An Empirical Evaluation of Using Large Language Models to Recommend Exception Handling Tasks

Jairo Souza
Centro de Informática
Universidade Federal de Pernambuco
Recife, PE, Brasil
jrmcs@cin.ufpe.br

Tales Alves
Centro de Informática
Universidade Federal de Pernambuco
Recife, PE, Brasil
tvac@cin.ufpe.br

Rodrigo Lima
Centro de Informática
Universidade Federal de Pernambuco
Recife, PE, Brasil
rsl@cin.ufpe.br

Ericlecio Araújo
Centro de Informática
Universidade Federal de Pernambuco
Recife, PE, Brasil
etma@cin.ufpe.br

Leopoldo Teixeira
Centro de Informática
Universidade Federal de Pernambuco
Recife, PE, Brasil
lmt@cin.ufpe.br

Baldoino Fonseca
Instituto de Computação
Universidade Federal de Alagoas
Maceió, AL, Brasil
baldoino@ic.ufal.br

Abstract

Large Language Models (LLMs) are increasingly used to support software engineering tasks such as code generation, bug fixing, and program comprehension. However, their ability to support exception handling tasks remains largely unexplored. Prior work on automated exception handling has mostly relied on task-specific models, fine-tuning, or specialized pipelines, which may limit reuse across projects and as APIs evolve. In this paper, we empirically evaluate whether open-source and proprietary LLMs can support Python exception handling tasks through prompting alone, without fine-tuning. We study four tasks: (1) detecting whether exception handling is necessary, (2) localizing statements that should be enclosed in a `try` block, (3) recommending exception types for `except` clauses, and (4) generating exception handlers. We further complement the generation task with a quality assessment of the generated exception handlers. We evaluated several LLMs, including CodeLlama, Phi-4, GPT-4 Mini, Claude Haiku 4.5, and Gemini 3 Flash, across four prompting strategies using a benchmark from 21 open-source Python projects, collecting 479,345 functions and curating 379 code snippets. Results showed stronger effectiveness for exception necessity detection and try-block localization, with best F-scores of 0.752 and 0.733, respectively, while exception-type recommendation and handler generation remained substantially more challenging. Overall, our findings indicate that prompt-based LLMs are more suitable as early-stage exception handling tasks for detecting need contexts and localizing risky statements. Also, generated exception handlers still require validation, as they often diverge semantically from developer-written implementations and introduce more exception-related quality violations.

Keywords

Empirical Study, Large Language Models, Exception Handling, Antipatterns

1 Introduction

Large Language Models (LLMs) have transformed how developers interact with source code. Recent models (e.g., Claude, Gemini, ChatGPT) have been successfully applied to code generation, debugging, refactoring, testing, and documentation [14, 19, 21, 27, 29, 30, 34, 35]. These advances highlight the potential of LLMs to support a wide range of tasks through natural-language instructions.

One such task is exception handling, a fundamental aspect of robust software design that enables systems to gracefully manage unexpected runtime conditions [5]. However, developers often struggle with key decisions when implementing exception handling mechanisms, including whether a `try-except` block is necessary, which statements should be protected, which exception types to handle, and how to implement appropriate recovery actions. Prior studies have shown that inadequate exception handling can introduce bugs, obscure failures, and reduce maintainability, often through the presence of exception handling antipatterns [3, 4, 8, 16, 28, 31, 32]. Existing automated approaches, such as mining-based techniques, fuzzy-logic recommenders, neural models, and fine-tuned transformers, typically require task-specific training and are often evaluated in Java [1, 7, 13, 24, 36, 37].

In contrast, this work investigates whether general-purpose LLMs can support multiple Python exception handling tasks through prompting alone, without fine-tuning. Compared with Neurex [7], which fine-tunes CodeBERT on Java for three of these tasks, and L2HE [36], which trains a neural model from scratch on Java, we (i) target Python, whose dynamic typing and lack of checked exceptions make API-level exception inference different; (ii) rely on prompting alone, eliminating the need for task-specific training data and enabling reuse as models evolve; and (iii) go beyond previous works by also examining exception-handler generation through automatic metrics, human assessment, and static analysis.

To this end, we design an empirical study that decomposes exception handling into four complementary tasks reflecting developer reasoning: (1) determining whether exception handling is necessary, (2) localizing statements that should be enclosed in a `try` block, (3) recommending appropriate exception types, and (4) generating exception handlers. We evaluate four prompting strategies (default, one-shot, few-shot, and chain-of-thought) across open-source and proprietary LLMs, namely CodeLlama, Phi-4, GPT-4 Mini, Claude Haiku 4.5, and Gemini 3 Flash. We also complement exception handler generation with a static-analysis-based quality assessment of the generated handlers.

Our results show that LLMs are more effective for early-stage reasoning tasks than for generation. GPT-4 Mini achieved the highest observed values in necessity detection (F-score = 0.752), while Gemini 3 Flash achieved the highest observed values in try-block localization (F-score = 0.733). In contrast, exception-type recommendations remain challenging, with low F1 scores across models, and exception handler generation achieved limited semantic alignment with reference implementations despite moderate lexical similarity (best CrystalBLEU = 0.104). Human evaluation indicated that 48%-67% of generated handlers were perceived as weakly aligned with developer-written code, and static analysis showed that generated handlers introduced 104.2% more exception-related quality violations on average, particularly due to missing exception chaining and overly broad exception catching.

This study makes four main contributions. **First**, we provide empirical evidence on prompt-based LLMs for Python exception handling, complementing the Java-and-fine-tuning line of work represented by Neurex [7] and L2HE [36]. We find that LLMs reach reasonable scores on necessity detection and try-block localization but remain weak on type recommendation and handler generation. **Second**, we compare five open-source and proprietary LLMs across four prompting strategies, characterizing where prompt complexity helps and where it does not. **Third**, we triangulate evaluation across automatic similarity metrics, human assessment, and static-analysis-based quality measurement, showing that lexical similarity to reference handlers is a poor proxy for semantic correctness or anti-pattern avoidance. **Finally**, we release a curated benchmark of Python code snippets, prompt templates, and evaluation scripts to support reproducibility and future work.

1.1 Motivating Example

To illustrate the motivation and scope of our study, Figure 1 presents a real function extracted from the *Django* framework [1].

The function `get_language_from_path` extracts and validates a language code from a path string. In Figure 1a, the function calls `get_supported_language_variant()` without exception handling, assuming that the language code is always valid. However, this call may raise a `LookupError` when the language variant is invalid or unsupported. Figure 1b shows the corrected version, where the risky call is enclosed in a `try-except` block and the exception type `LookupError` is handled without interrupting program execution.

This example illustrates the reasoning steps developers perform when adding exception handling mechanisms. In this paper, we decompose this process into four exception handling tasks:

(1) **#1. Is a `try-except` Block Necessary?** This task determines whether exception handling is needed. In the example, a `try-except` block is necessary because `get_supported_language_variant()` may fail when the extracted language code is invalid or unsupported.
(2) **#2. Try Block Localization.** This task identifies which statements should be enclosed within the `try` block. In the example, the statement `return get_supported_language_variant(lang_code, strict=strict)` is the risky operation. The previous statements, such as the regular expression match and variable assignment, remain outside the `try` block to avoid unnecessarily broad exception handling.
(3) **#3. Determining the Required Exception Types.** This task identifies which exception type should be handled in the `except` clause. In the example, the appropriate type is `LookupError`, which represents invalid or unsupported language variants returned by the validation function.
(4) **#4. Except Handler Generation.** This task synthesizes the complete `except` block. In the example, the expected handler catches `LookupError` and returns `None`, providing a safe recovery behavior instead of propagating the failure.

