Exploring the Effectiveness of LLMs in Automated Logging Statement Generation: An Empirical Study

Yichen Li*, Yintong Huo*, Zhihan Jiang, Renyi Zhong, Pinjia He, Yuxin Su

Lionel C. Briand and Michael R. Lyu

The Chinese University of Hong Kong, Sun Yat-Sen University,

University of Ottawa and Lero Centre, University of Limerick

Read the paper!

LERO uOttawa 中山大學 SUN YAT-SEN UNIVERSITY ARISE Automated Reliable Intelligent Software Engineering 香港中文大學 The Chinese University of Hong Kong

Background & Motivation: Logs & Debugging

System logs are important for

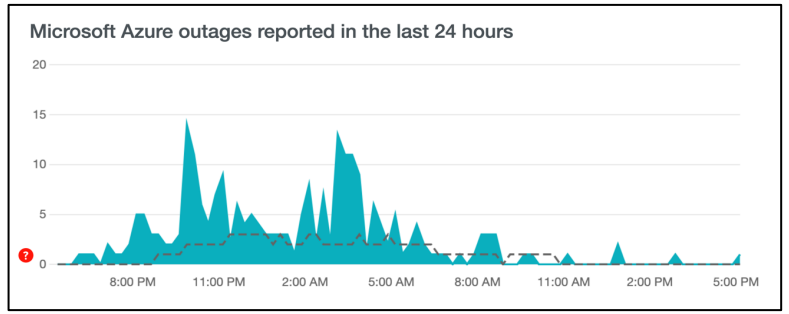
Diagnose runtime failures

Identify performance bottlenecks

Detect security issues

Market trends prediction

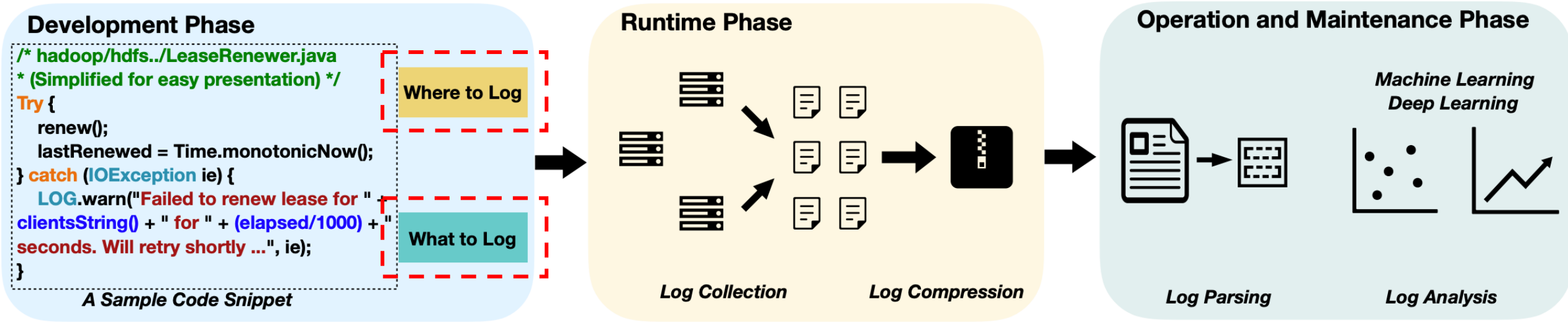
More than half of the studied bugs, root-cause diagnosis relies on log entries beyond the failure symptoms[1].



[1] How are distributed bugs diagnosed and fixed through system logs?

Background & Motivation: Logs & AIOps

Intelligent solutions for log analysis are now data-driven



Framework of Modern Intelligent Log Analytics

Background & Motivation

Logging is critical:

> *"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."*
>
> *-- Brian Kernighan*

```
if (current_value > max) {
  LOG.info("Update max value to: " + max);
  max = current_value;
}
```

Background & Motivation

Writing logs is hard:

- In the era of agile development, developers prioritize functional code.
- Dev

How to **automatically** generate **appropriate** logging statements

- It is hard to find the balance between informativeness and overhead

Motivation

Automated Logging Statement Generation

Writing better logging statements in the source code

Automated Coding Tools (Powered by LLM) [2]

How good are they?

[2] All tools and models are selected and evaluated by May, 2024.

The Overall Study Framework

RQs

1. How do different LLMs perform in deciding ingredients of logging statements generation?
2. How do LLMs compare to conventional logging models in logging ability?
3. How do the prompts for LLMs affect logging performance?
4. How do external factors influence the effectiveness in generating logging statements?
5. How do LLMs perform in logging unseen code?

