



A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization

[Sungmin Kang, Gabin An], Shin Yoo
Presented on 2024-07-17 by Sungmin
Painting by Elena Katsyura, *Slice of Citrus*, 2013





1. Background

Fault Localization

```
1790  ✓      public LegendItemCollection getLegendItems() {
1791          LegendItemCollection result = new LegendItemCollection();
1792          if (this.plot == null) {
1793              return result;
1794          }
1795          int index = this.plot.getIndexOf(this);
1796          CategoryDataset dataset = this.plot.getDataset(index);
1797          if (dataset != null) {
1798              return result;
1799          }
1800          int seriesCount = dataset.getRowCount();
1801          if (plot.getRowRenderingOrder().equals(SortOrder.ASCENDING)) {
1802              for (int i = 0; i < seriesCount; i++) {
1803                  if (isSeriesVisibleInLegend(i)) {
1804                      LegendItem item = getLegendItem(index, i);
1805                      if (item != null) {
1806                          result.add(item);
1807                      }
1808                  }
1809              }
1810          }
1811      }
```

Explanations for FL results are supported by devs

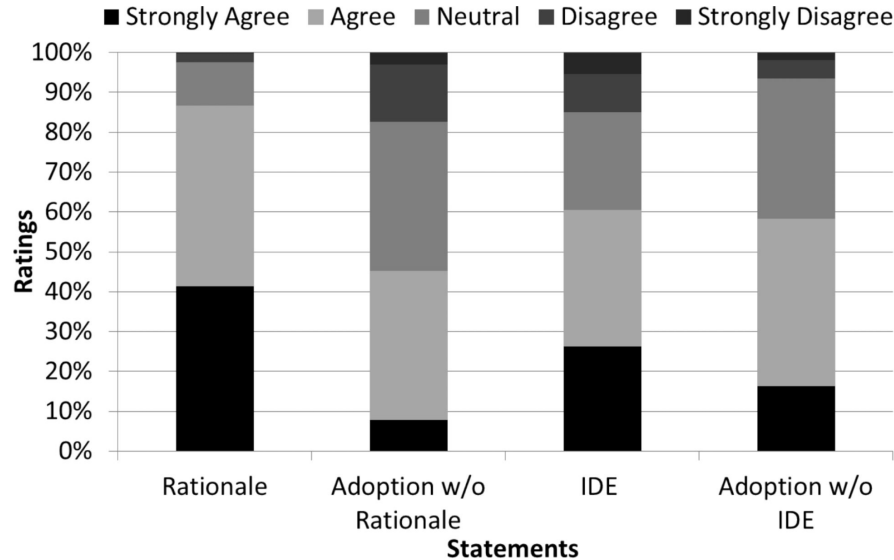


Figure 9: Other Factors Affecting Adoption

Kochhar et al. (2016) find that developers felt that FL results should be accompanied with a rationale of the results.

Developer quotes from Kochhar et al.

- Rationale is needed for bug fixing and code quality improvement
 - *“Because to make a decisions about bug fixing I want to *exactly* know why the automated tool “thinks” that the code have a bug.”*
 - *“... I would also need to provide the fix, so I feel some rationale would also help with that.”*
 - *“Rationale gives understanding which will help in improving the code quality for future”*

In our own developer study as well, developers expressed similar sentiments on the need of explanations/rationales.

However, few FL techniques provide rationales

Table 1. A comparison of existing FL techniques with AutoFL. The precision of SBFL, MBFL, and IRFL was recalculated based on the artifacts of Zou et al. [53]; for other techniques, precision comes from the corresponding papers. Wu et al. [46] only evaluate statement-level FL, so their precision could not be compared.

	Required Artifact	Prec@5 on D4J	Time	Multilang.	Rationale
SBFL	Test suite	61% [53]	minutes	yes	no
MBFL	Test suite	54% [53]	hours	yes	no
IRFL	Bug report	3% ² [53]	seconds	yes	no
CombineFL [53]	All of the above	69% [53]	hours	no	no
DeepRL4FL [24]	Test suite	79% [24]	hours	no	no
UniVal [22]	Pass/Fail test	75% [22]	minutes ³	no	no
SmartFL [52]	Pass/Fail test	70% [52]	minutes	no	no
Wu et al. [46]	Buggy method/class	-	-	yes	yes
AutoFL	Single test	Up to 71%	minutes	yes	yes

In our comparison of FL techniques at the time of writing, only AutoFL and Wu et al. provided a rationale.

Large Language Models (LLMs)



Would you say that you are good at the use of the English language, and explaining things in it?



Yes, I am designed to be proficient in using the English language and explaining concepts clearly. Whether you need help with grammar, writing, understanding complex topics, or simply seeking information, I'm here to assist. How can I help you today?

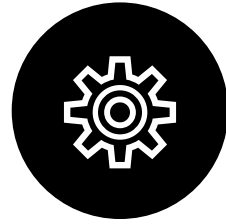


LLMs are trained over massive natural language data, so they are good at natural language.