[1] https://github.com/django/django

```python
def get_language_from_path(path, strict=False):
    """
    Return the language code if there's a valid
        language code found in `path`.
    """
    regex_match = language_code_prefix_re.match(
        path)
    if not regex_match:
        return None
    lang_code = regex_match[1]
    return get_supported_language_variant(
        lang_code, strict=strict)
```

1: (a) Code snippet without exception handling.

```python
def get_language_from_path(path, strict=False):
    """
    Return the language code if there's a valid
        language code found in `path`.
    """
    regex_match = language_code_prefix_re.match(
        path)
    if not regex_match:
        return None
    lang_code = regex_match[1]
    try:
        return get_supported_language_variant(
            lang_code, strict=strict)
    except LookupError:
        return None
```

2: (b) Code snippet after adding exception handling.

Figure 1: Code snippets illustrating exception handling.

By applying these steps, the unhandled version in Figure 1a can be transformed into the handled version in Figure 1b. This example shows how exception handling can be decomposed into interpretable tasks for systematic evaluation. We use this decomposition to assess how well LLMs perform distinct Python exception handling tasks on real-world open-source code.

2 Study Design

This study investigates how effectively Large Language Models (LLMs) can support exception handling tasks through prompt-based strategies. We evaluate four tasks that capture distinct reasoning aspects of exception handling, ranging from identifying whether a

`try-except` block is needed to generating complete `except` handlers. In addition, we conduct a complementary quality assessment of the generated handlers to determine whether LLM outputs adhere to exception handling quality guidelines. Therefore, we define the following research questions:

RQ_1: How effectively can LLMs determine whether exception handling is necessary? This research question evaluates the model's ability to identify code snippets that require exception handling. The task involves binary classification, predicting whether a code snippet should contain a `try-except` block. This task indicates how well LLMs perform risk assessment and context-aware reasoning for exception handling mechanisms.

RQ_2: How effectively can LLMs identify which statements should be enclosed in a `try` block? This research question examines whether LLMs can identify which statements should be enclosed within a `try` block. The task requires distinguishing operations that may raise exceptions from statements that can safely remain outside the protected region. This task reflect how well a model captures statement-level execution flow, data and control dependencies, and the propagation of failures.

RQ_3: How effectively can LLMs recommend the exception types that should be handled in the `except` clauses? This research question examines whether a model can infer which exception type is appropriate for a given code context. Because each instance requires selecting one label from multiple possible exception types, we formulate the task as a multi-class recommendation problem. This task indicates how well the model maps code operations and control-flow context to language-specific exception hierarchies, such as distinguishing between input-conversion errors, missing resources, indexing failures, and invalid state conditions.

RQ_4: How effectively can LLMs generate complete exception handlers? This research question evaluates LLMs' ability to synthesize coherent, semantically meaningful exception handling code from the provided context. Unlike previous prediction tasks, this task requires producing syntactically valid `except` blocks, selecting appropriate exception types, and generating useful recovery actions or diagnostic messages.

RQ_5: What exception-related quality violations are introduced by LLM-generated exception handlers? This research question complements Task 4 by assessing the quality of the generated exception handlers. Even when generated handlers are syntactically valid or lexically similar to reference implementations, they may still contain exception handling anti-patterns such as overly broad exception catching as overly broad catches, or empty handlers. We therefore use static analysis to determine whether generated handlers introduce exception-related quality violations commonly associated with poor practices.

2.1 Dataset Collection and Filtering

In the first step, we collected functions from 21 open-source Python projects to construct the datasets to answer the research questions. We selected repositories using the *Software Engineering Research Tool* (SEART),[2] focusing on active and widely used Python projects. We required each repository to have at least 5,000 stars, 5,000 commits, 100 pull requests, open issues, and recent development activity. Selected projects include *Django*, *FastAPI*, *LangChain*, *pytest*, *pip*, *spaCy*, and *Azure SDK for Python*. From these repositories, we extracted functions containing at least one `try-except` block using a custom parser built on top of Python's Tree-Sitter `AST` module [6]. The list of selected repositories, along with the tool for data collection and filtering, is available in the supplementary material [12].

We applied three filtering criteria in a single pass to the 479,345 initially collected functions: at least one `try-except` block, non-trivial size (≥5 statements), and no nested or incomplete `try-except` blocks. Because the criteria were applied conjunctively, we report only the collective reduction (Table 1). After filtering, 25,348 functions remained (~5.2%), meaning the combined filters removed 453,997 functions.

To prepare the dataset for our four evaluation tasks, we systematically removed the original `try-except` blocks from these Python functions, simulating code segments without exception handling. These modified code snippets served as input to the LLMs, while the originals (with `try-except` blocks) were retained as developer-written reference implementations for evaluation. We define a **code snippet** as a complete Python function, including its signature and all internal statements; for positive samples, the snippet represents the function with its original `try-except` blocks removed. Finally, to ensure statistical robustness, we selected a random sample with a 95% confidence level and a 5% margin of error, yielding a final dataset of 379 representative code snippets (out of the filtered population). Table 1 summarizes the population, filters, final sample size, and dataset usage per research question.

Table 1: Sample Selection and Usage per Research Question

Stage / RQ	Count
Initial Population	479,345 functions
Filters Applied	1. Must contain ≥1 `try-except` block 2. Non-trivial (≥5 executable statements) 3. No nested or incomplete `try-except`
Total Filtered Out	453,997 functions (applied collectively)
Filtered Population	25,348 functions (approx. 5.2% retained)
Selected Sample	379 code snippets (95% conf., 5% margin of error)
Usage per RQ	
RQ_1 (Necessity)	379 positive + 379 negative (758 total)
RQ_2 (Localization)	379 code snippets
RQ_3 (Type Recom.)	379 code snippets
RQ_4 (Generation)	379 code snippets (100 for human eval)

For **Task 1 (Exception Handling Necessity Detection)**, we aimed to evaluate whether LLMs can determine if a given code snippet requires exception handling mechanisms. Using the 379 positive samples (functions containing `try-except` blocks collected from the process mentioned above), we constructed an equal number of negative samples by randomly selecting functions without any `try-except` structure from the initial dataset. This produced a balanced binary dataset for evaluating accuracy.

For **Task 2 (Try Block Localization)**, we converted each code snippet into a sequence of individual statements and annotated which ones were enclosed within a `try` block. Each statement inside a `try` block was labeled as `1`, while others were labeled as `0`. This

[2] https://seart-ghs.si.usi.ch/

binary representation allowed us to evaluate how accurately LLMs can localize the code regions that require exception handling. For example, consider a function composed of six statements, where only the fourth and fifth are protected by a `try` block. The resulting annotation array would be `[0, 0, 0, 1, 1, 0]`, indicating that statements 4 and 5 are enclosed by exception handling.

For **Task 3 (Exception Type Recommendation)**, we extracted all exception types appearing in the `except` clauses of the collected functions. Each function could have one or more handled exceptions (e.g., `FileNotFoundError`, `KeyError`, `ValueError`), which we treated as multi-class labels. This dataset then evaluated how effectively LLMs can infer appropriate exception types given a code snippet and its context.

Finally, for **Task 4 (Except Handler Generation)**, we extracted all `except` blocks from the collected dataset, capturing both the exception type and the statements contained within each handler, preserving the contextual relationships between the handled exception and its corresponding recovery logic. These annotated blocks were used to evaluate how LLMs can generate contextually relevant, semantically correct exception handlers.

2.2 Prompt Engineering

Prompt design plays a central role in evaluating LLMs in a prompt-based setting, as small changes to task instructions, examples, or output constraints can substantially affect model behavior. To systematically analyze this effect, we evaluate four prompting strategies based on previous studies [20] across all exception handling tasks: default, one-shot, few-shot, and chain-of-thought (CoT). These strategies allow us to assess whether additional examples or explicit reasoning instructions improve LLM effectiveness for different types of exception handling tasks.