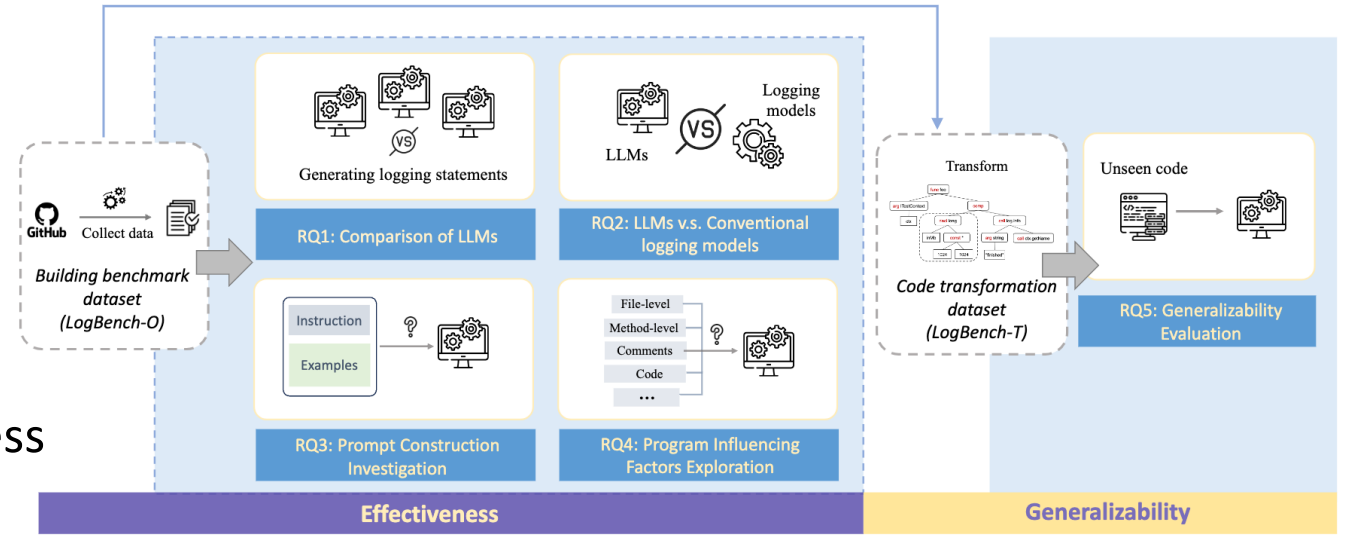


Fig. 2. The overall framework of this study involving five research questions.

Study Objects: Tools and Models

Both models and commercial products[2], parameters vary form 60M to over 175B, offering the service in different formats.

Also, we include the academic methods before the era of LLM:

TABLE III
CONVENTIONAL LOGGING APPROACH FOR SINGLE INGREDIENT RECOMMENDATIONS.

Ingredient	Model	Description	#Params	Venue	Year
Logging levels	DeepLV	DeepLV [11] leverages syntactic context and message features of the logging statements extracted from the source code to make suggestions on choosing log levels by feeding all the information into a deep learning model. We reimplement the model based on the replication package provided by the authors*.	0.2M	ICSE	2021
Logging Variables	WhichVar	WhichVar [13] applies an RNN-based neural network with a self-attention mechanism to learn the representation of program tokens, then predicts whether each token should be logged through a binary classifier. We reimplement the model based on its paper due to missing code artifacts*.	40M†	TSE	2021
Logging Text	LoGenText-Plus	LoGenText-Plus [44] generates the logging texts by neural machine translation models (NMT). It first extracts a syntactic template of the target logging text by code analysis, then feeds such templates and source code into Transformer-based NMT models. We reproduce the model based on the replication package provided by the authors.	22M	TOSEM	2023

TABLE II
STUDY SUBJECTS INVOLVED IN OUR EMPIRICAL STUDY.

Model	Access	Description	Pre-trained corpus (Data size)	#Params	Year
General-purpose LLMs					
Davinci	API	Davinci is derived from InstructGPT [31] is an "instruct" model meant to generate texts with clear instructions. We access the Text-davinci-003 model by calling the official API from OpenAI.	-	175B	2022
ChatGPT	API	ChatGPT is an enhanced version of GPT-3 models [32], with improved conversational abilities achieved through reinforcement learning from human feedback [33]. It forms the core of the ChatGPT system [34]. We access the GPT3.5-turbo model by calling the official API from OpenAI.	-	175B	2022
GPT-4o	API	GPT-4o [35] is the latest version of GPT series models, with significantly enhanced contextual understanding and generation capabilities, achieved through extensive fine-tuning and optimization. We access the GPT-4o model by calling the official API from OpenAI.	-	-	2024
Llama2	Model	Llama2 [36] is an open-sourced LLM trained on publicly available data and outperforms other open-source conversational models on most benchmarks. We deploy the Llama2-70B model provided by the authors	Publicly available sources (2T tokens)	70B	2023
Llama3.1	M			405B	2024
LANCE	M			60M	2022
Code-based LLMs					
InCoder	Model	InCoder [18] is a unified generative model trained on vast code benchmarks where code regions have been randomly masked. It thus can infill arbitrary code with bidirectional code context for challenging code-related tasks. We deploy the InCoder-6.7B model provided by the authors.	GitHub, GitLab, StackOverflow (159GB code, 57GB StackOverflow)	6.7B	2022
CodeGeeX	IDE Plugi	CodeGeeX [39] is an open-source code generation model, which has been trained on 23 programming languages and fine-tuned for code translation. We access the model via its plugin in VS Code.	GitHub code (158.7B tokens)	13B	2022
StarCoder	Model	StarCoder [40] has been trained on 1 trillion tokens from 80+ programming languages, and fine-tuned on another 35B Python tokens. It outperforms every open LLM for code at the time of release. We deploy the StarCoder-15.5B model provided by the authors.	The Stack (1T tokens)	15.5B	2023
CodeLlama	Model	CodeLlama [41] is a family of LLMs for code generation and infilling derived from Llama2. After they have been pretrained on 500B code tokens, they are all fine-tuned to handle long contexts. We deploy the CodeLlama-34B model provided by the authors.	Publicly available code (500B tokens)	34B	2023
TabNine	IDE Plugi	TabNine [42] is an AI code assistant that can suggest the following lines of code. It can automatically complete code lines, generate entire functions, and produce code snippets from natural languages. We access the model via its plugin in VS Code.	-	-	2022
Copilot	IDE Plugi	Copilot [21] is a widely-studied AI-powered code generation tool relying on the CodeX [43]. It can extend existing code by generating subsequent code trunks based on natural language descriptions. We access the model via its plugin in VS Code.	-	-	2021
CodeWhisperer	IDE Plugi	CodeWhisperer [23], developed by Amazon, serves as a coding companion for software developers. It can generate code snippets or full functions in real time based on comments written by developers. We access the model via its plugin in VS Code.	-	-	2022

