COCA: Generative Root Cause Analysis for Distributed Systems with Code Knowledge

Yichen Li, Yulun Wu, Jinyang Liu, Zhihan Jiang, Zhuangbin Chen, Guangba Yu

Michael R. Lyu

The Chinese University of Hong Kong

Read the paper!

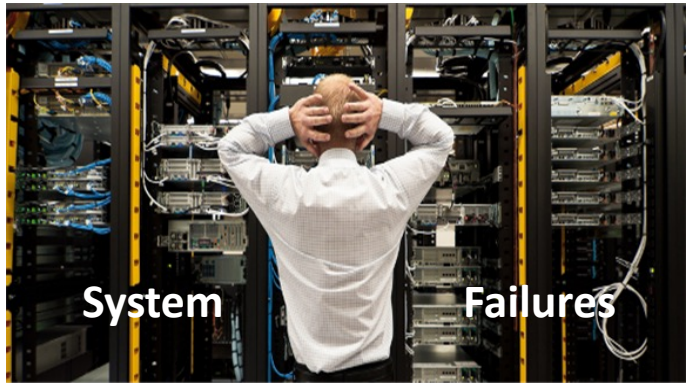
ARISE Automated Reliable Intelligent Software Engineering

香港中文大學 The Chinese University of Hong Kong

Background: Reliability

Cloud systems suffer from unplanned interruptions and failures.

Hard to conduct Root Cause Analysis (RCA) to these failures.



YOUTUBE · Published December 14, 2020 9:43am EST

Google lost $1.7M in ad revenue during YouTube outage, expert says

YouTu

Amazon outage frustrates sellers and shoppers

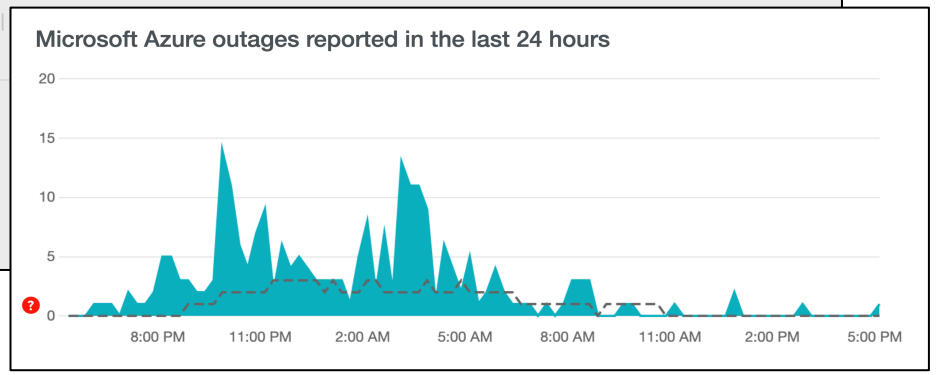
Gretchen Salois |

News > Business

Amazon 'missed out on $34m in sales during internet outage'

The e-commerce giant generates $9,615 in sales per second – but not when it's website is down

Ben Chapman • Tuesday 08 June 2021 16:54 • 1 Comments



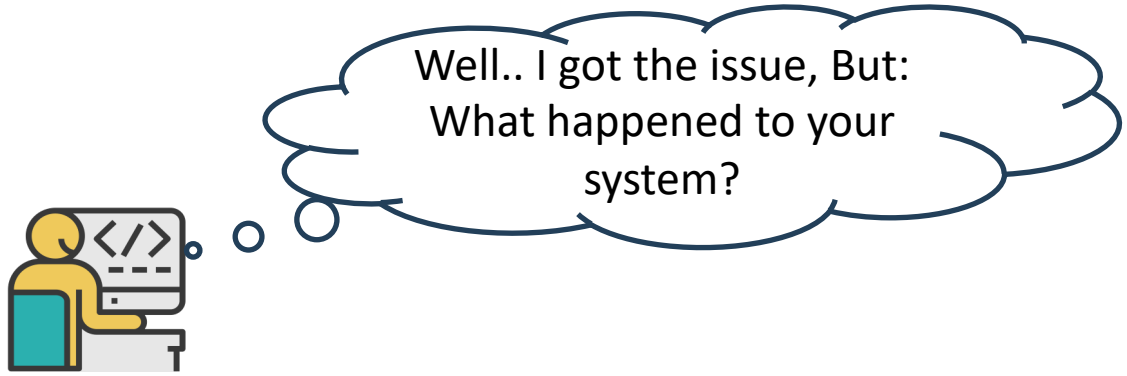
Background: How to Define “RCA”

RCA is some kind of general term.