The default prompt consists of a concise instruction that directly describes the task and specifies the expected output format, without examples or explicit reasoning steps. The one-shot prompt includes one illustrative input-output example before the target code snippet. This example is selected from our initial dataset and demonstrates the expected format and reasoning pattern for the corresponding exception handling task. The few-shot prompt includes up to three representative examples before the target snippets, demonstrating distinct exception handling mechanisms. Finally, the Chain-of-Thought (CoT) prompt explicitly instructs the model to reason step by step about the exception handling decision before producing the final output, allowing us to evaluate whether explicit reasoning benefits tasks that require control-flow or semantic analysis.

Prompts were developed through an iterative, model-agnostic process to avoid bias toward the evaluated models. All refinements were performed using an external LLM (Gemini) not included in the experiments, starting from basic task descriptions and progressively clarifying the programming language, task objective, expected output format, and reasoning instructions for CoT prompts. This process resulted in a unified set of prompt templates for all exception handling tasks. In all prompting strategies, the models receive only the isolated target function, including its signature, without external project context such as imports, class definitions, called-method implementations, module-level variables, or dependencies.

For one-shot and few-shot settings, examples were randomly selected from the curated dataset and automatically injected into the prompts, along with task instructions and example input-output pairs, to form a combined prompt that included the target code snippet. For example, Figure 2 shows a representative one-shot prompt used for Task 1, where the goal is to decide whether exception handling is necessary. The prompt is composed of three main parts: i) the **task instruction**, specifying what the model should determine; ii) the **example pair**, showing one sample input and its expected output; and iii) the **target query**, containing the new code snippet to evaluate.

```
### Task: Determine if exception handling is
    necessary.

#### Example
Code:
result = 10 / user_input
Expected answer: Yes
Reason: user_input could be zero (
    ZeroDivisionError).

#### Now analyze this code:
data = load_data()
data[0] = process(data[0])
Answer only "Yes" or "No" depending on whether a
    try-except block is needed.
```

Figure 2: Example of a one-shot prompt used for Task 1 (Exception Handling Necessity Detection).

The same template structure was adapted to all exception handling tasks, using consistent instructions, output formats, and task-specific examples. The complete prompts are available in the supplementary material [12].

2.3 Model Selection and Experimental Setup

We executed the prepared prompts across open-source and proprietary LLMs to compare models with different architectures, sizes, and deployment settings. The selected models include locally deployable open-source models that support reproducibility and on-premises execution, as well as compact proprietary models that reflect current commercial coding assistants. This setup enables a controlled comparison of trade-offs between openness, deployment flexibility, inference efficiency, and task effectiveness.

The evaluated models were **Phi-4** [23], an open-source model with approximately 14B parameters designed for general reasoning and code-understanding tasks; **CodeLlama** [22], an open-source model with approximately 7B parameters specialized in programming-related tasks; and **GPT-4 Mini** [25], **Claude Haiku 4.5** [2], and **Gemini 3 Flash** [10], three compact proprietary models designed for efficient reasoning and code generation.

All models were executed with default parameters and without model-specific tuning. Open-source models were run locally using the *Ollama* framework on a workstation with an Intel i9-14900 CPU (24 cores), 64GB RAM, and an NVIDIA RTX 4000 GPU (20GB).

Proprietary models (Claude Haiku 4.5, GPT-4 Mini, and Gemini 3-flash-preview) were accessed via official APIs (Anthropic, OpenAI, and Google GenAI SDKs) on vendor-hosted infrastructure. The full pipeline required approximately 28 hours for open-source models and 12 additional hours for proprietary models.

2.4 Evaluation Metrics

To evaluate LLM effectiveness across the research questions (four exception handling tasks and the complementary exception handling quality analysis), we adopted widely used metrics from prior work on code understanding and generation [36]. Each research question focuses on a specific reasoning ability (e.g., classification, code generation) that requires a different evaluation metric. Details are summarized as follows.

For **RQ_1 (Exception Necessity Detection)** and **RQ_2 (Try Block Localization)**, we modeled the problems as classification tasks. We report Accuracy, Precision, Recall, and F-score. They are computed as Precision $= \frac{TP}{TP+FP}$, Recall $= \frac{TP}{TP+FN}$, and F-Score $= \frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision}+\text{Recall}}$, where TP, FP, and FN denote true positives, false positives, and false negatives, respectively. Accuracy measures the proportion of correct predictions, while Precision and Recall capture the trade-offs between false positives and false negatives. The F-score summarizes their harmonic balance and is particularly useful when dealing with class imbalance.

For **RQ_3 (Exception Type Recommendation)**, we evaluate recommendation quality using four complementary metrics: Exact Accuracy, Hierarchy Accuracy, Weighted F1, and Macro F1. Exact Accuracy measures whether the predicted exception matches the reference label exactly. Hierarchy Accuracy rewards predictions that correspond to semantically related parent or child exception types in Python's exception hierarchy. Weighted F1 emphasizes common exception types according to class frequency, whereas Macro F1 gives equal importance to both common and rare types.

For **RQ_4 (Exception Handler Generation)**, we compared generated and developer-written handlers using CrystalBLEU [9], Jaccard Similarity [11], and Exact Match. We also used Python's Abstract Syntax Tree parser (`ast.parse`) to identify syntactically valid generated handlers. Because models often produced isolated `except` blocks, each output was embedded in a minimal valid `try` structure before parsing. To complement the automatic metrics, we conducted a human evaluation of 100 randomly selected outputs (20 per model). Each sample contained (i) the input function with its original `try-except` block removed, (ii) the developer-written `except` block as the reference handler, and (iii) the generated handler. Three evaluators with Software Engineering backgrounds and 1–5 years of Python experience independently rated all samples on a five-point scale for semantic similarity and logical coherence. Evaluators were unaware of the model identities, and agreement was measured using Krippendorff's alpha. Finally, we assessed the syntactic validity of all generated handlers using Python's `ast.parse`. All outputs, including syntactically invalid ones, were retained in the automatic and human evaluations to avoid survivorship bias.

For **RQ_5 (Exception-Related Quality Violations)**, we applied Pylint[3] to the generated handlers and their developer-written references. We analyzed four exception-related rules: Bare Except (W0702), Try-Except-Pass (W0706), Missing Exception Chaining (W0707), and Broad Exception Catching (W0718). We report the average number of violations per sample, the percentage difference from the references, and their statistical significance.

[3]https://www.pylint.org/

3 Results

In this section, we describe the results and answer the research questions.

3.1 RQ_1: How effectively can LLMs determine whether exception handling is necessary?

Table 2 presents the results for the necessity prediction task across both open-source and proprietary LLMs. Overall, GPT-4 Mini with the few-shot prompt achieved the highest observed values, with accuracy (0.708), precision (0.654), and f-score (0.752). This combination (prompt style and LLM) suggests that providing illustrative examples helps the model better distinguish when `try-except` blocks are necessary, enabling it to focus on key contextual indicators without overgeneralizing.

Table 2: `Try-except` necessity results by model and prompt.

