = \{\hat{t}_1, \hat{t}_2, \ldots, \hat{t}_Q\}$ be the set of code regions detected by the **Predator** agent as requiring try-catch blocks (detected try-blocks).
Define an indicator function:

$$I_{\text{correct}}(\hat{t}_j) = \begin{cases} 1, & \text{if } \hat{t}_j \in T \\ 0, & \text{otherwise} \end{cases}$$

The number of correctly detected try-blocks is:

$$\text{TP} = \sum_{j=1}^{Q} I_{\text{correct}}(\hat{t}_j)$$

The number of false positives (incorrectly detected try-blocks) is:

$$\text{FP} = Q - \text{TP}$$

The number of false negatives (actual try-blocks not detected) is:

$$\text{FN} = P - \text{TP}$$

We define the Coverage Pass (COV-P) as:

$$\text{COV-P} = \frac{\text{TP}}{P + \text{FP}} \times 100\%$$

This formulation penalizes over-detection by including the false positives in the denominator. A try-block is considered correct if it exactly matches the actual code lines; any over-marking or under-marking is counted as incorrect.

4. **Accuracy (ACC)**: This metric evaluates the correctness of the exception types identified by the **Predator** agent compared to the actual exception types.

   Let $E = \{e_1, e_2, \ldots, e_R\}$ be the set of actual exception types that should be handled.

   Let $\hat{E} = \{\hat{e}_1, \hat{e}_2, \ldots, \hat{e}_S\}$ be the set of exception types identified by the **Predator** agent.

   Define an indicator function:

$$I_{\text{correct}}(\hat{e}_j) = \begin{cases} 1, & \text{if } \hat{e}_j = e_i \\ 1, & \text{if } \hat{e}_j \text{ is a subclass of } e_i \\ 0, & \text{otherwise} \end{cases}$$

   Then, the Accuracy is defined as:

$$\text{ACC} = \frac{\sum_{j=1}^{S} I_{\text{correct}}(\hat{e}_j)}{S} \times 100\%$$

   This metric reflects the proportion of identified exception types that are correct, considering subclass relationships. Over-detection of incorrect exception types decreases the accuracy.

5. **Edit Similarity (ES)**: This metric computes the text similarity between the generated try-catch blocks and the actual try-catch blocks.

   Let G be the generated try-catch code, and A be the actual try-catch code.

   The Edit Similarity is defined as:

$$\text{ES} = 1 - \frac{\text{LevenshteinDistance}(G, A)}{\max(|G|, |A|)}$$

   where $\text{LevenshteinDistance}(G, A)$ is the minimum number of single-character edits (insertions, deletions, or substitutions) required to change G into A, and $|G|$, $|A|$ are the lengths of G and A, respectively.

   A higher ES indicates that the generated code closely matches the actual code.

6. **Code Review Score (CRS)**: This metric involves submitting the generated try-catch blocks to an LLM-based code reviewer (e.g., GPT-4o) for evaluation. The language model provides a binary assessment: *good* or *bad*.

   Let N_{good} be the number of generated try-catch blocks evaluated as *good*, and N_{total} be the total number of try-catch blocks evaluated.

   The Code Review Score is defined as:

$$\text{CRS} = \frac{N_{\text{good}}}{N_{\text{total}}} \times 100\%$$

   This metric reflects the proportion of generated exception handling implementations that are considered good according to engineering best practices.

Table 4: The Excerpt Data source

Repo	Commits	Stars	Forks	Issue Fix	Doc	Under Maintenance
Anki-Android	18410	8500	2200	262	Y	Y
AntennaPod	6197	6300	1400	295	Y	Y
connectbot	1845	2480	629	321	N/A	Y
FairEmail	30259	3073	640	N/A	Y	Y
FBReaderJ	7159	1832	802	248	Y	N/A
FP2-Launcher	1179	25	2	16	Y	N/A
NewsBlur	19603	6800	995	158	Y	Y
Launcher3	2932	91	642	2	N/A	Y
Lawnchair-V1	4400	93	43	394	Y	Y
MozStumbler	1727	619	212	203	Y	N/A

A.3.2 Datasets

To ensure the quality and representativeness of the dataset, we carefully selected projects on GitHub that are both active and large in scale. We applied stringent selection criteria, including the number of stars, forks, and exception handling repair suggestions in the project [30], to ensure that the dataset comprehensively covers the exception handling practices of modern open-source projects. By automating the collection of project metadata and commit history through the GitHub API, and manually filtering commit records related to exception handling, we have constructed a high-quality, representative dataset for exception handling that provides a solid foundation for evaluating Seeker.

We quantify the quality of datasets in the context of code generation and exception handling using multiple dimensions, encompassing project popularity, community engagement, codebase quality, security posture, documentation integrity and dynamic maintenance. To provide a holistic assessment, we propose a Composite Quality Metric (CQM) that aggregates these dimensions into a single quantitative indicator. Open source code repositories that perform well under this metric enter our semi-automated review process to screen high-quality exception handling blocks for few-shot, CEE building, or testing.

To avoid data leakage, we also performed a round of variations on the test set. Considering that our method does not directly rely on data but fully utilizes the LLM's ability to understand and reason about code, the evaluation results are consistent with our predictions, and the impact of data leakage on the credibility of our method is negligible.

A.3.3 Prompt and Document

CEE Prompt Template

genscenario = “““Below is a kind of exception in java. Please according to the sample discription of scenario of errortype, provide a scenario description of the exception in java just like the sample description.Please note that the granularity of the scenario descriptions you generate should be consistent with the examples.