Time flies so fast in the era of LLMs!

[2] All tools and models are selected and evaluated by May, 2024.

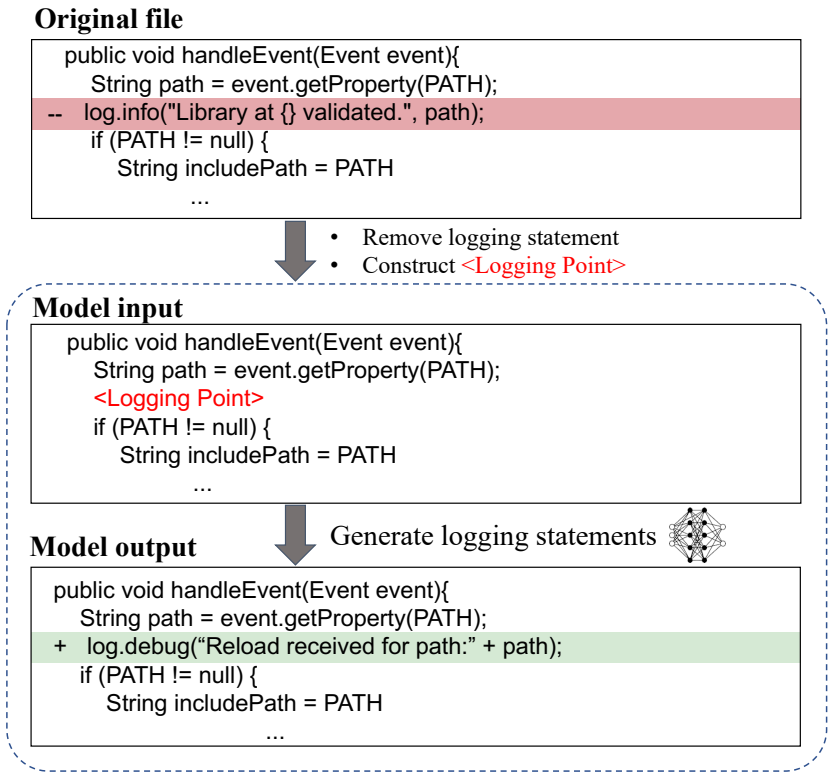
Study Objects: LogBench-O

We created the **first** benchmark dataset, LogBench-O, consists of **high-quality** and **well-maintained** Java projects with logging statement.

Logs exist for system maintenance.

How to define **high-quality** and **well-maintained**?

- Gaining more than 20 stars
- Receiving more than 100 commits.
- Engaging with at least 5 contributors.



Study Objects: LogBench-T

Sometimes, **models are memorizing results instead of inference [3].**

What's more: The models have most likely encountered data from LogBench-O during their training process.

So we created LogBench-T , the transformed version of LogBench-O.

- Lightweight and "daily"
- Readability-preserving

TABLE IV
OUR CODE TRANSFORMATION TOOLS WITH EIGHT CODE TRANSFORMERS, DESCRIPTIONS, AND ASSOCIATED EXAMPLES.