Application of LLMs to FL is nontrivial

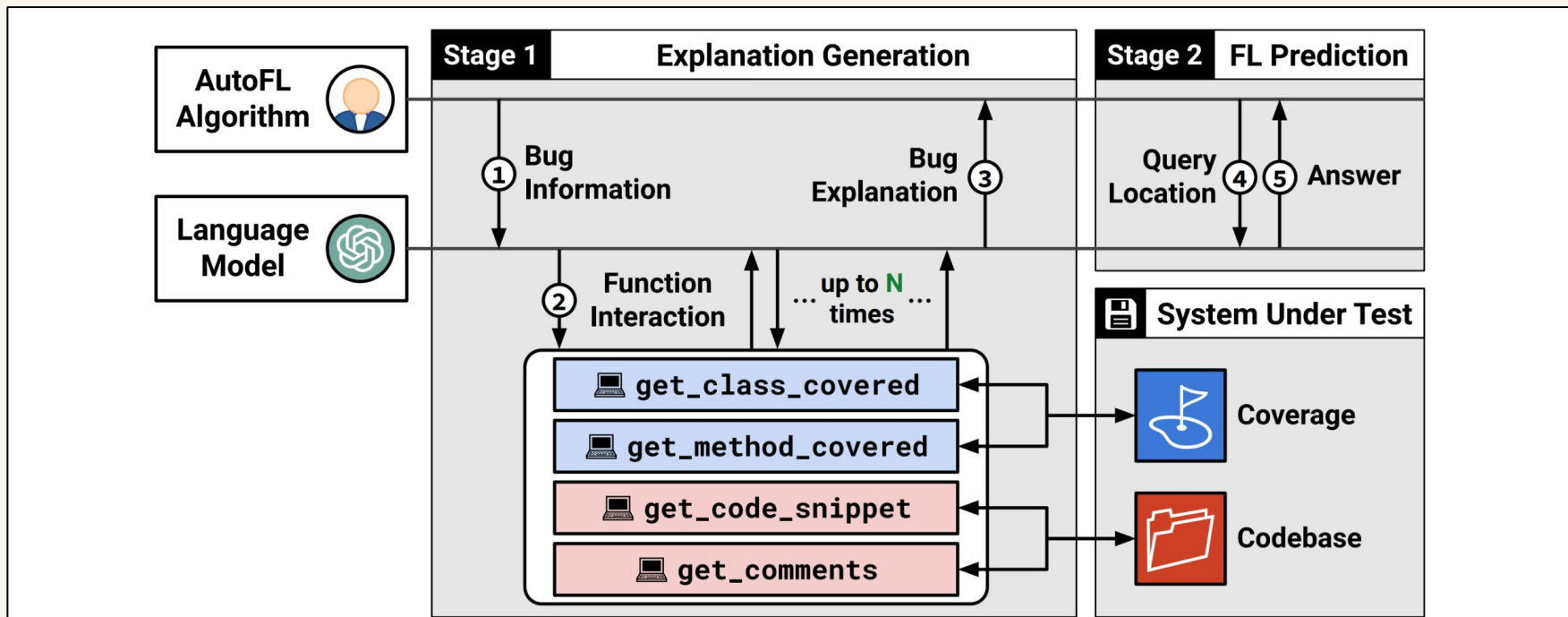
		# Tokens
Context Window	gpt-3.5-turbo-0613	4,096
	gpt-4	8,192
	gpt-4o	128,000
Source Code Size (Only Java files)	Lang	535,075
	Time	547,840
	Closure	1,084,356
	Math	1,696,162
	Chart	1,717,406

LLM context length prevents LLMs from being naively applied to FL.



2. **AutoFL**

Overview



Two large “stages” in AutoFL execution

Stage 1: Starting prompt for AutoFL

Listing 1. System Prompt for LLM

You are a debugging assistant. You will be presented with a failing test, and tools (functions) to

- ↪ access the source code of the system under test (SUT). Your task is to provide a step-by-
- ↪ step explanation of how the bug occurred, based on the failing test and the information
- ↪ you retrieved using tests about the SUT. You will be given N chances to interact with
- ↪ functions to gather relevant information. An example answer would look like follows.

<HANDCRAFTED ROOT CAUSE ANALYSIS EXAMPLE>

The starting (“system”) prompt provides general instructions to the LLM.

Stage 1: Bug-specific prompt

Listing 2. Example prompt from Defects4J Lang-48

```
1 The test `...EqualsBuilderTest::testBigDecimal()` failed. The test looks like:
2
3 ```java
4 381 : public void testBigDecimal() {
5 382 : BigDecimal o1 = new BigDecimal("2.0");
6 383 : BigDecimal o2 = new BigDecimal("2.00");
7 385 : assertTrue(new EqualsBuilder().append(o1, o2).isEqual()); // error occurred here
8 386 : }
9 ```
10
11 It failed with the following error message and call stack:
12 ```
13 junit.framework.AssertionFailedError
14   at ...EqualsBuilderTest::testBigDecimal(EqualsBuilderTest.java:385)
15 ```
16 Start by calling the `get_failing_tests_covered_classes` function.
```

The bug-specific (“user”) prompt provides bug details, along with a hint to start by calling a function.