[Sample Description]
{*sample_desc*}

[Exception]
{*ename*}

Note you should output in the json format like below, please note that the granularity of the scenario descriptions you generate should be consistent with the examples:
{{
    “scenario": ...
}}
"""
genproperty = “““Below is a kind of exception in java and its scenario description. Please according to the sample discription of scenario and property of errortype, provide a property description of the exception in java just like the sample description. You can alse adjust the given scenario description to make them consistent. Please note that the granularity of the property descriptions you generate should be consistent with the examples.

[Sample Description]
{*sample_desc*}

[Exception]
{*ename*}

[Scenario Description]
{*scenario*}

Note you should output in the json format like below, please note that the granularity of the property descriptions you generate should be consistent with the examples:
{{
    “scenario": ...;
    “property": ...
}}
"""

Scanner Prompt Template

scanner_prompt = “““You are a software engineer tasked with analyzing a codebase. Your task is to segment the given codebase into manageable units for further analysis. The criteria for segmentation are:
- Each unit should have a length within 200 lines.
- The function nesting level should be low.
- The logical flow should be clear and self-contained.
- The segment should be complete and readable.

Given the following codebase:

[Codebase]
{*codebase*}

Please segment the codebase into units and list them as:

Unit 1:[Code Segment]
{{unit1}}

Unit 2:[Code Segment]
{{unit2}}
...

Ensure that each unit complies with the criteria specified above.
"""

Detector Prompt Template

detector_senario_match = “““You are a java code auditor. You will be given a doc describe different exception scenarios and a java code snippet.

Your task is to label each line of the code snippet with the exception scenario that it belongs to. If a line does not belong to any scenario, label it with “None". If a line belongs to one of the given scenarios, label it with all the scenarios it belongs to.

[Scenario description]
{*scenario*}

[Java code]
{*code*}

Please output the labeling result in the json format like below:
{{
  “code_with_label": ...
}}
"""
detector_prop_match = “““You are a java code auditor. You will be given a doc describe different exception properties and a java code snippet.

Your task is to label each line of the code snippet with the exception property that it belongs to. If a line does not belong to any property, label it with “None". If a line belongs to one of the given properties, label it with all the properties it belongs to.

[property description]
{*property*}

[Java code]
{*code*}

Please output the labeling result in the json format like below:
{{
  “code_with_label": ...
}}
"""

Predator Prompt Template

predator_prompt = """"You are a code analysis assistant. Your task is to process the given code unit and identify specific exception types that may be thrown.

[Code Unit]
{*code_unit*}

[Code Summary]
{*code_summary*}

Based on the code summary and the potential exception branches provided, identify the specific exception nodes that may be thrown.

[Potential Exception Branches]
{*exception_branches*}

Please answer in the following JSON format:

```
{{
  "ExceptionNodes": [
    {{
      "ExceptionType": "ExceptionType1",
    }},
    {{
      "ExceptionType": "ExceptionType2",
    }},
    ...
  ]
}}
```

Ensure that your response strictly follows the specified format.
"""

Ranker Prompt Template

ranker_prompt = """"You are an exception ranking assistant. Your task is to assign grades to the identified exceptions based on their likelihood and the suitability of their handling strategies.

For each exception, please calculate:

- Exception Likelihood Score (from 0 to 1) based on its attributes and impact.
- Suitability Score (from 0 to 1) of the proposed handling strategy.

[Identified Exceptions and Handling Strategies]
{*exception_nodes*}

Provide your calculations and the final grades in the following JSON format:

```
{{
  "Exceptions": [
    {{
      "ExceptionType": "ExceptionType1",
      "LikelihoodScore": value,
      "SuitabilityScore": value,
    }},
    ...
  ]
}}
```

Please ensure your response adheres to the specified format.

"""

Handler Prompt Template

handler_prompt = """You are a software engineer specializing in exception handling. Your task is to optimize the given code unit by applying appropriate exception handling strategies.

[Code Unit]
{*code_unit*}

[Handling Strategy]
{*strategy1*}

Generate the optimized code with the applied exception handling strategies.

Please provide the optimized code in the following format:

[Optimized Code]
{{optimized_code}}

Ensure that the code is syntactically correct and adheres to best practices in exception handling.
"""

Sample CEE Node

{
“name": “IOException",
“children": [...],
“info": {
“definition": “IOException is a checked exception that is thrown when an input-output operation failed or interrupted. It’s a general class of exceptions produced by failed or interrupted I/O operations.",
“reasons": “There are several reasons that could cause an IOException to be thrown. These include: File not found error, when the file required for the operation does not exist; Accessing a locked file, which another thread or process is currently using; The file system is read only and write operation is performed; Network connection closed prematurely; Lack of access rights.",
“dangerous_operations": “Operations that could typically raise an IOException include: Reading from or writing to a file; Opening a non-existent file; Attempting to open a socket to a non-existent server; Trying to read from a connection after it’s been closed; Trying to change the position of a file pointer beyond the size of the file.",
“sample_code": “String fileName = ’nonexistentfile.txt’; \n FileReader fileReader = new FileReader(fileName);",
“handle_code": “String fileName = ’nonexistentfile.txt’; \n try { \n FileReader fileReader = new FileReader(fileName); \n } catch(IOException ex) { \n System.out.println(’An error occurred while processing the file ’ + fileName); \n ex.printStackTrace(); \n }",
“handle_logic": “Try the codes attempting to establish connection with a file/stream/network, catch corresponding IOException and report it, output openpath is suggested."
},
“scenario": “attempt to read from or write to a file/stream/network connection",
“property": “There might be an unexpected issue with accessing the file/stream/network due to reasons like the file not being found, the stream being closed, or the network connection

```
being interrupted"
}
```

A.3.4 Computation Cost Analysis

Integrating a comprehensive exception handling mechanism like **Seeker** introduces potential challenges in computational overhead, especially when dealing with a large number of exception types and complex inheritance relationships. To address this, we designed a high-concurrency interface that keeps the additional computing time overhead constant, regardless of the code volume level. This ensures scalability and controllable complexity when processing any size of codebase.