Model	Prompt	Accuracy	Precision	Recall	F-Score
CodeLlama	Default	0.533	0.544	0.406	0.465
CodeLlama	One-shot	0.530	0.527	0.596	0.559
CodeLlama	Few-shot	0.493	0.496	0.763	0.601
CodeLlama	CoT	0.512	0.512	0.525	0.518
Phi-4	Default	0.648	0.592	0.947	0.729
Phi-4	One-shot	0.635	0.581	0.966	0.725
Phi-4	Few-shot	0.620	0.583	0.842	0.689
Phi-4	CoT	0.586	0.547	**1.000**	0.707
GPT-4 Mini	Default	0.670	0.617	0.900	0.732
GPT-4 Mini	One-shot	0.690	0.632	0.908	0.745
GPT-4 Mini	Few-shot	**0.708**	**0.654**	0.884	**0.752**
GPT-4 Mini	CoT	0.567	0.536	0.995	0.697
Claude Haiku 4.5	Default	0.681	0.623	0.913	0.741
Claude Haiku 4.5	One-shot	0.672	0.609	0.958	0.745
Claude Haiku 4.5	Few-shot	0.668	0.611	0.923	0.735
Claude Haiku 4.5	CoT	0.658	0.595	0.989	0.743
Gemini 3 Flash	Default	0.649	0.609	0.831	0.703
Gemini 3 Flash	One-shot	0.656	0.599	0.939	0.732
Gemini 3 Flash	Few-shot	0.673	0.614	0.934	0.741
Gemini 3 Flash	CoT	0.562	0.543	0.776	0.639

Among the evaluated models, proprietary models generally achieved higher effectiveness. GPT-4 Mini achieved a balanced effectiveness across metrics, particularly under one-shot and *few-shot* prompting, while Claude Haiku 4.5 showed stable effectiveness across prompt styles (f-scores between 0.735 and 0.745). Gemini 3 Flash remained competitive but was more sensitive to prompt formulation, improving under few-shot prompting and declining under CoT.

Regarding open-source models, Phi-4 achieved the highest effectiveness on this task, with an F-score of 0.729 (default) and a recall of 1.000 under CoT. The perfect recall, however, comes from Phi-4 predicting "Yes" on essentially every input under CoT, which means it is degenerate as a classifier even though the metric looks strong; the corresponding precision (0.547) confirms this. We therefore caution

against reading the recall column in isolation. In contrast, CodeLlama achieved lower effectiveness across configurations, although few-shot examples substantially improved its recall (0.763). Overall, proprietary models tended to outperform open-source baselines in precision and F-score, while Phi-4 remained competitive in recall-oriented scenarios (it identified all risky cases but also flagged some safe code unnecessarily). This finding highlights a trade-off between balanced predictive effectiveness and high-coverage detection.

Consistent with prior research on binary code reasoning tasks [14, 19], lightweight contextual prompts such as *one-shot* and *few-shot* were more effective than longer reasoning traces for necessity prediction. In contrast, CoT prompting tended to maximize recall, indicating that models became more conservative and classified more cases as requiring exception handling mechanisms.

Summary: LLMs show promising results for identifying whether exception handling mechanisms is needed. GPT-4 Mini with few-shot prompting achieved the highest F-score (0.752), suggesting that LLMs can support early-stage exception handling triage.

3.2 RQ_2: How effectively can LLMs identify which statements should be enclosed in a `try` block?

Table 3 presents the results for the statement localization task, where the goal is to identify which specific statements should be enclosed by a `try` block. Overall, Gemini 3 Flash with the CoT prompt achieved the highest effectiveness, obtaining the highest accuracy (0.949), precision (0.687), recall (0.786), and f-score (0.733). This result indicates that reasoning-oriented prompting substantially improves fine-grained localization by helping the model to analyze the execution flow and identify exception-prone statements more precisely.

Table 3: Try block localization results by model and prompt.

Model	Prompt	Accuracy	Precision	Recall	F-Score
CodeLlama	Default	0.892	0.319	0.155	0.209
CodeLlama	One-shot	0.903	0.414	0.155	0.226
CodeLlama	Few-shot	0.885	0.350	0.294	0.320
CodeLlama	CoT	0.904	0.372	0.086	0.139
Phi-4	Default	0.875	0.271	0.225	0.246
Phi-4	One-shot	0.889	0.167	0.053	0.081
Phi-4	Few-shot	0.905	0.000	0.000	0.000
Phi-4	CoT	0.889	0.085	0.021	0.034
GPT-4 Mini	Default	0.901	0.458	0.610	0.523
GPT-4 Mini	One-shot	0.891	0.418	0.561	0.479
GPT-4 Mini	Few-shot	0.862	0.327	0.487	0.391
GPT-4 Mini	CoT	0.884	0.402	0.561	0.469
Claude Haiku 4.5	Default	0.923	0.564	0.684	0.618
Claude Haiku 4.5	One-shot	0.867	0.338	0.476	0.396
Claude Haiku 4.5	Few-shot	0.927	0.579	0.743	0.651
Claude Haiku 4.5	CoT	0.933	0.621	0.701	0.658
Gemini 3 Flash	Default	0.923	0.559	0.711	0.626
Gemini 3 Flash	One-shot	0.893	0.437	0.610	0.509
Gemini 3 Flash	Few-shot	0.915	0.525	0.668	0.588
Gemini 3 Flash	CoT	**0.949**	**0.687**	**0.786**	**0.733**

Claude Haiku 4.5 demonstrated consistent effectiveness across prompting strategies, with high accuracy and f-scores ranging from 0.396 to 0.658, particularly benefiting from few-shot and CoT prompting. GPT-4 Mini also achieved competitive results, especially under the default prompt (f-score = 0.523), but showed limited improvement with additional context (e.g., few-shot). Among open-source models, CodeLlama achieved the highest observed f-score, 0.320, with few-shot prompting, indicating that examples help smaller models generalize. In contrast, Phi-4 maintained relatively high accuracy but exhibited low recall and F-score, suggesting overly conservative predictions.

Proprietary models outperformed the open-source models on balanced metrics such as precision and F-score, while open models generally maintained moderate accuracy. These results suggest that identifying which statements should be enclosed in a `try` block depends on contextual reasoning about APIs, side effects, and data dependencies. Accordingly, CoT prompting was more beneficial in this task than in RQ_1, where shorter prompts were sufficient. Overall, statement localization appears substantially more challenging than `try-except` necessity and benefits from good reasoning capabilities in the proprietary models.

Summary: LLMs help identify which statements should be enclosed in a `try` block. Gemini 3 Flash with CoT achieved the best effectiveness (F-score = 0.733), and proprietary models consistently outperformed open-source models.

3.3 RQ_3: How effectively can LLMs recommend exception types for `except` clauses?

Table 4 presents the results for the exception type recommendation task. Among all evaluated models, Gemini 3 Flash (CoT) achieved the highest Exact Accuracy (0.422) and the highest Hierarchy Accuracy (0.493), indicating the strongest ability to predict either the exact exception type or a semantically related parent class. In contrast, Claude Haiku 4.5 (CoT) achieves balanced classification results across metrics, with the highest Weighted F1 (0.203) and Macro F1 (0.275), suggesting better results across both frequent and infrequent exception types.

Proprietary models generally achieved higher effectiveness. Claude Haiku 4.5 showed the most consistent effectiveness across prompt styles, with all configurations remaining competitive, and its CoT variant leading the F1-based metrics. GPT-4 Mini also remained competitive, particularly under CoT prompting (Exact Accuracy = 0.359; Hierarchy Accuracy = 0.438), although it underperformed Claude Haiku 4.5 and Gemini 3 Flash on weighted and macro F1 scores. Among the open-source models, Phi-4 delivered the best results, achieving Exact Accuracy of 0.274 (Few-shot) and Weighted F1 of 0.160 (Default), approaching GPT-4 Mini on selected metrics and indicating that recent compact open-source models can be viable, low-cost alternatives for exception recommendation. In contrast, CodeLlama achieved lower effectiveness across metrics.

Prompt effectiveness also differed by model family. As observed in RQ_2, CoT prompting proved beneficial for several proprietary models, especially Gemini 3 Flash and Claude Haiku 4.5. However, for open-source models, Few-shot prompting often outperformed CoT, particularly for Phi-4 and CodeLlama, suggesting that explicit

Table 4: Exception type recommendation results by model and prompt.