Stage 1: Iterative calling of functions



Function Call Made: `get_failing_tests_covered_classes()`



Function success. Response length: 272 tokens. Click to Show:



The failing test `test2947660` is in `AbstractCategoryItemRendererTests`, so it likely involves classes `LineAndShapeRenderer` and `AbstractCategoryItemRenderer` in the `org.jfree.chart.renderer.category` package. It could also involve `DefaultCategoryDataset` and `CategoryPlot` in the `org.jfree.data.category` and `org.jfree.chart.plot` packages respectively, which are used within the test. Lastly, `LegendItemCollection` from the `org.jfree.chart` package is the object whose state leads to the test assertion failure.

Let's investigate the method in `AbstractCategoryItemRenderer` that is involved in the failing test. Call the `get_failing_tests_covered_methods_for_class` function for the class `AbstractCategoryItemRenderer`.

Function Call Made:

```
get_failing_tests_covered_methods_for_class(class_name=org.jfree.chart.renderer.category.AbstractCategoryItemRenderer)
```



Function success. Response length: 23 tokens. Click to Show:



The failing test covers three methods in the `AbstractCategoryItemRenderer` class: the constructor `.AbstractCategoryItemRenderer()`, the method `.setPlot(org.jfree.chart.plot.CategoryPlot)`, and the method `.getLegendItems()`.

Given the error message and the test code, the problem seems to arise when the `getLegendItems()` method is called, as this is where the assertion fails. Therefore, let's take a closer look at the `getLegendItems()` method.

Call the `get_code_snippet` function for `getLegendItems()`.

Function Call Made:

```
get_code_snippet(signature=org.jfree.chart.renderer.category.AbstractCategoryItemRenderer.getLegendItems())
```




Function success. Response length: 366 tokens. Click to Show:




The `getLegendItems()` method in `AbstractCategoryItemRenderer` follows these steps:

1. It creates an empty `LegendItemCollection` named `result`.
2. If the `plot` of `this` object is `null`, it immediately returns the empty `result`.
3. It gets the index of `this` object within the `plot`.

Stage 1: Functions to call (Coverage)

**Function Call Made:** `get_failing_tests_covered_classes()`

 **Function success.** Response length: 272 tokens. [Click to Show:](#)

```
{
  "org.jfree.chart.renderer.category": [
    "LineAndShapeRenderer",
    "AbstractCategoryItemRenderer"
  ],
  "org.jfree.chart.plot": [
    "DefaultDrawingSupplier",
    "CategoryPlot",
    "PlotOrientation",
    "Plot",
    "DatasetRenderingOrder"
  ],
  "org.jfree.chart.util": [
    "PaintList",
    "RectangleInsets",
    "AbstractObjectList",
    "UnitType",
    "ShapeList",
    "DefaultShadowGenerator",
    "SortOrder",
    "StrokeList",
    "ResourceBundleWrapper",
    "ObjectList",
    "BooleanList"
  ],
  "org.jfree.data.category": [
    "AbstractCategoryDataset",
    "DefaultCategoryDataset"
  ]
}
```

Function Call Made:
`get_failing_tests_covered_methods_for_class(class_name=org.jfree.chart.renderer.category.AbstractCategoryItemRenderer)`

 **Function success.** Response length: 23 tokens. [Click to Show:](#)

```
[
  ".getLegendItems()",
  ".setPlot(org.jfree.chart.plot.CategoryPlot)",
  ".AbstractCategoryItemRenderer()"
]
```

[Click to Fold](#)

Stage 1: Functions to call (Codebase)

Function Call Made:

```
get_code_snippet(signature=org.jfree.chart.renderer.category.AbstractCategoryItemRenderer.getLegendItems())
```



Function success. Response length: 366 tokens. Click to Show:

```
1790 :     public LegendItemCollection getLegendItems() {
1791 :         LegendItemCollection result = new LegendItemCollection();
1792 :         if (this.plot == null) {
1793 :             return result;
1794 :         }
1795 :         int index = this.plot.getIndexOf(this);
1796 :         CategoryDataset dataset = this.plot.getDataset(index);
1797 :         if (dataset != null) {
1798 :             return result;
1799 :         }
1800 :         int seriesCount = dataset.getRowCount();
1801 :         if (plot.getRowRenderingOrder().equals(SortOrder.ASCENDING)) {
1802 :             for (int i = 0; i < seriesCount; i++) {
1803 :                 if (isSeriesVisibleInLegend(i)) {
```

Function Call Made:

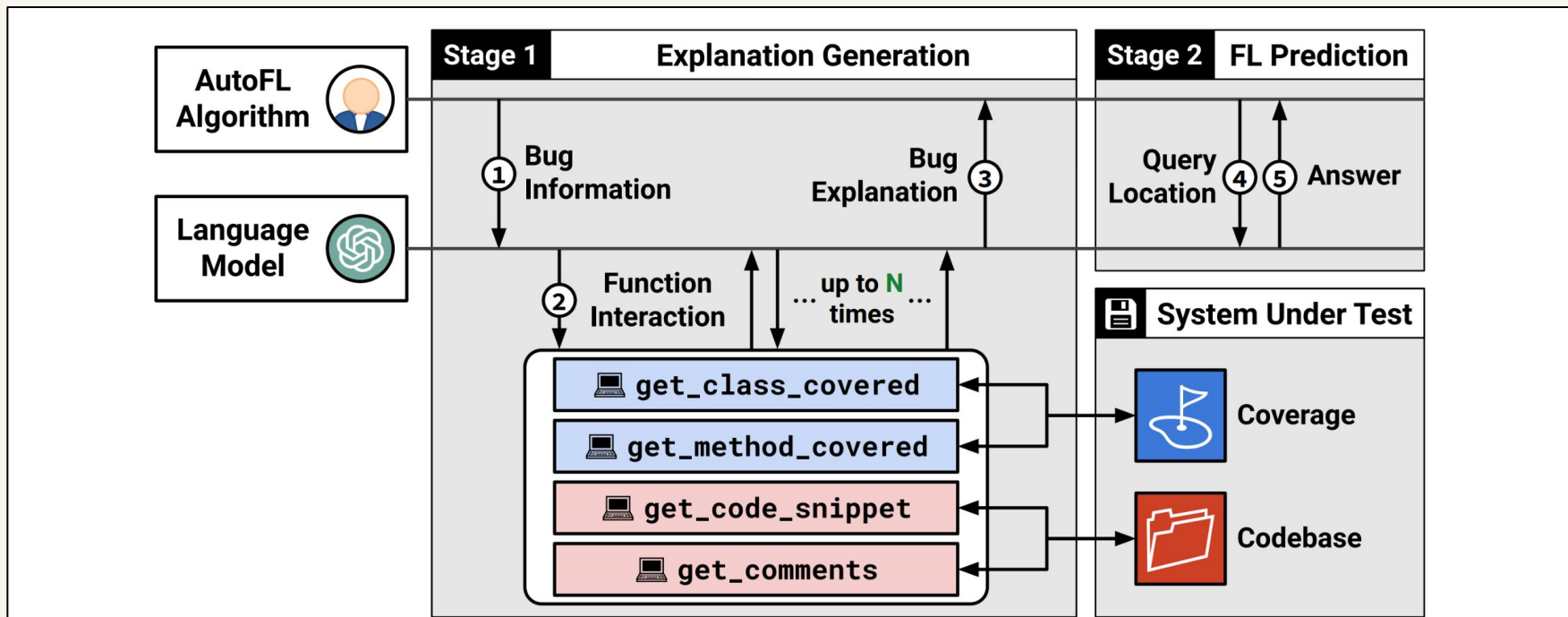
```
get_comments(signature=org.jfree.chart.renderer.category.AbstractCategoryItemRenderer.getLegendItems())
```



Function success. Response length: 75 tokens. Click to Show:

```
* Returns a (possibly empty) collection of legend items for the series
* that this renderer is responsible for drawing.
*
* @return The legend item collection (never null).
*
* @see #getLegendItem(int, int)
```

Stage 1: Termination



Stage 1 terminates when the LLM no longer calls a function, or when it reaches its function call limit (10).

Stage 2: Getting concrete FL results

Listing 3. Prompt to Request the Fault Location

Based on the available information, provide the signatures of the most likely culprit methods for
↳ the bug. Your answer will be processed automatically, so make sure to only answer with the
↳ accurate signatures of the most likely culprit (in ``ClassName.MethodName(ArgType1,
↳ ArgType2, ...)`` format), without commentary (one per line).

The bug-specific (“user”) prompt provides bug details, along with a hint to start by calling a function.

Getting final results - example



The JavaDoc for the `getLegendItems()` method in `AbstractCategoryItemRenderer` class states that this method returns a (possibly empty) collection of legend items for the series that this renderer is responsible for drawing.

This aligns with our understanding regarding this method's behavior, based on its code and its use in the test.

From the available evidence, it is reasonable to suggest that the fault causing the test failure lies within the `getLegendItems()` method in the `AbstractCategoryItemRenderer` class. The code should populate `result` with relevant `LegendItem` objects when a `dataset` exists; currently, it just returns the initially empty legend item collection whether or not a dataset exists.

The fix should involve modifying the logic in `getLegendItems()`, ensuring that legend items are correctly added to `result` when a `dataset` is present. `org.jfree.chart.renderer.category.AbstractCategoryItemRenderer.getLegendItems()`



Based on the available information, provide the signatures of the most likely culprit methods for the bug. Your answer will be processed automatically, so make sure to only answer with the accurate signatures of the most likely culprit (in `ClassName.MethodName(ArgType1, ArgType2, ...)` format), without commentary (one per line).