Transformer	Descriptions	Example
Condition-Dup	Add logically neutral elements (e.g., *&& True* or *|| False*)	if (exp0) → if (exp0 || false)
Condition-Swap	Swap the symmetrical elements of condition statements	if (var0 != null) → if (null != var0)
Local variable	Extract constant values and assign them to local variables	var0 = const0; → int var1 = const0; var0 = var1;
Assignment	Separate variable declaration and assignment	int var0 = var1; → int var0; var0 = var1;
Constant	Replace constant values with equivalent expressions	int var0 = const0 → int var0 = const0 + 0
For-While	Convert *for-loops* to equivalent *while-loops*	for (var0 = 0; var0 < var1; var0++) {} ↔
While-For	Convert *while-loops* to equivalent *for-loops*	var0 = 0; while (var0++ < var1) {}
Parenthesis	Add redundant parentheses to expression	var0 = arithExpr0 → var0 = (arithExpr0)

[3] Your Code Secret Belongs to Me: Neural Code Completion Tools Can Memorize Hard-Coded Credentials

Study Objects: LogBench-T

An Example from LogBench-O to LogBench-T

Original Code

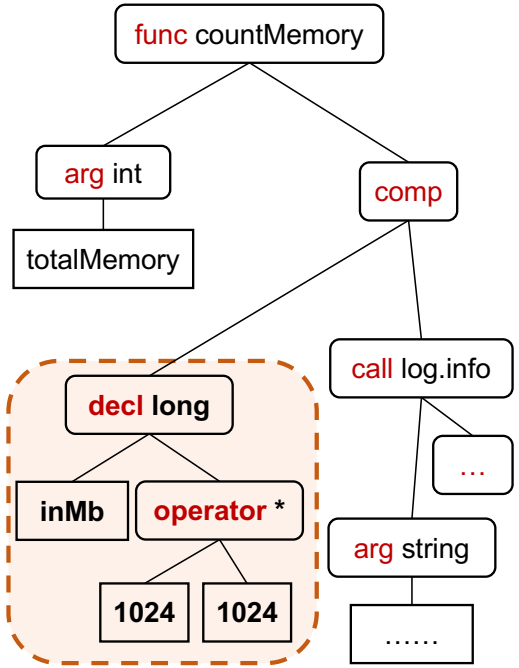
```
public void countMemory(ITestContext ctx, int totalMemory) {
    long inMb = 1024 * 1024;
    log.info("Total memory : " + totalMemory / inMb + " MB");
     …
```

Readability-preserving Code Transformation

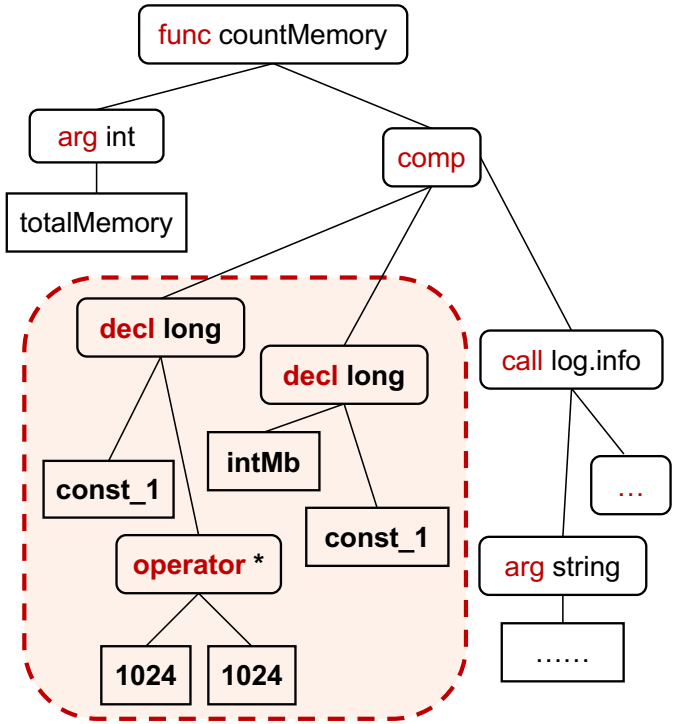
```
public void countMemory(ITestContext ctx, int totalMemory) {
  long const_1 = 1024 * 1024;
  long inMb = const_1;
  log.info("Total memory : " + totalMemory / inMb+ " MB");
   …
```

Readability-degrading Code Transformation

```
public void foo(ITestContext a, int b) {
    long c = 1024 * 1024;
    log.info("Total memory : " + b / c + " MB");
    …
```



Original AST



Transformed AST

RQ1: Comparison of LLMs

RQ1: How do different LLMs perform in deciding ingredients of logging statements generation?

Metrics:

- Logging Level: ACC, AOD (Ordinal (Log) Distance Score)
- Logging Variable: (Set level) Precision, Recall, F1
- Logging Text: BLEU, Rouge, Semantics similarity

RQ1: Results

RQ1: How do different LLMs perform in deciding ingredients of logging statements generation?

TABLE VI
THE EFFECTIVENESS OF LLMS IN PREDICTING LOGGING LEVELS AND LOGGING VARIABLES.

	Logging Levels		Logging Variables		
M					
Da					
Ch					
GI					
Ll					
Ll					
LA					
Code-based LLMs					
InCoder	0.608	0.800	0.712	0.655	0.682
CodeGeex	0.673	0.855	0.704	0.616	0.657
TabNine	0.734	0.880	0.729	0.670	0.698
Copilot	**0.743**	**0.882**	0.722	**0.703**	0.712
CodeWhisperer	0.741	0.881	**0.787**	0.668	**0.723**
CodeLlama	0.614	0.814	0.583	0.603	0.593
StarCoder	0.661	0.829	0.656	0.649	0.653

† Since LANCE decides logging point and logging statements simultaneously, we only consider its generated logging statements with correct locations.