To evaluate the efficiency of our high-concurrency interface, we conducted experiments on 100 Java code files both before and after implementing parallel processing. For each code file, we executed the exception handling process and recorded the time taken. In the parallelized version, while the processing between different code files remained sequential, the processing within each code file—specifically, the CEE retrieval involving branch and layered processing—was parallelized.

The results are summarized in Table 5. After applying parallel processing, the average time per code file was reduced to approximately 19.4 seconds, which is about $\frac{1}{15}$ of the time taken with sequential processing. This significant reduction demonstrates the effectiveness of our parallelization strategy.

Table 5: Computation Time Before and After Parallelization

Processing Method	**Average Time per Code File (s)**	**Speedup Factor**
Sequential Processing	291.0	1x
Parallel Processing (Seeker)	19.4	15x

Notably, the size of the code files did not affect the processing time, indicating that our method efficiently handles codebases of varying sizes without compromising on speed. This stability ensures that **Seeker** can perform consistent and efficient exception handling across any code, making it highly suitable for practical applications.

A.3.5 Further Results on different LLMs

We use different open-source (e.g. Code Llama-34B [34], WizardCoder-34B [26], Vicuna-13B [49], Qwen3-32B) and closed-source (e.g. Claude-2 [3], GPT-3-davinci [10], GPT-3.5-turbo [11], GPT-4-turbo [12], GPT-4o [13], Claude 4.5 Sonnet, GPT-5, DeepSeek R1) LLMs as the agent's internal model to further analyze models' ability for exception handling. The results are summarized in Table 6.

The performance variations among different models can be explained by:

- **Pre-training Data**: Models pre-trained on larger and more diverse code datasets (e.g., GPT-4o) have a better understanding of programming constructs and exception handling patterns.

- **Model Architecture**: Advanced architectures with higher capacities and more layers (e.g., GPT-4o) capture complex patterns more effectively.

- **RAG Performance**: Models that efficiently integrate retrieval-augmented generation, effectively utilizing external knowledge (as in our method), perform better.

- **Understanding Capability**: Models with superior comprehension abilities can accurately detect sensitive code regions and predict appropriate exception handling strategies.

Open-source models, while valuable, may lack the extensive training data and architectural sophistication of closed-source models, leading to lower performance. Closed-source models like GPT-4o and GPT-4 benefit from advanced training techniques and larger datasets, enabling them to excel in tasks requiring nuanced understanding and generation of code, such as exception handling.

Table 6: Performance of Different Models on Exception Handling Code Generation

Model	**ACRS**	**COV (%)**	**COV-P (%)**	**ACC (%)**	**ES**	**CRS (%)**
			Open-Source Models			
Qwen3-32B	0.58	75	68	62	0.50	68
Code Llama-34B	0.31	37	35	32	0.25	34
WizardCoder-34B	0.37	35	31	29	0.28	35
Vicuna-13B	0.23	15	9	11	0.19	26
			Closed-Source Models			
GPT-5	**0.94**	**96**	**90**	**88**	**0.78**	**97**
Claude 4.5 Sonnet	0.92	95	86	85	0.75	95
DeepSeek R1	0.87	93	85	82	0.73	93
Claude-2	0.42	64	59	54	0.40	54
GPT-3-davinci	0.56	78	68	60	0.48	58
GPT-3.5-turbo	0.63	79	72	66	0.52	71
GPT-4-turbo	0.84	91	83	77	0.63	89
GPT-4o	0.85	91	81	79	0.64	92

Table 7: Performance on SWE-bench Lite Exception Handling Issues

Method	**Resolve Rate (%)**	**Apply Rate (%)**
SweAgent + GPT-4o	19	43
Seeker + GPT-4o	**26**	**61**

A.4 Other Applicable Scenarios Analysis

Figure 7 shows the migration application of Seeker multi-agent framework in APP requirement engineering that also includes parent-child inheritance relationship. We have reason to believe that Seeker framework can try to be compatible with more complex inheritance relationship, being responsible for reasoning representation, while having high performance and interpretability. The above achievements are not easy to accomplish based on graphs or traditional algorithms.

To validate the general applicability of our system in diverse scenarios, we evaluated **Seeker** on standard code generation benchmarks, including **SWE-bench** and **CoderEval**. We present comparative results demonstrating the incremental improvements achieved by our method.

SWE-bench is an evaluation framework comprising 2,294 software engineering problems derived from real GitHub issues and corresponding pull requests across 12 popular Python repositories[18]. It challenges language models to edit a given codebase to resolve specified issues, often requiring understanding and coordinating changes across multiple functions, classes, and files simultaneously. This goes beyond traditional code generation tasks, demanding interaction with execution environments, handling extremely long contexts, and performing complex reasoning.

For our experiments, we sampled half issues from the SWE-bench dataset. Using **GPT-4o** as the internal model, the **SweAgent**[43] coupled with GPT-4o achieved a **19.10%** *resolve rate* and a **43.56%** *apply rate*. In contrast, our **Seeker** framework attained a **27.98%** resolve rate and a **62.11%** apply rate, indicating a significant improvement.