Model	Prompt	Exact Acc.	Hierarchy Acc.	Weighted F1	Macro F1
CodeLlama	Default	0.018	0.018	0.023	0.035
CodeLlama	One-shot	0.061	0.095	0.052	0.083
CodeLlama	Few-shot	0.113	0.177	0.062	0.071
CodeLlama	CoT	0.037	0.061	0.040	0.038
Phi-4	Default	0.251	0.280	0.160	0.139
Phi-4	One-shot	0.206	0.261	0.150	0.179
Phi-4	Few-shot	0.274	0.340	0.159	0.140
Phi-4	CoT	0.240	0.274	0.150	0.086
GPT-4 Mini	Default	0.298	0.377	0.143	0.146
GPT-4 Mini	One-shot	0.230	0.290	0.163	0.173
GPT-4 Mini	Few-shot	0.227	0.322	0.137	0.155
GPT-4 Mini	CoT	0.359	0.438	0.152	0.161
Claude Haiku 4.5	Default	0.314	0.383	0.183	0.254
Claude Haiku 4.5	One-shot	0.261	0.311	0.175	0.172
Claude Haiku 4.5	Few-shot	0.311	0.391	0.196	0.233
Claude Haiku 4.5	CoT	0.356	0.435	**0.203**	**0.275**
Gemini 3 Flash	Default	0.414	0.485	0.201	0.236
Gemini 3 Flash	One-shot	0.285	0.325	0.194	0.205
Gemini 3 Flash	Few-shot	0.354	0.427	0.193	0.199
Gemini 3 Flash	CoT	**0.422**	**0.493**	0.191	0.198

examples may be more helpful than reasoning for smaller local models.

Finally, exception-type recommendation remains challenging. The gap between Exact and Hierarchy Accuracy suggests that models often identify semantically related exception classes but fail to predict the precise developer-selected type.

Summary: LLMs showed limited effectiveness in recommending precise exception types. Gemini 3 Flash achieved the highest Exact and Hierarchy Accuracy, while Claude Haiku 4.5 obtained the best F1-based scores.

3.4 RQ_4: How effectively can LLMs generate complete exception handling blocks?

Table 5 presents the results for the exception handler generation task. Overall, the proprietary models achieved the highest observed values, though open-source models also performed competitively. Among all evaluated LLMs, Claude Haiku 4.5 (Default) achieved the highest lexical and structural similarity to the reference code, with CrystalBLEU (0.104) and Jaccard Similarity (0.353) scores. Gemini 3 Flash followed closely, particularly under CoT prompting, achieving CrystalBLEU = 0.101 and Jaccard = 0.346.

GPT-4 Mini produced intermediate effectiveness overall under Few-shot prompting (CrystalBLEU = 0.032; Jaccard = 0.294; Exact Match = 0.021), although the results were lower than those of Claude and Gemini. Regarding open-source models, Phi-4 substantially outperformed CodeLlama across all metrics and achieved the highest effectiveness under Few-shot prompting, with CrystalBLEU = 0.032, Jaccard = 0.268, and Exact Match = 0.026. Phi-4 matched or slightly exceeded GPT-4 Mini on CrystalBLEU in selected settings, reinforcing the low-cost-alternative pattern observed for RQ_3 (§3) The Exact Match metric showed that the best models reproduced the reference exception handler exactly in 4-6% of the evaluated cases. Although exact replication remained difficult, these results indicate that LLMs can occasionally generate code equivalent to developer-written implementations.

Table 5: Exception handler generation results by model and prompt.

Model	Prompt	CrystalBLEU	Jaccard	Exact Match
CodeLlama	Default	0.026	0.216	0.003
CodeLlama	One-shot	0.021	0.203	0.011
CodeLlama	Few-shot	0.021	0.189	0.008
CodeLlama	CoT	0.022	0.194	0.008
Phi-4	Default	0.023	0.248	0.003
Phi-4	One-shot	0.028	0.250	0.018
Phi-4	Few-shot	0.032	0.268	0.026
Phi-4	CoT	0.024	0.214	0.005
GPT-4 Mini	Default	0.031	0.252	0.011
GPT-4 Mini	One-shot	0.028	0.243	0.013
GPT-4 Mini	Few-shot	0.032	0.294	0.021
GPT-4 Mini	CoT	0.024	0.250	0.008
Claude Haiku 4.5	Default	**0.104**	**0.353**	0.042
Claude Haiku 4.5	One-shot	0.081	0.301	0.047
Claude Haiku 4.5	Few-shot	0.091	0.324	**0.061**
Claude Haiku 4.5	CoT	0.082	0.325	0.037
Gemini 3 Flash	Default	0.100	0.353	0.040
Gemini 3 Flash	One-shot	0.071	0.304	0.029
Gemini 3 Flash	Few-shot	0.090	0.326	0.034
Gemini 3 Flash	CoT	0.101	0.346	0.047

Syntax validity varied across prompting strategies. More than 99% of handlers generated with one-shot and few-shot prompts were syntactically valid, whereas validity decreased under the default and CoT settings; for example, GPT-4 Mini achieved 60.2% validity with CoT prompting. Invalid outputs commonly contained malformed code or Markdown artifacts, suggesting that examples improved adherence to the expected format. Because invalid handlers were retained in all evaluations, these errors may also have contributed to lower lexical-similarity scores and human ratings. Detailed results are reported in the supplementary material [12].

Moreover, Figure 3 presents the human evaluation results using a five-point Likert scale, in which raters assessed the semantic similarity and logical coherence of generated `except` blocks relative to the reference implementations. Overall, ratings were concentrated at scores 1 and 2, indicating that the generated exception handling code was often perceived as weakly aligned with the human-written references. Across models, low-similarity ratings (scores 1–2) accounted for approximately 70%–93% of all evaluations. Inter-rater agreement, measured using Krippendorff's alpha [17], was 0.60, indicating moderate agreement but remaining below the 0.667 threshold for tentative reliability. We therefore interpret the human ratings as directional rather than definitive.

Among the evaluated models, Gemini 3 Flash achieved a more consistent human assessment, with the lowest proportion of score 1 ratings (48.3%) and the highest overall distribution across higher similarity levels (scores 4–5), totaling approximately 13%. Claude Haiku 4.5 followed, with a more balanced distribution across scores 2–5, including visible proportions of high-similarity responses. In contrast, CodeLlama exhibited lower similarity ratings, with 77% of responses concentrated at score 1 and only marginal representation in higher similarity levels (e.g., 7% at score 4). Similarly, GPT-4 Mini and Phi-4 showed strong concentration in low-similarity ratings,