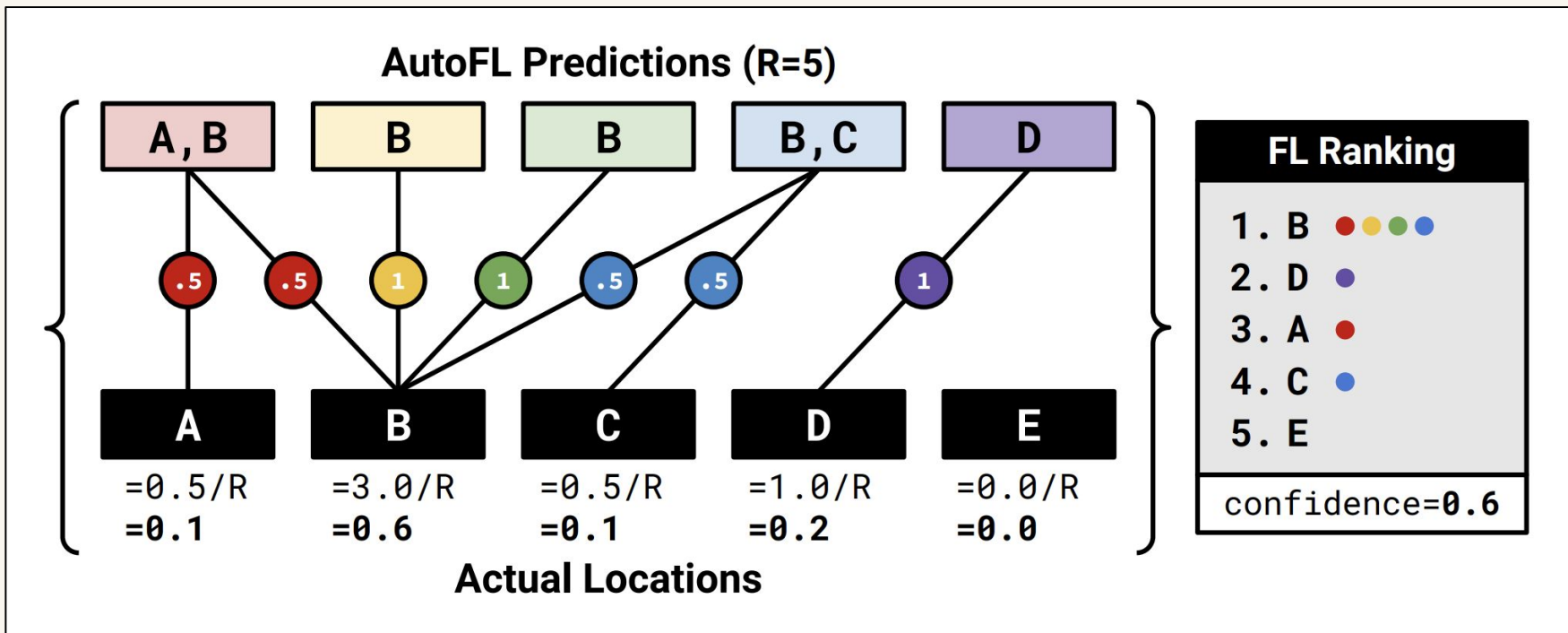
`org.jfree.chart.renderer.category.AbstractCategoryItemRenderer.getLegendItems()`



The final answer matched the true buggy method

`org.jfree.chart.renderer.category.AbstractCategoryItemRenderer.getLegendItems()`