CoderEval is a benchmark designed to assess the performance of models on pragmatic code generation tasks, moving beyond generating standalone functions to handling code that invokes or accesses custom functions and libraries[44]. It evaluates a model's ability to generate functional code in real-world settings, similar to open-source or proprietary projects.

In the Java code generation tasks on CoderEval, using **Codex**[5] directly yielded a **Pass@1** score of **27.83%**. When integrating our **Seeker** framework with Codex, the Pass@1 score increased to **38.16%**, demonstrating a substantial enhancement in code generation performance.

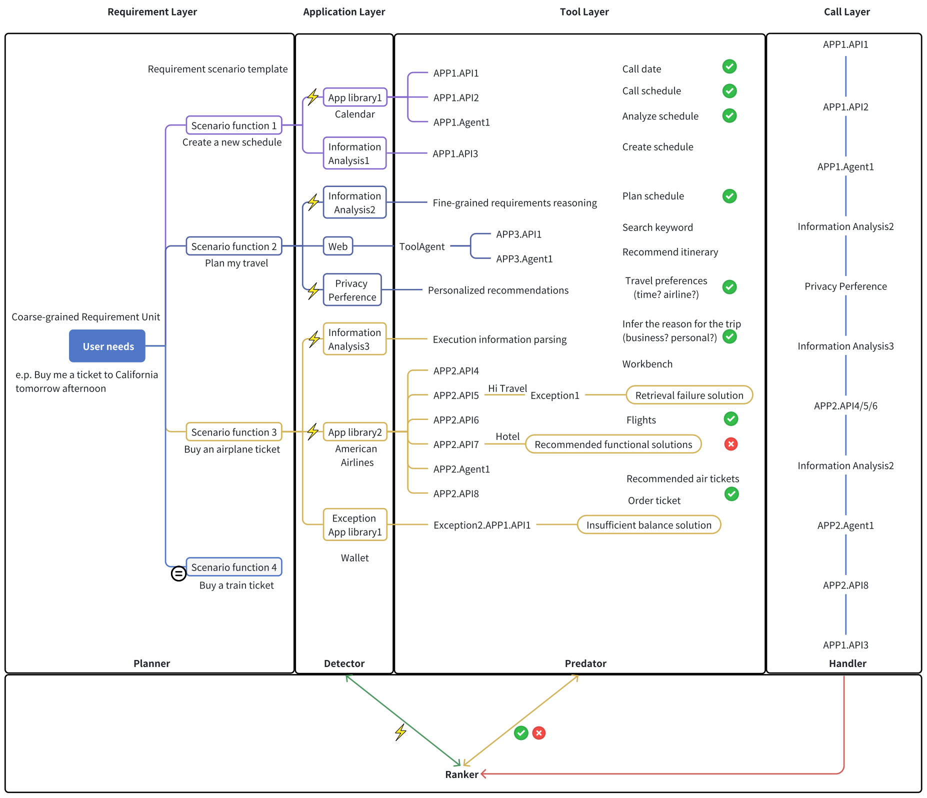


Figure 7: A schematic depiction of integrating the Seeker multi-agent framework into APP requirement engineering workflows. By bridging layered requirements, application functionalities, tool integrations, and call-level operations, Seeker generalizes beyond isolated exception handling to more complex inheritance relationships. This approach improves interpretability, scalability, and reasoning capabilities, demonstrating the framework's adaptability and high performance across diverse, real-world engineering scenarios.

Table 8: Performance on CoderEval Java Code Generation Tasks

Method	**Pass@1 (%)**
Codex	27.83
Seeker + Codex	**38.16**

These experiments conclusively demonstrate that our **Seeker** framework can achieve significant incremental improvements across different scenarios and benchmarks. By effectively handling exception-related tasks and enhancing code robustness, **Seeker** proves to be a valuable addition to existing code generation models, improving their practical applicability in real-world software engineering problems.

Inspired by OpenAI o1 [31] and DoT [48], we found that Seeker framework has more space for development in LLM reasoning. Through pre-deduction in tree inference, LLM is expected to enter the problem-solving ideas more efficiently and optimize its reasoning actions through interaction with the external environment. In the future, we will continue to explore research in this direction.

B Related Work

At present, machine learning has been widely integrated in the field of software engineering, especially in code generation tasks. In this section, we will discuss the progress of Seeker-related work from the latest progress of automatic exception handling tools. These methods have contributed to the robustness or productivity of software engineering, but they also have limitations, which is also the focus of Seeker.

B.1 Automatic Exception Handling Tools

Recent work[46] introduced a neural network approach for automated exception handling in Java, which predicts try block locations and generates complete catch blocks in relatively high accuracy. However, the approach is limited to Java and not generalize well without retraining. Additionally, the reliance on GitHub data could introduce biases based on the types of projects and code quality present in the dataset.

SCG[21] conducted an exploratory study on fine-tuning LLM for secure code generation. Their results showed that after fine-tuning issue fixing commits, the secure code generation rate was slightly improved. The best performance was achieved by fine-tuning using function-level and block-level datasets. However, the limitation of this study is still generalization, not directly applicable to other languages. In addition, it limits the amount and the domain of code that can be effectively processed. Little much code beyond training data scale will affect the processing effect. Besides, in terms of automatic vulnerability detection, the use of traditional fine-tuning methods may not fully utilize the domain knowledge in the pre-trained language model, and may overfit to a specific dataset, resulting in misclassification, excessive false positives and false negatives[25]. Its performance is not as good as emerging methods such as prompt-based learning.