Finding 1: While existing models correctly predict levels, there is **significant room for improvement** in producing variables and texts.

TABLE VII
THE EFFECTIVENESS OF LLMS IN PRODUCING LOGGING TEXTS.

	Logging Texts						
	BLEU-1	BLEU-2	BLEU-4	ROUGE-1	ROUGE-2	ROUGE-L	Semantics Similarity
	0.288						
T	0.291						
	0.384						
	0.235						
1	0.372						
†	0.306						
Code-based LLMs							
InCoder	0.369	0.288	0.203	0.390	0.204	0.383	0.640
CodeGeex	0.330	0.248	0.160	0.339	0.149	0.333	0.598
TabNine	0.406	0.329	0.242	0.421	0.241	0.415	0.669
Copilot	**0.417**	**0.338**	0.244	**0.435**	0.247	**0.428**	**0.703**
CodeWhisperer	0.415	0.338	**0.249**	0.430	**0.248**	0.425	0.672
CodeLlama	0.216	0.146	0.089	0.258	0.103	0.251	0.546
StarCoder	0.353	0.278	0.195	0.378	0.195	0.369	0.593

† Since LANCE decides logging point and logging statements simultaneously, we only consider its generated logging statements with correct locations.

Finding 2: LLMs may perform inconsistently on deciding different ingredients.

RQ2: Results

RQ2: How do LLMs compare to conventional logging models in logging ability?

We chose three representative pre-LLM methods for comparison.

- DeepLV for logging level
- WhichVar for logging var
- LogGenText-Plus for logging text

Finding 3: When directly applying LLMs without fine-tuning, they still yield better performance than conventional logging baselines.

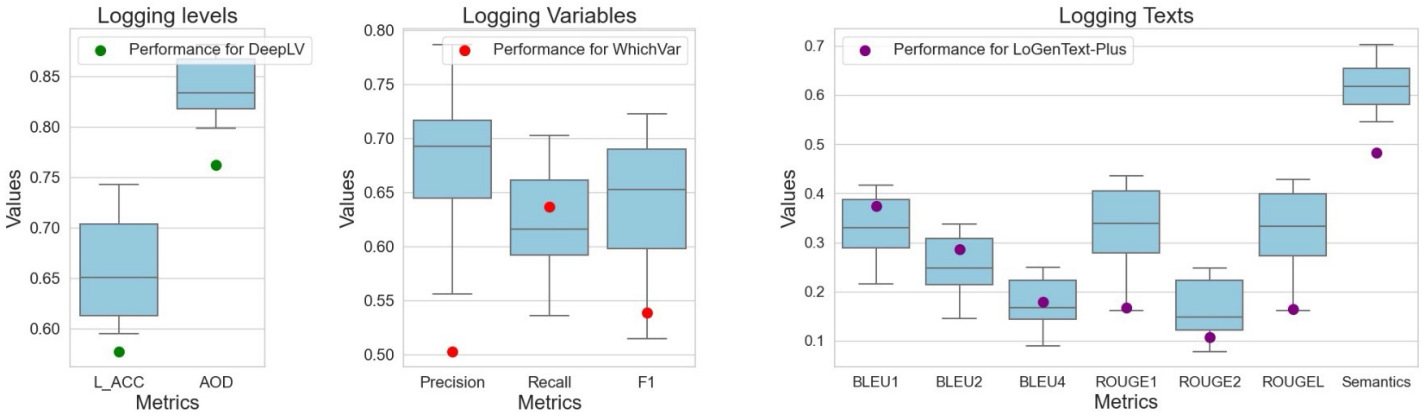


Fig. 4. The box plot (whisker plot) for traditional logging models and LLM-powered models. The box denotes the performance range of LLM-powered models, whereas the dot denotes the performance of traditional approaches. The box is drawn from the first quartile to the third quartile. A vertical line goes through the box at the median. The whiskers go from each quartile to the minimum or maximum.

RQ3: Results

RQ3: How do the prompts for LLMs affect logging performance?

- Impact of different instructions: Voted top 5 prompts:
  1. Your task is to generate the logging statement for the corresponding position.
  2. You are an expert in software DevOps; please help me write the informative logging statement.
  3. Complete the logging statement while taking the surrounding code into consideration.
  4. Your task is to write the corresponding logging statement. Note that you should keep consistent with current logging styles.
  5. Please help me write an appropriate logging statement below.

- Impact of different numbers of logging examples:
  - From 1 to 9

RQ3: Results

RQ3: How do the prompts for LLMs affect logging performance?