Collating multiple LLM runs

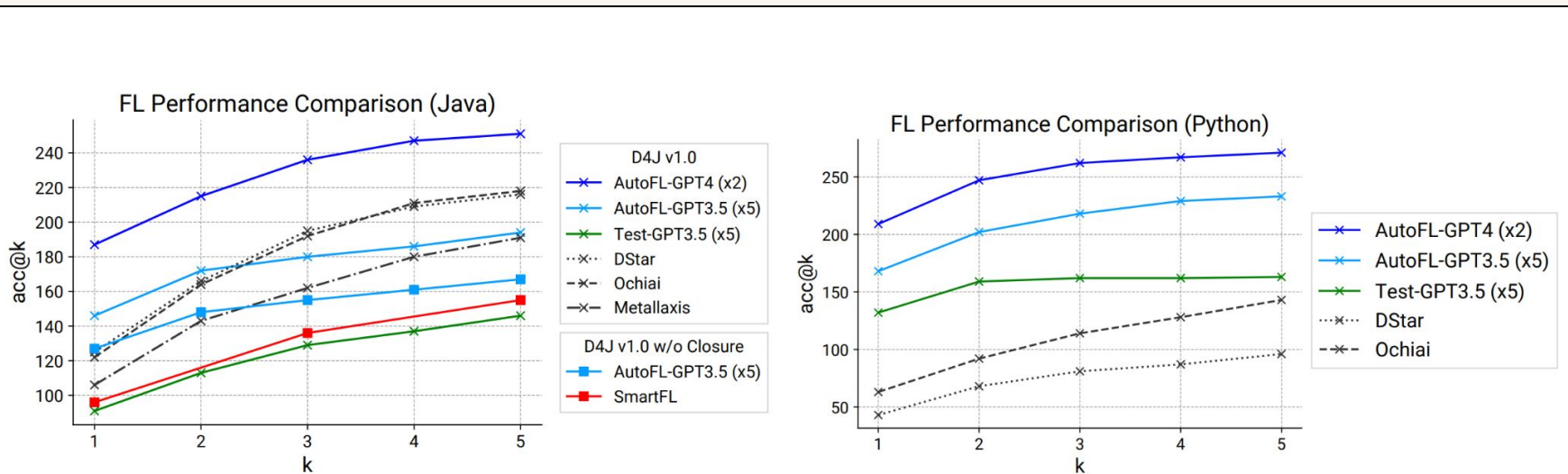


Multiple LLM answers are combined to yield a ranking, improving performance and provides a confidence estimate.



3. **Empirical Results**

RQ1: FL performance comparison with baselines



(a) FL evaluation on Defects4J

(b) FL evaluation on BugsInPy

AutoFL with GPT-4 outperforms all standalone techniques that we compared against.

RQ1: Performance gain from reruns

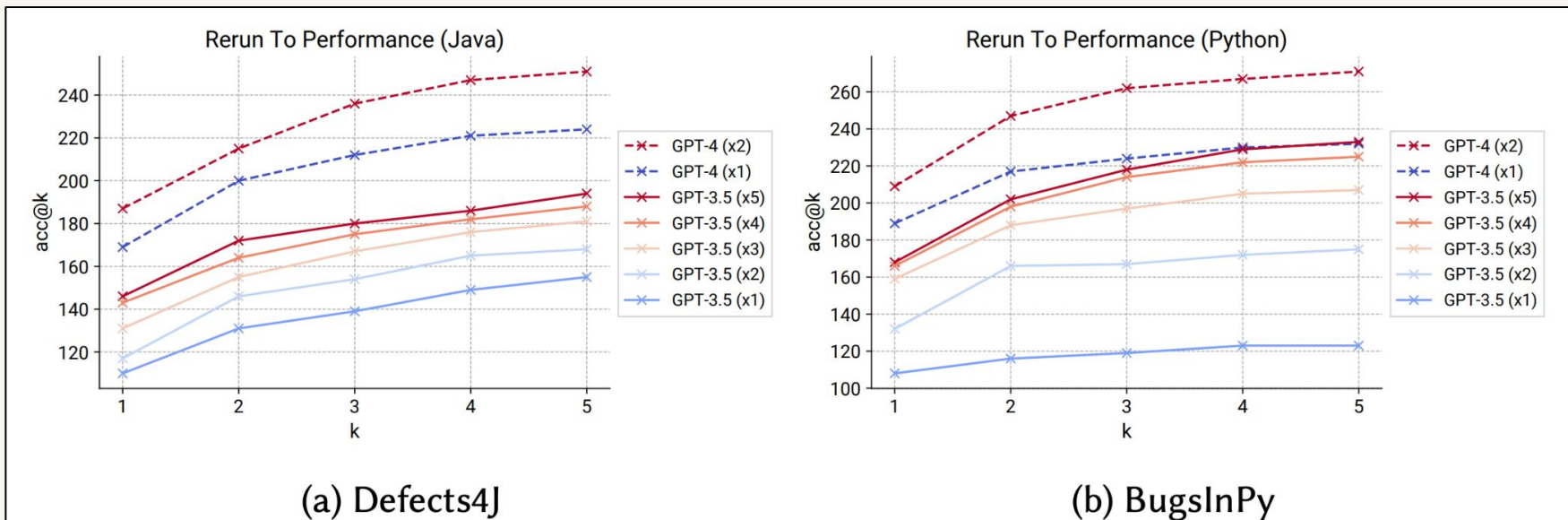
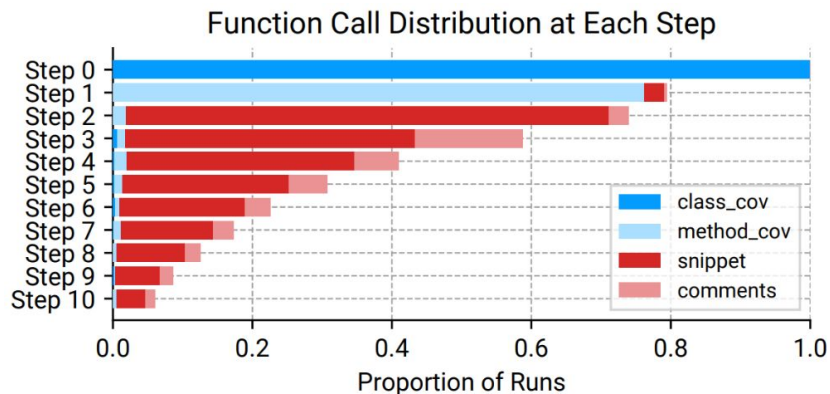