Knowledge-driven Prompt Chaining (KPC)[33], an approach to improve code generation by chaining fine-grained knowledge-driven prompts. Their evaluation with 3,079 code generation tasks from Java API documentation showed improvements in exception handling. However, the approach's efficiency relies heavily on the inquiry about built-in exceptions for each built-in JDK, and its practical application is limited if the codebase is complex.

FuzzyCatch[29], a tool for recommending exception handling code for Android Studio based on fuzzy logic. However, the performance of FuzzyCatch depends on the quality and relevance of the training data. In addition, the tool does not perform well for less common exceptions or domains that are not well represented in the training data.

Neurex[1], a learning-based exception handling recommender that leverages the CodeBERT model to suggest appropriate try-catch blocks, the statements to include within try blocks, and the exception types to catch. However, Neurex still has several limitations. It cannot generate new exception types that were not in the training corpus with low cost. It does not support the generation of exception handling code inside the catch body. Each project might have a different way to handle exception types in the catch body. And Neurex also needs training data, thus, does not work for a new library without any API usage yet. Most importantly, we compared the experimental results and found that even in the experimental granularity of their method, they perform averagely and are primarily good at finding existing exception handling bugs, which is not our focus. Above all, we have had similar method for baseline so we did not compare with them in the formal experimental part.

A common limitation of these studies is that the training data they rely on may not fully represent all possible coding scenarios. This may result in a model that is effective in specific situations, but may not generalize well to other situations. In addition, the complexity of exception handling in real-world applications may exceed the capabilities of models trained on more common or simpler cases, so it is crucial to call on the understanding and reasoning capabilities of the model itself. The interpretability of exception handling also provides a guarantee for the improvement of developers' programming literacy. The comparison between the above methods and Seeker is shown in figure 5.

B.2 Multi-agent Collaberation

Multi-agent collaboration refers to the coordination and collaboration between multiple artificial intelligence (AI) systems, or the symbiotic collaboration between AI and humans, working together to achieve a common goal [37]. This direction has been explored for quite some time [4] [27]. Recent developments show that multi-agent collaboration techniques are being used to go beyond the limitations of LLM, which is a promising trajectory. There are many ways for multi-agents to collaborate with LLM.

VisualGPT [42] and HuggingGPT [35] explored the collaboration between LLM and other models. Specifically, LLM was used as a decision center to control and call other models to handle more domains, such as vision, speech, and signals. CAMEL [19] explored the possibility of interaction between two LLMs. These studies mainly use case studies in the experimental stage to demonstrate their effectiveness and provide specific hints for each case.

For multi-agent collaborative software engineering, which is most relevant to Seeker, [8] introduces quantitative analysis to evaluate agent collaborative code generation. It introduces the waterfall model in software development methods into the collaboration between LLMs. However, there is still a gap between the evaluation benchmarks used and the actual software development scenarios. In addition, although this work builds a fully autonomous system, adding a small amount of guidance from human experts to supervise the operation of the virtual team will help improve the practicality of the method in actual application scenarios. These problems are exactly what we have improved on Seeker.

CODEAGENT[47] formalized the repo-level code generation task and proposed a new agent framework based on LLM. CODEAGENT developed five programming tools to enable LLM to interact with software artifacts and designed four agent strategies to optimize the use of tools. The experiment achieved improvements on various programming tasks. However, it only integrated simple tools into CODEAGENT. Some advanced programming tools were not explored. This limitation limits the ability of the agent in some challenging scenarios, such as exception handling tasks.

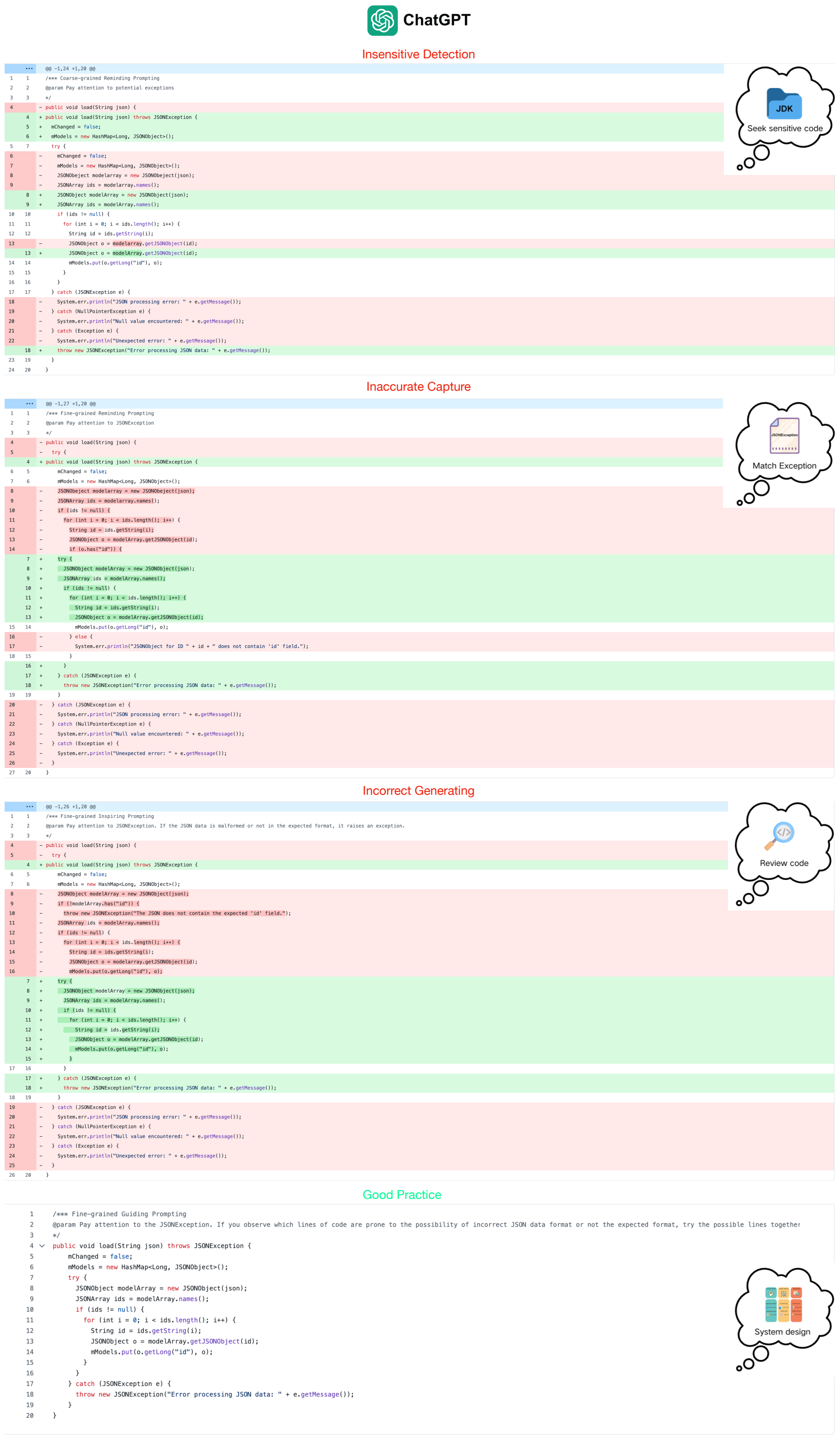
Above all, nowadays, most code-agent works focus on the transformation from the requirements to code and overlook the code robustness during software evolution, which requires not only understanding the requirement but also dealing with potential exceptions.