- Impact of different instructions: Voted top 5 prompts:

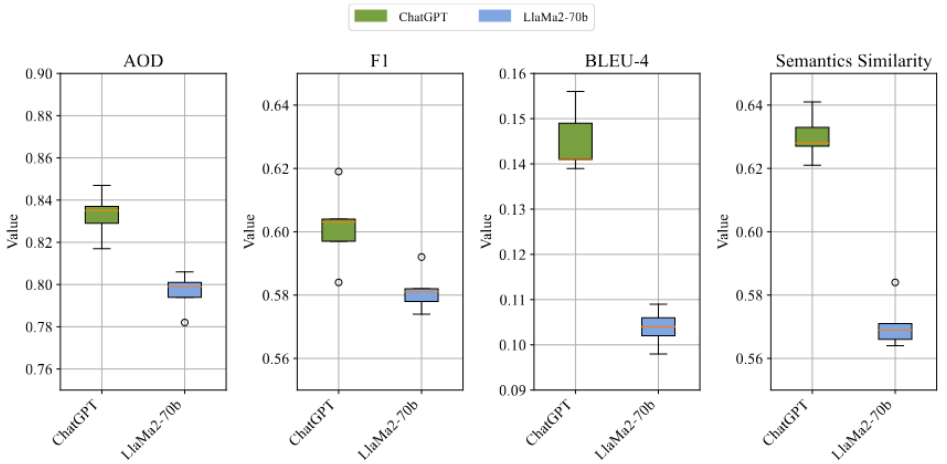


Fig. 7. The selected metrics of LLMs' logging performance with different instructions.

Finding 4: Although instructions influence LLMs to various extents, there is cohesiveness in the relative ranking of LLMs with the same instructions.

RQ3: Results

RQ3: How do the prompts for LLMs affect logging performance?

- Impact of different numbers of logging examples:

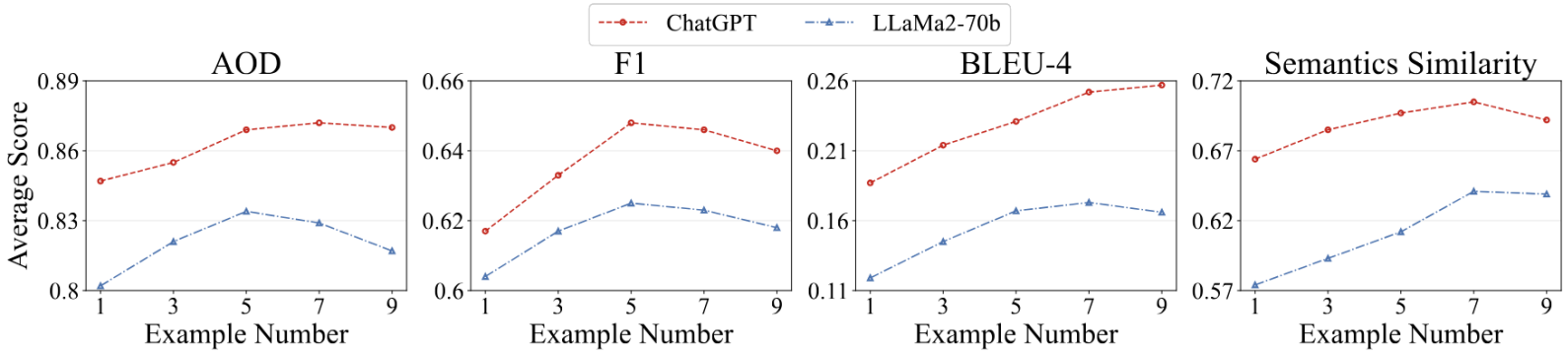


Fig. 8. The selected metrics of LLMs' logging performance with different numbers of examples.

Finding 5: More demonstration examples in the prompt do not always improve performance. It is recommended to use 5-7 examples in the demonstration to achieve optimal results.

RQ4: Results

RQ4: How do external factors influence the effectiveness in generating logging statements?

- With comment v.s. without comment

TABLE VIII
THE RESULTS OF LOGGING STATEMENT GENERATION WITHOUT COMMENTS.

Model	Logging Levels	Logging Variables	Logging Texts		
	AOD	F1	BLEU-4	ROUGE-L	Semantics Similarity
Davinci	0.834 (0.0%-)	0.587 (3.1%↓)	0.133 (3.6%↓)	0.283 (1.0%↓)	0.608 (1.5%↓)
ChatGPT	0.833 (0.2%↓)	0.592 (2.0%↓)	0.149 (0.0%-)	0.294 (1.3%↓)	0.614 (3.0%↓)
GPT-4o	0.861 (0.8%↓)	0.664 (2.2%↓)	0.168 (3.4%↓)	0.397 (2.7%↓)	0.678 (2.3%↓)
Llama2	0.789 (1.3%↓)	0.574 (1.2%↓)	0.099 (2.9%↓)	0.255 (2.3%↓)	0.544 (4.4%↓)
Llama3.1	0.811 (0.7%↓)	0.652 (2.5%↓)	0.161 (4.2%↓)	0.358 (3.8%↓)	0.671 (2.5%↓)
InCoder	0.789 (1.4%↓)	0.674 (1.2%↓)	0.201 (1.0%↓)	0.377 (9.2%↓)	0.622 (2.8%↓)
CodeGeex	0.848 (0.8%↓)	0.617 (6.1%↓)	0.149 (6.9%↓)	0.306 (8.1%↓)	0.578 (3.3%↓)
TabNine	0.876 (0.5%↓)	0.690 (1.1%↑)	0.239 (1.2%↓)	0.412 (0.7%↓)	0.655 (2.1%↓)
Copilot	**0.878** (0.5%↓)	0.696 (2.2%↓)	0.241 (1.2%↓)	**0.419** (2.1%↓)	**0.689** (2.0%↓)
CodeWhisperer	0.877 (0.7%↓)	**0.718** (0.7%↓)	**0.244** (2.0%↓)	0.418 (1.6%↓)	0.661 (1.6%↓)
CodeLlama	0.804 (1.2%↓)	0.581 (2.0%↓)	0.087 (2.2%↓)	0.247 (1.6%↓)	0.544 (0.3%↓)
StarCoder	0.823 (0.7%↓)	0.647 (0.9%↓)	0.193 (1.0%↓)	0.369 (2.4%↓)	0.591 (**0.3%**↓)
Avg.Δ	- (0.7%↓)	- (2.1%↓)	- (2.5%↓)	- (3.1%↓)	- (2.2%↓)