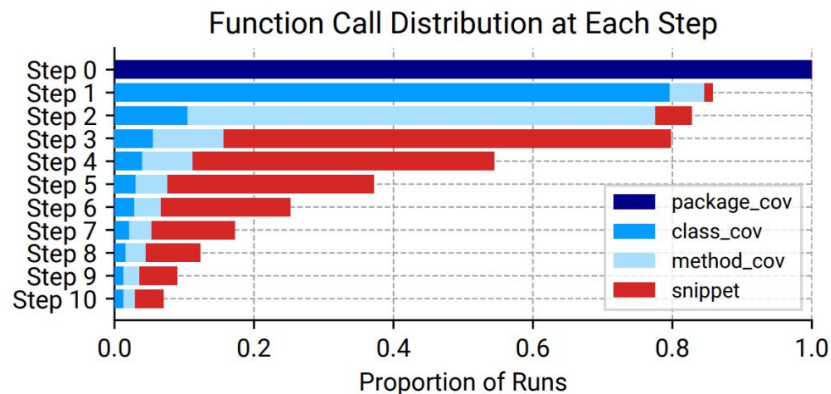
Fig. 4. Performance of AUTOFL as R increases, for Defects4J and BugsInPy.

Combining the result of multiple LLM runs improves the FL performance of AutoFL.

RQ1: AutoFL function call patterns



(a) Defects4J



(b) BugsInPy

Fig. 5. Function call distribution for AutoFL-GPT3.5.

In the process of fault localization, AutoFL tends calls functions according to the patterns given above.

RQ2: Confidence and FL performance

Table 4. Spearman’s rank correlation coefficients between AutoFL confidence and FL performance metrics in each benchmark (with ‘*’ denoting $p < 0.0001$). AutoFL is rerun 5 times using GPT-3.5.

Correlation with	Precision@1	Reciprocal Rank	Average Precision
Defects4J	+0.57*	+0.67*	+0.70*
BugsInPy	+0.52*	+0.50*	+0.49*

AutoFL confidence was strongly predictive of FL performance; thus confidence could help improve AutoFL precision.