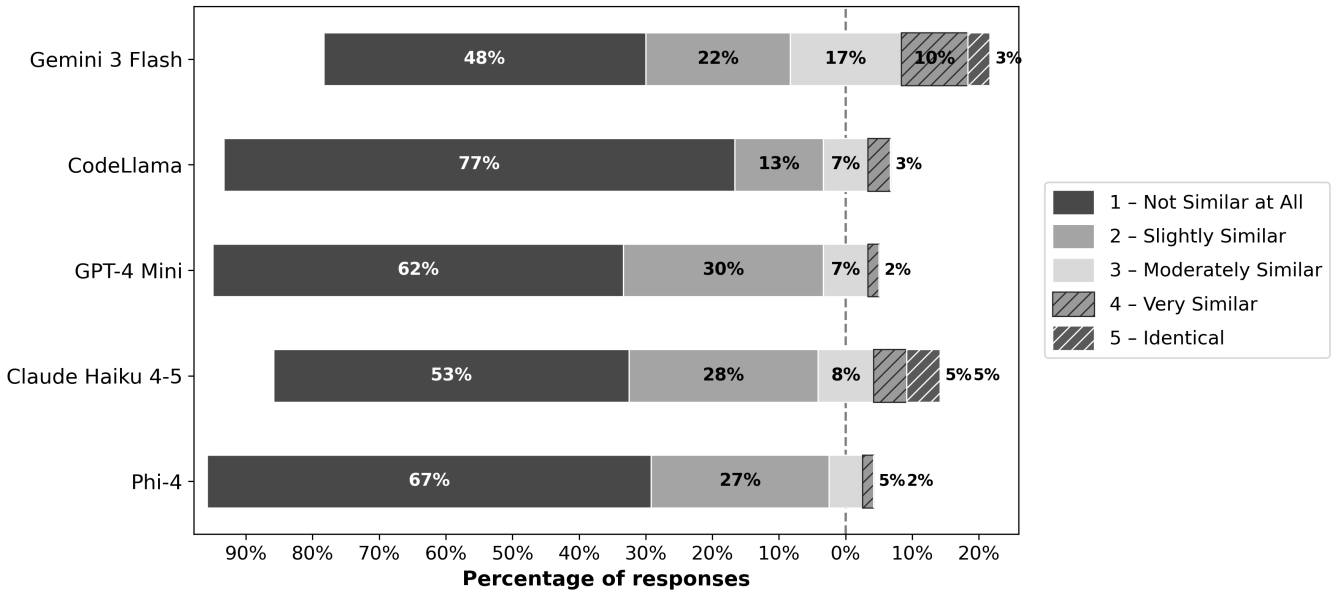

Figure 3: Human evaluation of generated except blocks using a 5-point Likert scale. Each bar shows the percentage of ratings assigned by evaluators for each model, where 1 indicates "Not Similar at All" and 5 indicates "Highly Similar or Identical".

with 62% and 67% of responses scoring 1, respectively, and very limited presence in scores 4 and 5.

These findings complement the automatic metrics in Table 5. Although some models achieved competitive lexical similarity scores, human raters consistently judged most generated handlers as weakly aligned with the reference implementations. Nevertheless, Gemini 3 Flash produced a non-negligible proportion of outputs that were moderately to highly similar (approximately 30% of its ratings were in the 3–5 range), indicating that LLMs can occasionally generate semantically meaningful exception handling code. Overall, these results reinforce that lexical similarity alone is insufficient to capture semantic adequacy, highlighting the need for human evaluation, richer contextual inputs, or post-generation validation when recommending exception handling mechanisms.

Summary: LLM-generated handlers showed limited alignment with developer-written code. Claude Haiku 4.5 achieved the best automatic scores, while Gemini 3 Flash received the strongest human ratings. Overall, generated handlers remain unreliable without human validation.

3.5 RQ_5: What exception-related quality violations are introduced by LLM-generated exception handlers?

Table 6 presents the average number of exception-related Pylint violations in generated and developer-written handlers, the percentage increase over the reference implementations (Δ), and the statistical significance of that increase. Overall, all models generated more violations than developer-written code, with an average of 0.166 violations per sample. Among them, Gemini 3 Flash produced outputs with fewer exception-related quality violations (0.290), followed by CodeLlama (0.305), GPT-4 Mini (0.331), Phi-4 (0.382), and Claude Haiku 4.5 (0.388). All differences were statistically significant ($p < 0.05$).

Although all models exceeded the reference baseline, Gemini 3 Flash showed a smaller increase, generating 74.7% more exception-related quality violations than the reference handlers, followed by CodeLlama, which showed an increase of 83.7%. In contrast, Phi-4 and Claude Haiku 4.5 showed a larger increase, with 130.1% and 133.7% more violations, respectively. These results show that greater similarity to reference handlers does not necessarily imply better exception handling quality. For example, although Claude Haiku 4.5 achieved the highest automatic generation scores in Task 4, it also produced more exception-related quality violations. This suggests that generated handlers should be evaluated not only for lexical or structural similarity to reference implementations, but also for adherence to exception handling quality guidelines.

Table 6: Average exception-related quality violations in generated handlers. Lower values are better.

Model	Avg Violations	GT Avg Violations	Δ Violations (%)	$p < 0.05$
Gemini 3 Flash	**0.290**	0.166	+74.7%	✓
CodeLlama	0.305	0.166	+83.7%	✓
GPT-4 Mini	0.331	0.166	+99.4%	✓
Phi-4	0.382	0.166	+130.1%	✓
Claude Haiku 4.5	0.388	0.166	+133.7%	✓
Average	0.339	0.166	+104.2%	✓

Table 7 provides a finer-grained view of these violations by grouping them according to Pylint rule type. Violations were concentrated in two dominant categories: W0707 (Missing Exception Chaining), with 1,277 cases, and W0718 (Broad Exception Catching), with 995 cases. W0707 occurs when a new exception is raised without explicit chaining through the `from` keyword, which can

obscure the original error context and hinder debugging, whereas W0718 occurs when the broad `Exception` class is caught instead of a more specific exception type, potentially masking unexpected bugs and causing silent failures. In contrast, W0702 (Bare Except) was rare, with only 27 occurrences, all generated by CodeLlama. These results indicate that models generally avoid completely bare `except` clauses but still frequently generate handlers that catch overly generic exceptions or re-raise exceptions without preserving the original context.

Summary: LLM-generated handlers produced 104.2% more exception-related quality violations than developer-written references on average. The most frequent violation types were Missing Exception Chaining and Broad Exception Catching, indicating that syntactically valid generated handlers may still violate exception handling quality guidelines.

4 Implications

Implications for Developers and Tools. Our findings suggest that LLMs are better suited to support exception handling within human-in-the-loop workflows than autonomously generate exception handling tasks. More favorable results were observed for early-stage tasks, namely detecting the need for exception handling and localizing statements that should be enclosed in a `try` block, with F-scores of 0.752 and 0.733, respectively. These capabilities could support code review by highlighting potentially risky regions for developer inspection, where false positives primarily increase review effort rather than directly alter the code. In contrast, exception-type recommendation and handler generation remain challenging: generated handlers often diverged from developer-written implementations and introduced 104.2% more exception-related quality violations on average. Thus, directly adopting such outputs may reduce exception handling quality, even when the generated code is syntactically valid or lexically similar to the reference. LLM-based tools should therefore follow a multi-stage workflow that identifies risky regions, generates candidate handlers, and validates them through static analysis, testing, and developer review. This workflow uses the strengths of LLMs while mitigating their limitations in exception-type selection, handler generation, and code quality. These findings reinforce that LLMs are most effective within a human-in-the-loop workflow. While they can help identify the need for exception handling, generated exception handlers should be treated as suggestions that require validation.

Implications for Researchers. Our findings suggest that exception handling is better studied as a sequence of distinct reasoning tasks rather than as a single generation problem. The effectiveness gap between detection and localization, on the one hand, and exception-type recommendation and handler generation, on the other, reinforces the value of this decomposition for both model design and evaluation. Notably, Claude Haiku 4.5 achieved the highest CrystalBLEU score while producing 133.7% more exception-related quality violations than the developer-written references. This divergence shows that lexical similarity does not reliably reflect semantic adequacy or exception-handling quality. Future research should develop evaluation frameworks that jointly assess semantic correctness, exception-handling quality, and runtime behavior. Also, promising directions include integrating LLMs with static analysis, API knowledge, and testing.

5 Threats to Validity

There are several potential threats to the validity of our results.

Prompt Engineering. Prompt construction and selection were manually guided, which introduces potential bias and variability. Different phrasing, ordering of examples, or contextual instructions could influence model behavior. To mitigate this, we standardized prompt templates through iterative refinement and evaluated multiple styles (default, one-shot, few-shot, and CoT) to capture variability. Moreover, we craft prompts using a separate dataset with a specific LLM (ChatGPT) that is not used in our evaluation.