B.3 Robust Software Development Mechanism

Code robustness refers to the practices and mechanisms that ensure software to run as expected without causing unexpected side effects, security vulnerabilities, or errors. It involves techniques such as type safety, memory safety, and ensuring that all code paths are well-defined, including when exceptions exist. Exception handling is a necessary programming mechanism to maintain code robustness, allowing programs to manage and respond to runtime errors or other abnormal events. It helps maintain the normal flow of execution and ensures that resources are properly released even when errors occur. Exception handling is critical to code robustness because it ensures that unexpected errors do not compromise the stability or security of the system, prevents resource leaks, ensures data integrity, and keeps the program running correctly even when unforeseen errors occur[40].

From the perspective of code robustness, the defect repair work in the field of software engineering is closely related to exception handling mechanisms, because exception handling involves solving potential errors in the program flow, and developers can mitigate or eliminate defects that may cause program failures or unpredictable behavior[17]. Currently, since each defect represents a potential vulnerability or instability in the software and is directly related to the functional correctness of the program, research focuses more on defect repair[41], Devign [41], VulAdisor [41], while the program's exception safety and exception handling, the powerful program defense mechanisms are not considered.

When a program lacks good exception handling, errors may propagate uncontrollably, leading to resource leakage, data corruption, and potential security vulnerabilities. This situation is called fragile code. After the error occurs, Automatic Program Repair related work performs post-processing to fix the code bug[50]. Representative works include Magis [38], PatchFinder [23]. However, they lack the ability to perceive and repair program risks in advance, and there is a risk of accidentally changing the original function of the code[16].