RQ4: Results

Case

```
public void setPhysicalName(String physicalName) {
        ...
    // Parse off the sequenceId off the end.
    // this can fail if the temp destination is
    // generated by another JMS system via the JMS<->JMS Bridge
        ...
          try {
             sequenceId = Integer.parseInt(seqStr);
          } catch (NumberFormatException e) {
+            LOG.debug("Did not parse sequence Id from " + physicalName);
                ...
```

Comments

Without commets (wo/ cmt)

```
LOG.warn("Invalid sequence number: " + seqStr);
```

With commets

```
LOG.debug("Failed to parse sequence Id from " + physicalName);
```

Fig. 9. A logging statement generation case using code comments.

Finding 6: Ignoring code comments impedes LLMs in generating logging statements, resulting in an average 2.2% semantics decrease when recommending logging texts.

RQ4: Results

RQ4: How do external factors influence the effectiveness in generating logging statements?

- Programming contexts: method v.s. file.

TABLE IX
THE RESULTS OF LOGGING STATEMENT GENERATION WITH FILE-LEVEL CONTEXTS.

Model	Logging Levels	Logging Variables	Logging Texts		
	AOD	F1	BLEU-4	ROUGE-L	Semantics Similarity
Davinci	0.854 (2.6%↑)	0.638 (5.3%↑)	0.156 (13.0%↑)	0.318 (11.2%↑)	0.635 (2.9%↑)
ChatGPT	0.858 (2.8%↑)	0.650 (7.6%↑)	0.253 (51.5%↑)	0.389 (30.5%↑)	0.704 (11.2%↑)
GPT-4o	0.905 (4.3%↑)	0.724 (6.6%↑)	0.318 (82.8%↑)	0.552 (35.3%↑)	0.809 (16.6%↑)
Llama2	0.832 (4.1%↑)	0.617 (6.2%↑)	0.149 (46.1%↑)	0.392 (50.2%↑)	0.669 (17.6%↑)
Llama3.1	0.848 (3.8%↑)	0.714 (6.7%↑)	0.301 (79.1%↑)	0.493 (32.5%↑)	0.794 (15.4%↑)
InCoder	0.815 (1.9%↑)	0.745 (9.2%↑)	0.307 (51.2%↑)	0.521 (35.3%↑)	0.734 (11.7%↑)
CodeGeex	0.869 (1.6%↑)	0.696 (5.9%↑)	0.241 (50.6%↓)	0.395 (18.6%↑)	0.644 (7.7%↑)
TabNine	0.912 (3.6%↑)	0.767 (9.9%↑)	0.375 (55.0%↑)	0.530 (27.7%↑)	0.783 (17.0%↑)
Copilot	**0.916** (3.9%↑)	0.742 (4.2%↑)	0.346 (41.8%↑)	0.522 (22.0%↑)	**0.816** (16.1%↑)
CodeWhisperer	0.913 (3.6%↑)	**0.792** (9.6%↑)	**0.401** (61.0%↑)	**0.559** (31.5%↑)	0.811 (20.7%↑)
CodeLlama	0.817 (0.4%↑)	0.607 (2.4%↑)	0.144 (61.8%↑)	0.378 (50.6%↑)	0.642 (17.6%↑)
StarCoder	0.847 (2.2%↑)	0.714 (9.3%↑)	0.314 (61.0%↑)	0.517 (40.1%↑)	0.679 (14.5%↑)
Avg.Δ	2.9%↑	6.8%↑	54.6%↑	32.1%↑	14.1%↑

RQ4: Results

Case

```
public abstract class BrokerPluginSupport extends MutableBrokerFilter
        ...
   public void start() throws Exception {                          Method1
      super.start();
      LOG.info("Broker Plugin {} started", getClass().getName());}

   public void stop() throws Exception {                           Target method
      super.stop();
+     LOG.info("Broker Plugin {} stopped", getClass().getName());}
        ...
```

Method-level input

```
LOG.info("Stopped");
```

File-level input (w/ file)

```
LOG.info("Broker Plugin {} stopped", getClass().getName());
```

Fig. 10. A logging statement generation case using different programming contexts.