Selected Projects. Our dataset was derived from Python projects such as LangChain, FastAPI, and Pytest, selected due to their active maintenance and popularity. Although these projects represent realistic and diverse use cases, they may not generalize to other domains or programming languages.

Dataset Filtering. Our criteria retained 5.2% of the collected functions, largely because most Python functions do not contain exception handling (Peng et al. found usage between 6.74% to 16.36% [26]). Also, we excluded trivial functions ($\leq$5 executable statements) and nested or incomplete handlers to provide sufficient context and avoid ambiguous labels, respectively. However, these filters may omit short functions or complex real-world patterns. Future work should adopt stratified sampling by function size, handler complexity, project, and exception type.

Conclusion Validity. The sample size and task definitions are sufficient for exploratory analysis but may not support strong statistical generalization. We mitigated this by employing multiple models and prompt styles, comparing across distinct evaluation metrics. Nevertheless, replication with larger datasets and alternative model families (e.g., ChatGPT-5, Claude Opus) would corroborate confidence in our conclusions.

6 Related Work

Prior automation of exception handling has progressed along four lines: identifying missing API error-handling calls [1], detecting improper exception messages [33], recommending exception types [37], recommending where to handle an exception [13], and recommending complete exception-handling code [7, 24, 36]. Most of this work has been evaluated on Java and relies on task-specific model training, leaving Python, which has a different exception model, essentially untouched at the level of automated tooling.

Acharya and Xie [1] proposed a framework to automatically mine API error-handling specifications from client code, identifying 62 specifications and 264 real defects in open-source projects. Xu and Zhong [33] introduced *EXMINER*, which detects inconsistencies in thrown exceptions by mining and comparing patterns in exception messages. Zhong [37] developed *ThEx*, a classifier that predicts contextually appropriate exception types based on code features, improving robustness and consistency.

Complementary studies have focused on where exceptions should be handled. Jia et al. [13] presented *EHAdvisor*, which combines static analysis and ranking algorithms to recommend optimal handling locations, achieving up to 86% Top-1 accuracy. Similarly,

Table 7: Detected *Pylint* violations by rule type across LLM-generated exception blocks.

Violation Type	Claude	CodeLlama	Gemini	GPT-4 Mini	Phi-4	Total
W0707 – Missing Exception Chaining	236	250	230	**217**	344	1277
W0718 – Broad Exception Catching	**147**	173	168	285	222	995
W0706 – Try Except Pass	205	13	41	**0**	13	272
W0702 – Bare Except	**0**	27	**0**	**0**	**0**	27
Total Violations	588	463	**439**	502	579	2571

Nguyen et al. [24] proposed *FuzzyCatch*, a fuzzy-logic-based tool that predicts likely exceptions and suggests appropriate handling code, attaining 77% accuracy in empirical evaluations.

Zhang et al. [36] proposed *Learning to Handle Exceptions (L2HE)*, a data-driven framework that learns exception handling practices from large codebases. L2HE jointly predicts whether a `try-catch` block is needed, localizes exception-prone code, and recommends exception types. By using deep neural networks trained on Java code, the authors demonstrated that learned representations can capture exception handling semantics more effectively than rule-based baselines. However, their evaluation was constrained to a single programming language (Java) and required task-specific model training from scratch, limiting reuse and generalization.

Building upon the same research line, Cai et al. [7] introduced *Neurex*, a multi-tasking exception handling recommender that fine-tunes CodeBERT to perform (1) try-catch necessity detection; (2) try-block statement localization; and (3) exception type recommendation. Neurex employs sequence chunking to capture dependencies among API calls and exceptions, achieving f-scores above 70% across tasks and outperforming previous information-retrieval (IR) baselines like *FuzzyCatch* and *XRank*. Its evaluation on over 5,700 Java projects demonstrated the benefits of using code-aware transformers. However, it required extensive fine-tuning and task-specific datasets, and remained limited to the Java ecosystem.

Recent research has explored the use of general-purpose Large Language Models (LLMs) for software engineering tasks. Liu et al. [19] showed that guiding ChatGPT with structured prompts can improve code generation quality, while Jin et al. [15] proposed *InferFix*, a transformer-based framework that integrates static analysis and LLM reasoning for automatic bug repair. However, these studies focus on general code synthesis or repair rather than the specialized problem of exception handling reasoning.

Although prior studies [7, 18, 24, 36] established the feasibility of neural models trained specifically for exception handling, and systems such as L2HE, Neurex, FuzzyCatch, and EHAdvisor provide important baselines for automated exception handling support, a direct comparison with our study is not straightforward. These approaches were designed and evaluated primarily in the Java ecosystem, relying on language-specific assumptions such as Java exception hierarchies, checked-exception semantics, API-call representations, and specialized static-analysis or task-specific training pipelines. Python differs from Java in its exception handling mechanisms, dynamic typing, and absence of checked exceptions, which make adapting these tools to our benchmark a non-trivial engineering and methodological effort. Therefore, our study does not aim to show that prompt-based LLMs outperform specialized exception handling models. Instead, it complements this line of work by evaluating general-purpose LLMs without fine-tuning or task-specific model training across four interpretable Python exception handling tasks: necessity detection, try-block localization, exception-type recommendation, and exception-handler generation. In this sense, our work bridges the gap between fine-tuned neural models and general-purpose LLM reasoning, characterizing the strengths and limitations of prompt-based strategies.

7 Conclusion

This work provides empirical evidence on the use of Large Language Models (LLMs) to support Python exception handling through prompt-based strategies. Evaluating five open-source and proprietary models across four interpretable tasks—and complementing handler generation with both human evaluation and a static-analysis-based quality assessment—we find that LLMs are more effective for early-stage reasoning tasks, such as detecting whether exception handling is needed (best F-score 0.752) and localizing statements that should be enclosed in a `try` block (best F-score 0.733). In contrast, exception-type recommendations and exception handler generation remain more challenging, with low exact-match rates, limited semantic alignment with developer-written handlers, and a higher number of exception-related quality violations. These findings indicate that prompt-based LLMs are better suited as exception handling assistants than as autonomous repair tools. As future work, we plan to incorporate richer project context, such as call graphs and dependency information, and investigate post-processing strategies that combine LLM-generated code with static analysis to improve the quality and reliability of generated handlers.

Artifact Availability

The replication package supporting this study is publicly available on Zenodo [12]. It includes the curated dataset, prompt templates, generated outputs, and scripts used for data processing and evaluation. The artifact is distributed under the MIT License.

Acknowledgements

This work was partially supported by INES.IA (National Institute of Science and Technology for Software Engineering Based on and for Artificial Intelligence) www.ines.org.br, CNPq grant 408817/2024-0. Leopoldo and Baldoino are partially supported by CNPq grants 310405/2025-4 and 309227/2025-9.

References

[1] Mithun Acharya and Tao Xie. 2009. Mining API Error-Handling Specifications from Source Code. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009* (York, UK) *(FASE '09)*. Springer-Verlag, Berlin, Heidelberg, 370–384.

[2] Anthropic. 2025. Claude Haiku 4.5. https://www.anthropic.com/claude. Accessed: 2026-05-01.

[3] Guilherme B. de Pádua and Weiyi Shang. 2018. Studying the Relationship between Exception Handling Practices and Post-Release Defects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 564–575.

[4] Eiji Adachi Barbosa, Alessandro Garcia, and Simone Diniz Junqueira Barbosa. 2014. Categorizing Faults in Exception Handling: A Study of Open Source Projects. In *2014 Brazilian Symposium on Software Engineering*. 11–20. doi:10.1109/SBES.2014.19

[5] Joshua Bloch. 2008. *Effective Java™, Second Edition* (second ed.). Prentice Hall Press, USA.