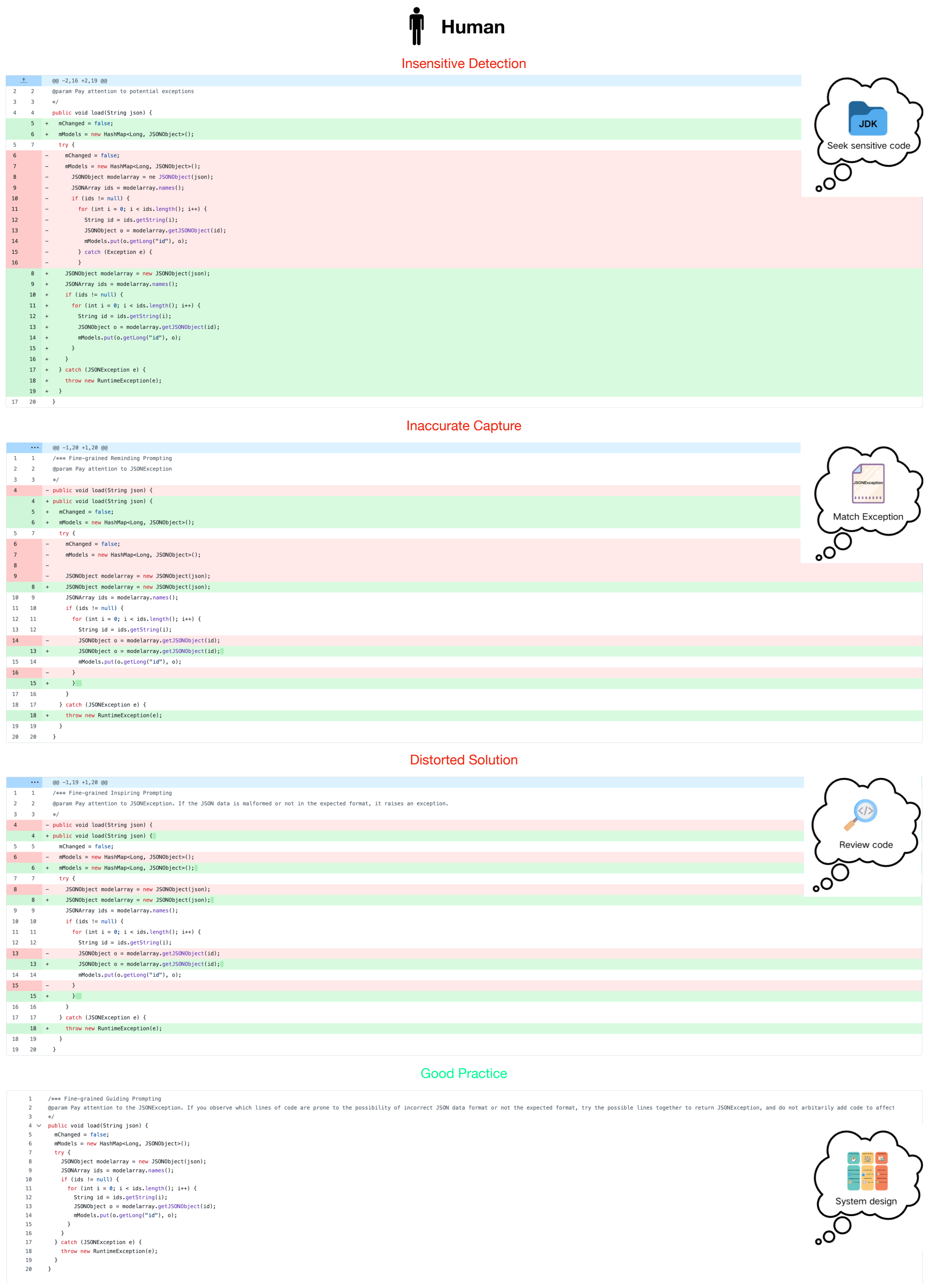
```
@@ -1,24 +1,20 @@
  /*** Coarse-grained Reminding Prompting
  @param Pay attention to potential exceptions
  */
- public void load(String json) {
+ public void load(String json) throws JSONException {
+   mChanged = false;
+   mModels = new HashMap<Long, JSONObject>();
    try {
-     mChanged = false;
-     mModels = new HashMap<Long, JSONObject>();
-     JSONObeject modelarray = new JSONObeject(json);
-     JSONArray ids = modelarray.names();
+     JSONObject modelArray = new JSONObject(json);
+     JSONArray ids = modelArray.names();
      if (ids != null) {
        for (int i = 0; i < ids.length(); i++) {
          String id = ids.getString(i);
-         JSONObject o = modelarray.getJSONObject(id);
+         JSONObject o = modelArray.getJSONObject(id);
          mModels.put(o.getLong("id"), o);
        }
      }
    } catch (JSONException e) {
-     System.err.println("JSON processing error: " + e.getMessage());
-   } catch (NullPointerException e) {
-     System.err.println("Null value encountered: " + e.getMessage());
-   } catch (Exception e) {
-     System.err.println("Unexpected error: " + e.getMessage());
+     throw new JSONException("Error processing JSON data: " + e.getMessage());
    }
  }
```

```
@@ -1,27 +1,20 @@
  /*** Fine-grained Reminding Prompting
  @param Pay attention to JSONException
  */
- public void load(String json) {
-   try {
+ public void load(String json) throws JSONException {
      mChanged = false;
      mModels = new HashMap<Long, JSONObject>();
-     JSONObeject modelarray = new JSONObeject(json);
-     JSONArray ids = modelarray.names();
-     if (ids != null) {
-       for (int i = 0; i < ids.length(); i++) {
-         String id = ids.getString(i);
-         JSONObject o = modelArray.getJSONObject(id);
-         if (o.has("id")) {
+     try {
+       JSONObject modelArray = new JSONObject(json);
+       JSONArray ids = modelArray.names();
+       if (ids != null) {
+         for (int i = 0; i < ids.length(); i++) {
+           String id = ids.getString(i);
+           JSONObject o = modelArray.getJSONObject(id);
            mModels.put(o.getLong("id"), o);
-         } else {
-           System.err.println("JSONObject for ID " + id + " does not contain 'id' field.");
          }
+       }
+     } catch (JSONException e) {
+       throw new JSONException("Error processing JSON data: " + e.getMessage());
      }
-   } catch (JSONException e) {
-     System.err.println("JSON processing error: " + e.getMessage());
-   } catch (NullPointerException e) {
-     System.err.println("Null value encountered: " + e.getMessage());
-   } catch (Exception e) {
-     System.err.println("Unexpected error: " + e.getMessage());
-   }
  }
```

```
@@ -1,26 +1,20 @@
  /*** Fine-grained Inspiring Prompting
  @param Pay attention to JSONException. If the JSON data is malformed or not in the expected format, it raises an exception.
  */
- public void load(String json) {
-   try {
+ public void load(String json) throws JSONException {
      mChanged = false;
      mModels = new HashMap<Long, JSONObject>();
-     JSONObject modelArray = new JSONObject(json);
-     if (!modelArray.has("id")) {
-       throw new JSONException("The JSON does not contain the expected 'id' field.");
-     JSONArray ids = modelArray.names();
-     if (ids != null) {
-       for (int i = 0; i < ids.length(); i++) {
-         String id = ids.getString(i);
-         JSONObject o = modelarray.getJSONObject(id);
-         mModels.put(o.getLong("id"), o);
+     try {
+       JSONObject modelArray = new JSONObject(json);
+       JSONArray ids = modelArray.names();
+       if (ids != null) {
+         for (int i = 0; i < ids.length(); i++) {
+           String id = ids.getString(i);
+           JSONObject o = modelArray.getJSONObject(id);
+           mModels.put(o.getLong("id"), o);
+         }
        }
+     } catch (JSONException e) {
+       throw new JSONException("Error processing JSON data: " + e.getMessage());
      }
-   } catch (JSONException e) {
-     System.err.println("JSON processing error: " + e.getMessage());
-   } catch (NullPointerException e) {
-     System.err.println("Null value encountered: " + e.getMessage());
-   } catch (Exception e) {
-     System.err.println("Unexpected error: " + e.getMessage());
-   }
  }
```

```
/*** Fine-grained Guiding Prompting
@param Pay attention to the JSONException. If you observe which lines of code are prone to the possibility of incorrect JSON data format or not the expected format, try the possible lines together
*/
public void load(String json) throws JSONException {
    mChanged = false;
    mModels = new HashMap<Long, JSONObject>();
    try {
      JSONObject modelArray = new JSONObject(json);
      JSONArray ids = modelArray.names();
      if (ids != null) {
        for (int i = 0; i < ids.length(); i++) {
          String id = ids.getString(i);
          JSONObject o = modelArray.getJSONObject(id);
          mModels.put(o.getLong("id"), o);
        }
      }
    } catch (JSONException e) {
      throw new JSONException("Error processing JSON data: " + e.getMessage());
    }
}
```