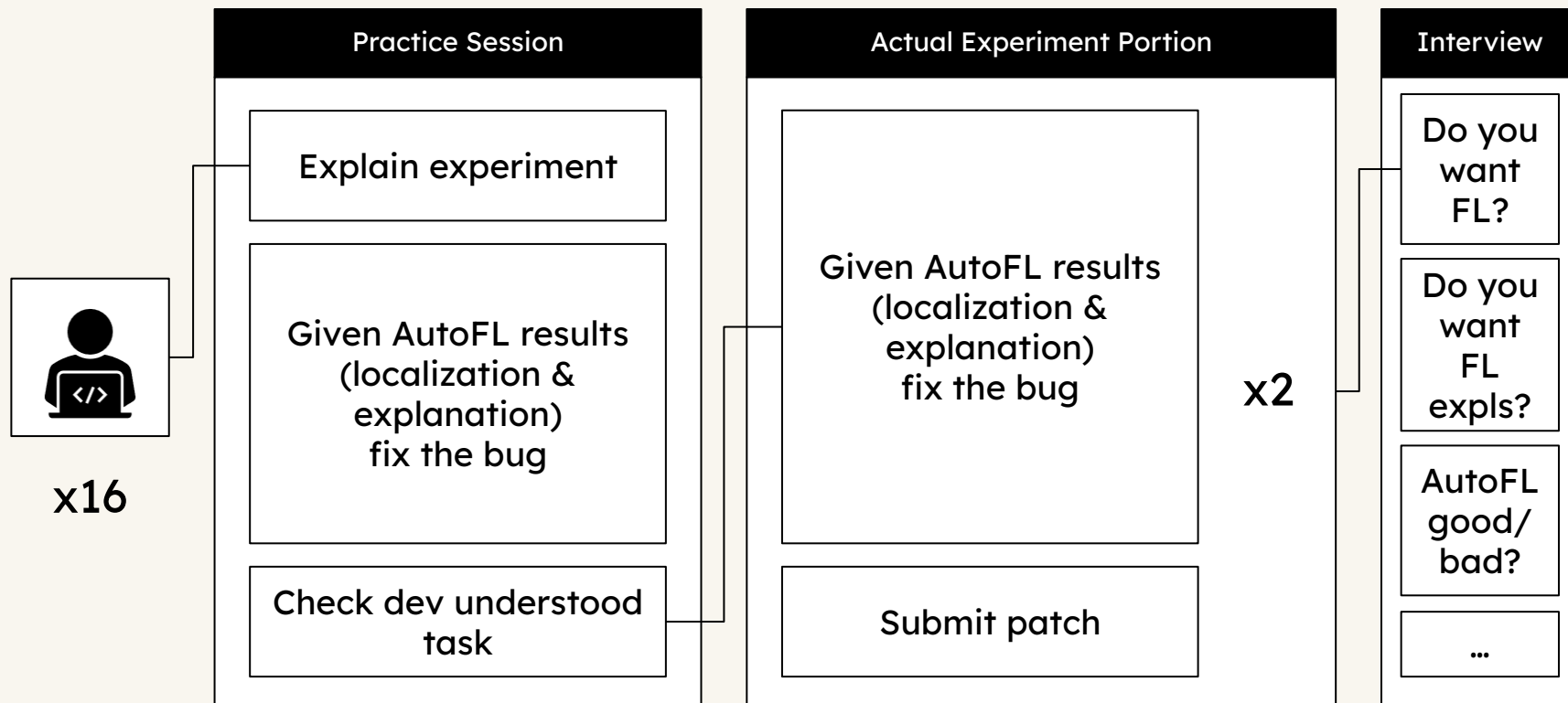
RQ3: Explanation characteristics

Table 5. Explanation rating results of AUTOFL-GPT3.5

Subset	Exists	Accurate	Imprecise	Concise	Useful	'Bland'	Total
Individual Explanations	83.7%	20.0%	26.3%	9.3%	8.0%	43.0%	300
$0.00 \leq \text{Confidence} < 0.25$	78.3%	10.0%	24.2%	3.3%	1.7%	46.7%	120
$0.25 \leq \text{Confidence} < 0.50$	87.5%	23.8%	28.8%	7.5%	11.3%	43.8%	80
$0.50 \leq \text{Confidence} < 0.75$	81.5%	26.2%	24.6%	16.9%	12.3%	36.9%	65
$0.75 \leq \text{Confidence} \leq 1.00$	97.1%	34.3%	31.4%	20.0%	14.3%	40.0%	35
Aggregated By Bug	100%	56.7%	66.7%	31.7%	23.3%	93.3%	60
$0.00 \leq \text{Confidence} < 0.25$	100%	37.5%	70.8%	16.7%	8.3%	95.8%	24
$0.25 \leq \text{Confidence} < 0.50$	100%	62.5%	68.8%	31.3%	31.3%	93.8%	16
$0.50 \leq \text{Confidence} < 0.75$	100%	69.2%	53.8%	46.2%	30.8%	84.6%	13
$0.75 \leq \text{Confidence} \leq 1.00$	100%	85.7%	71.4%	57.1%	42.9%	100%	7

Overall 20% of explanations were accurate; over five runs/bug, at least one explanation was accurate for 56.7% of bugs.

RQ4: Developer study - Setting



RQ4: Developer study - Results

**FL
wanted?**

13 developers said FL, even without explanations, would be helpful, particularly for unfamiliar code.

**FL expl.
wanted?**

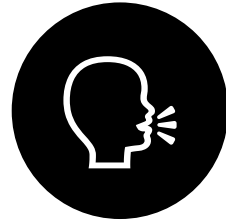
Four developers said explanations were necessary; eight said they were useful.

**AutoFL
good/bad**

Natural language description of error was helpful; inaccurate and redundant explanations were not.

**Ideal
Expl.**

Explanations with a clear format, along with dynamic values provided, presented with a few hypotheses.



4. **Discussion**

Future Directions



Applying AutoFL to software
on the industrial scale



Improving the interface for presenting
explanations generated by AutoFL



Automatically identifying accurate explanations
from a group of generated explanations

Predicting Execution Accuracy via Test Generation

Table 6. Spearman Correlation between explanation quality predictors and actual quality. Results with $p < 0.01$ are marked with *, and results significant with $p < 0.001$ are marked with **.

name	Test Score	APR Score	GPT _{useful}	Length
Accurate	+0.2358**	+0.1946*	+0.3759**	+0.3009**
'Wrong' (only imprecise)	+0.0408	-0.0643	+0.3266**	+0.3271**
Useful	+0.2635**	+0.1942*	+0.2371**	+0.1585
'Bland'	-0.2364**	-0.1105	-0.6026**	-0.5391**
FL Accurate	+0.2737**	+0.4923**	+0.1437	+0.1528

Execution results of executable artifacts were predictive of accurate bug explanations.

Conclusion



1

Fault localization is a task in which presenting **explanations** to developers is critical for usability.

2

We present **AutoFL**, which uses an LLM to autonomously inspect repository content, localize the fault, and explain the bug.

3

AutoFL shows **state-of-the-art** method-level FL performance, and **generated explanations** received positively by developers.

Contact us at sungmin.kang@kaist.ac.kr / gabin.an@kaist.ac.kr

Find our preprint with the QR code above, or by searching for

“A Quantitative and Qualitative Evaluation of LLM-Based Explainable Fault Localization”



*

Extra Slides