[6] Max Brunsfeld. 2018. Tree-sitter-a new parsing system for programming tools. In *Strange Loop Conference,. Accessed–. URL: https://www. thestrangeloop. com//tree-sitter—a-new-parsing-system-for-programming-tools. html.*

[7] Yuchen Cai, Aashish Yadavally, Abhishek Mishra, Genesis Montejo, and Tien Nguyen. 2024. Programming Assistant for Exception Handling with CodeBERT. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 94, 13 pages. doi:10.1145/3597503.3639188

[8] Felipe Ebert, Fernando Castor, and Alexander Serebrenik. 2020. A Reflection on "An Exploratory Study on Exception Handling Bugs in Java Programs". In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 552–556. doi:10.1109/SANER48275.2020.9054791

[9] Aryaz Eghbali and Michael Pradel. 2023. CrystalBLEU: Precisely and Efficiently Measuring the Similarity of Code. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 28, 12 pages. doi:10.1145/3551349.3556903

[10] Google DeepMind. 2025. Gemini 3.0 Flash. https://deepmind.google/technologies/gemini/. Accessed: 2026-05-01.

[11] Paul Jaccard. 1912. The Distribution of the Flora in the Alpine Zone. *New Phytologist* 11, 2 (1912), 37–50. arXiv:https://nph.onlinelibrary.wiley.com/doi/pdf/10.1111/j.1469-8137.1912.tb05611.x doi:10.1111/j.1469-8137.1912.tb05611.x

[12] Jairo Souza, Tales Alves, Rodrigo Lima , Ericlecio Araujo, Leopoldo Teixeira, Baldoino Fonseca. 2026. *Exception Handling Mining and LLM Analysis Framework*. doi:10.5281/zenodo.21417821

[13] Xiangyang Jia, Songqiang Chen, Xingqi Zhou, Xintong Li, Run Yu, Xu Chen, and Jifeng Xuan. 2021. Where to Handle an Exception? Recommending Exception Handling Locations from a Global Perspective. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 369–380. doi:10.1109/ICPC52881.2021.00042

[14] Kailun Jin, Chung-Yu Wang, Hung Viet Pham, and Hadi Hemmati. 2024. Can ChatGPT Support Developers? An Empirical Evaluation of Large Language Models for Code Generation. In *Proceedings of the 21st International Conference on Mining Software Repositories* (Lisbon, Portugal) *(MSR '24)*. Association for Computing Machinery, New York, NY, USA, 167–171. doi:10.1145/3643991.3645074

[15] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. 2023. InferFix: End-to-End Program Repair with LLMs. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1646–1656. doi:10.1145/3611643.3613892

[16] Mary Beth Kery, Claire Le Goues, and Brad A. Myers. 2016. Examining Programmer Practices for Locally Handling Exceptions. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. Association for Computing Machinery, New York, NY, USA, 484–487. doi:10.1145/2901739.2903497

[17] Klaus Krippendorff. 2008. Systematic and Random Disagreement and the Reliability of Nominal Data. *Communication Methods and Measures* 2, 4 (12 2008), 323–338. doi:10.1080/19312450802467134

[18] Rongfan Li, Bihuan Chen, Fengyi Zhang, Chao Sun, and Xin Peng. 2022. Detecting Runtime Exceptions by Deep Code Representation Learning with Attention-Based Graph Neural Networks. *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* 00 (2022), 373–384. doi:10.1109/saner53432.2022.00053

[19] Chao Liu, Xuanlin Bao, Hongyu Zhang, Neng Zhang, Haibo Hu, Xiaohong Zhang, and Meng Yan. 2024. Guiding ChatGPT for Better Code Generation: An Empirical Study. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 102–113. doi:10.1109/SANER60148.2024.00018

[20] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. 2023. Pre-train, Prompt, and Predict: A Systematic Survey of Prompting Methods in Natural Language Processing. *ACM Comput. Surv.* 55, 9, Article 195 (Jan. 2023), 35 pages. doi:10.1145/3560815

[21] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. 2023. Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138* (2023).

[22] Meta AI. 2023. Code Llama: Open Foundation Models for Code. https://ai.meta.com/blog/code-llama-large-language-model-coding/. Accessed: 2026-05-04.

[23] Microsoft. 2024. Phi-4: Small Language Models with Strong Reasoning Capabilities. https://www.microsoft.com/en-us/research/blog/phi-4/. Accessed: 2026-05-04.

[24] Tam Nguyen, Phong Vu, and Tung Nguyen. 2020. Code Recommendation for Exception Handling. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1027–1038. doi:10.1145/3368089.3409690

[25] OpenAI. 2024. GPT-4 Mini. https://openai.com/. Accessed: 2026-05-01.

[26] Yun Peng, Yu Zhang, and Mingzhe Hu. 2021. An Empirical Study for Common Language Features Used in Python Projects. *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)* 00 (2021), 24–35. doi:10.1109/saner50967.2021.00012

[27] Dorin Pomian, Abhiram Bellur, Malinda Dilhara, Zarina Kurbatova, Egor Bogomolov, Andrey Sokolov, Timofey Bryksin, and Danny Dig. 2024. EM-Assist: Safe Automated ExtractMethod Refactoring with LLMs. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering* (Porto de Galinhas, Brazil) *(FSE 2024)*. Association for Computing Machinery, New York, NY, USA, 582–586. doi:10.1145/3663529.3663803

[28] Guilherme Bicalho de Pádua and Weiyi Shang. 2017. Studying the Prevalence of Exception Handling Anti-Patterns. *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)* (2017), 328–331. doi:10.1109/icpc.2017.1

[29] Ahmadreza Saboor Yaraghi, Darren Holden, Nafiseh Kahani, and Lionel Briand. 2025. Automated Test Case Repair Using Language Models. *IEEE Trans. Softw. Eng.* 51, 4 (Feb. 2025), 1104–1133. doi:10.1109/TSE.2025.3541166

[30] Atsushi Shirafuji, Yusuke Oda, Jun Suzuki, Makoto Morishita, and Yutaka Watanobe. 2023. Refactoring programs using large language models with few-shot examples. In *2023 30th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 151–160.

[31] Dêmora Bruna Cunha de Sousa, Paulo Henrique Maia, Lincoln Souza Rocha, and Windson Viana. 2018. Analysing the Evolution of Exception Handling Anti-Patterns in Large-Scale Projects: A Case Study. *Proceedings of the VII Brazilian Symposium on Software Components, Architectures, and Reuse on - SBCARS '18* (2018), 73–82. doi:10.1145/3267183.3267191

[32] Dêmora B C de Sousa, Paulo Henrique M. Maia, Lincoln S Rocha, and Windson Viana. 2020. Studying the evolution of exception handling anti-patterns in a long-lived large-scale project. *Journal of the Brazilian Computer Society* 26, 1 (2020), 1. doi:10.1186/s13173-019-0095-5

[33] Lin Xu and Hao Zhong. 2021. Detecting Inconsistent Thrown Exceptions. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. 391–395. doi:10.1109/ICPC52881.2021.00045

[34] Zhiqiang Yuan, Yiling Lou, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, and Xin Peng. 2024. No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. arXiv:2305.04207 [cs.SE] https://arxiv.org/abs/2305.04207

[35] Jiyang Zhang, Yu Liu, Pengyu Nie, Junyi Jessy Li, and Milos Gligoric. 2025. exLong: Generating Exceptional Behavior Tests with Large Language Models. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. 1462–1474. doi:10.1109/ICSE55347.2025.00176

[36] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. 2021. Learning to Handle Exceptions. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (Virtual Event, Australia) *(ASE '20)*. Association for Computing Machinery, New York, NY, USA, 29–41. doi:10.1145/3324884.3416568

[37] Hao Zhong. 2023. Which Exception Shall We Throw?. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 116, 12 pages. doi:10.1145/3551349.3556895