Finding 7: File-level programming contexts leads to a greater improvement in logging practice by providing methods, variable definitions and intra-project logging styles.

RQ5: Results

RQ5: How do LLMs perform in logging unseen code?

TABLE X
THE GENERALIZATION ABILITY OF LLMS IN PRODUCING LOGGING STATEMENTS FOR UNSEEN CODE.

Model	Levels		Variables		Texts						Average
	AOD	Δ	F1	Δ	BLEU-4	Δ	ROUGE-L	Δ	Semantics	Δ	Avg. Δ
General-purpose LLMs											
Davinci	0.820	1.7%↓	0.523	13.7%↓	0.116	15.9%↓	0.234	20.7%↓	0.533	13.6%↓	13.1%↓
ChatGPT	0.830	0.6%↓	0.532	11.9%↓	0.118	20.8%↓	0.240	19.5%↓	0.541	14.5%↓	13.5%↓
GPT-4o	0.852	1.8%↓	0.624	8.1%↓	0.152	12.6%↓	0.341	16.4%↓	0.615	11.4%↓	10.1%↓
Llama2	0.788	1.4%↓	0.568	2.2%↓	0.094	7.8%↓	0.213	18.4%↓	0.513	9.8%↓	7.9%↓
Llama3.1	0.806	1.3%↓	0.645	3.6%↓	0.147	12.5%↓	0.310	17.8%↓	0.625	9.2%↓	8.9%↓
Logging-specific LLMs											
LANCE	0.817	0.6%↓	0.475	7.5%↓	0.153	8.4%↓	0.144	**11.1%↓**	0.301	13.3%↓	**8.2%↓**
Code-based LLMs											
InCoder	0.778	2.8%↓	0.587	13.9%↓	0.175	13.8%↓	0.316	17.5%↓	0.584	8.8%↓	11.4%↓
CodeGeex	0.850	0.6%↓	0.534	18.7%↓	0.115	28.1%↓	0.253	25.4%↓	0.549	8.2%↓	16.2%↓
TabNine	0.869	1.3%↓	0.596	14.6%↓	0.202	16.5%↓	0.342	18.8%↓	0.608	9.1%↓	12.1%↓
Copilot	**0.881**	**0.1%↓**	0.610	14.3%↓	**0.234**	**4.1%↓**	**0.377**	13.3%↓	**0.641**	8.8%↓	8.2%↓
CodeWhisperer	0.871	1.1%↓	**0.629**	13.0%↓	0.219	12.0%↓	0.362	14.6%↓	0.612	8.9%↓	9.9%↓
CodeLlama	0.801	1.6%↓	0.574	**3.2%↓**	0.078	12.6%↓	0.211	15.9%↓	0.482	11.7%↓	9.0%↓
StarCoder	0.811	2.2%↓	0.619	5.2%↓	0.175	10.3%↓	0.309	16.3%↓	0.546	**7.9%↓**	8.4%↓
Avg. Δ	-	1.3%↓	-	10.0%↓	-	13.5%↓	-	17.4%↓		10.4%↓	10.8%↓

RQ5: Results

Case

Original code

```
public void countMemory(ITestContext ctx, int totalMemory) {
  long inMb = 1024 * 1024;
+ log.info("Total memory : " + totalMemory / inMb + " MB");
  MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();
  MemoryUsage heapMemoryUsage = memoryMXBean.getHeapMemoryUsage();
  ...
```

Code after transformation

```
public void countMemory(ITestContext ctx, int totalMemory) {
  long const_1 = 1024 * 1024;
  long inMb = const_1;
+ log.info("Total memory : " + totalMemory / inMb+ " MB");
  MemoryMXBean memoryMXBean = ManagementFactory.getMemoryMXBean();
  MemoryUsage heapMemoryUsage = memoryMXBean.getHeapMemoryUsage();
  ...
```

Original code	
CodeWhisperer	log.info("Memory usage: " + totalMemory / inMb + " MB");
ChatGPT	log.info("Current heap memory usage: " + heapMemoryUsage.getUsed() / inMb + " MB");
Incoder	log.debug("Memory used: " + totalMemory / inMb + "MB");
Tansformed code	
CodeWhisperer	log.info("Total Memory: " + totalMemory + " MB");
ChatGPT	log.info("Starting memory count...");
Incoder	log.warn("Memory usage: " + heapMemoryUsage);

Fig. 11. A case of code transformation and its corresponding predicted logging statement from multiple models.

Finding 8: LLMs' performance drops significantly for unseen code, highlighting the need to improve the generalization capabilities of these models.

Takeaways & TL;DR

Lessons Learned:

1. **A paradigm shift in logging automation.**
2. **Context is everything for logging quality (even for general coding).**
3. **The generalization gap is real.**
4. **The new frontier is “how to express”.**

Future Directions

Future Directions:

1. **Develop Unified & Semantic-Aware Evaluation Metrics.**
2. **Intelligent Logging Context Retrieval.**
3. **Whether to Log.**

Q&A

Thank you sooooo much for listening!

Read the paper!

Check the data and code!