Figure 8: A schematic illustration of the preliminary phenomenon, showing how incremental, targeted guidance enhances LLM-based exception handling. The depicted code segments and annotations highlight which specific information supports more accurate detection and handling of fragile code scenarios.



```
@@ -2,16 +2,19 @@
  @param Pay attention to potential exceptions
  */
  public void load(String json) {
+   mChanged = false;
+   mModels = new HashMap<Long, JSONObject>();
    try {
-     mChanged = false;
-     mModels = new HashMap<Long, JSONObject>();
-       JSONObject modelarray = ne JSONObject(json);
-       JSONArray ids = modelarray.names();
-       if (ids != null) {
-         for (int i = 0; i < ids.length(); i++) {
-           String id = ids.getString(i);
-           JSONObject o = modelarray.getJSONObject(id);
-           mModels.put(o.getLong("id"), o);
-         } catch (Exception e) {
-         }
+     JSONObject modelarray = new JSONObject(json);
+     JSONArray ids = modelarray.names();
+     if (ids != null) {
+       for (int i = 0; i < ids.length(); i++) {
+         String id = ids.getString(i);
+         JSONObject o = modelarray.getJSONObject(id);
+         mModels.put(o.getLong("id"), o);
+       }
+     }
+   } catch (JSONException e) {
+     throw new RuntimeException(e);
+   }
  }
```

```
@@ -1,20 +1,20 @@
  /*** Fine-grained Reminding Prompting
  @param Pay attention to JSONException
  */
- public void load(String json) {
+ public void load(String json) {
+   mChanged = false;
+   mModels = new HashMap<Long, JSONObject>();
    try {
-     mChanged = false;
-     mModels = new HashMap<Long, JSONObject>();
-
-     JSONObject modelarray = new JSONObject(json);
+     JSONObject modelarray = new JSONObject(json);
      JSONArray ids = modelarray.names();
      if (ids != null) {
        for (int i = 0; i < ids.length(); i++) {
          String id = ids.getString(i);
-         JSONObject o = modelarray.getJSONObject(id);
+         JSONObject o = modelarray.getJSONObject(id);
          mModels.put(o.getLong("id"), o);
-       }
+       }
      }
    } catch (JSONException e) {
+     throw new RuntimeException(e);
    }
  }
```

```
@@ -1,19 +1,20 @@
  /*** Fine-grained Inspiring Prompting
  @param Pay attention to JSONException. If the JSON data is malformed or not in the expected format, it raises an exception.
  */
- public void load(String json) {
+ public void load(String json) {
    mChanged = false;
-   mModels = new HashMap<Long, JSONObject>();
+   mModels = new HashMap<Long, JSONObject>();
    try {
-     JSONObject modelarray = new JSONObject(json);
+     JSONObject modelarray = new JSONObject(json);
      JSONArray ids = modelarray.names();
      if (ids != null) {
        for (int i = 0; i < ids.length(); i++) {
          String id = ids.getString(i);
-         JSONObject o = modelarray.getJSONObject(id);
+         JSONObject o = modelarray.getJSONObject(id);
          mModels.put(o.getLong("id"), o);
-       }
+       }
      }
    } catch (JSONException e) {
+     throw new RuntimeException(e);
    }
  }
```

```
/*** Fine-grained Guiding Prompting
@param Pay attention to the JSONException. If you observe which lines of code are prone to the possibility of incorrect JSON data format or not the expected format, try the possible lines together to return JSONException, and do not arbitarily add code to affect
*/
public void load(String json) {
  mChanged = false;
  mModels = new HashMap<Long, JSONObject>();
  try {
    JSONObject modelarray = new JSONObject(json);
    JSONArray ids = modelarray.names();
    if (ids != null) {
      for (int i = 0; i < ids.length(); i++) {
        String id = ids.getString(i);
        JSONObject o = modelarray.getJSONObject(id);
        mModels.put(o.getLong("id"), o);
      }
    }
  } catch (JSONException e) {
    throw new RuntimeException(e);
  }
}
```

Figure 9: A schematic illustration of the preliminary phenomenon, demonstrating that incremental, targeted guidance similarly benefits both LLMs and human developers in exception handling. The highlighted case study underscores which information elements help bridge the gap between current human practice and reliable, automated handling strategies.