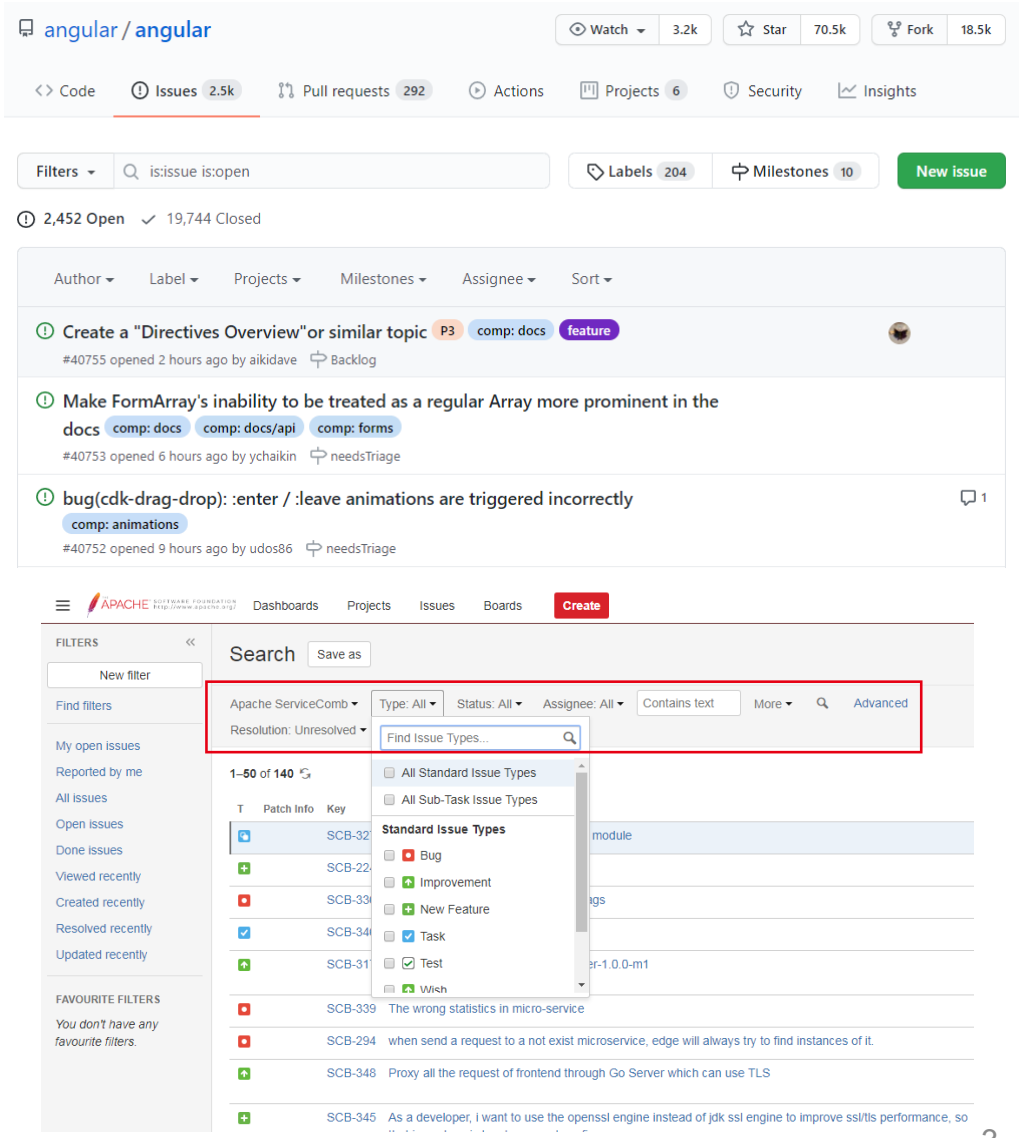
Some failures are code-related while others are not

Maintainers want to get

1. Operational Clues
2. Code Fixing Scope and Corresponding Logic



Poor Maintainer



Background

What we got from an issue report

We got

- Textual data
  - Title
  - Issue description
- Issue metadata
- Logs
- Stack traces

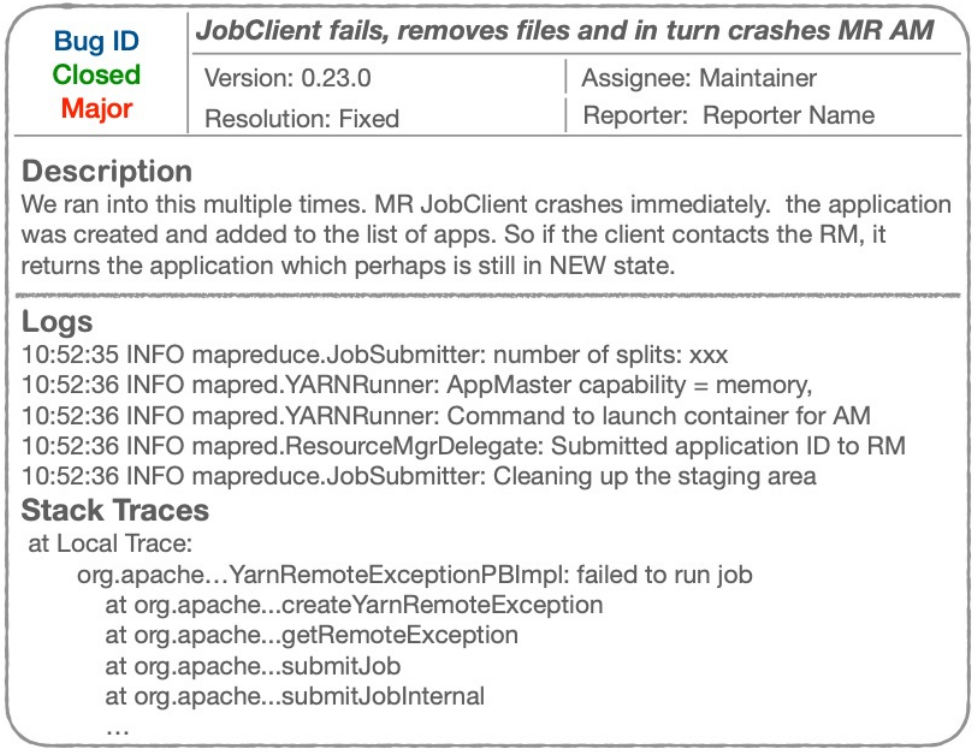


Fig. 1: Example of a issue report.

Background: A Motivating Example

Involving Code Knowledge for Distributed System Diagnosis: Intuitive but challenging

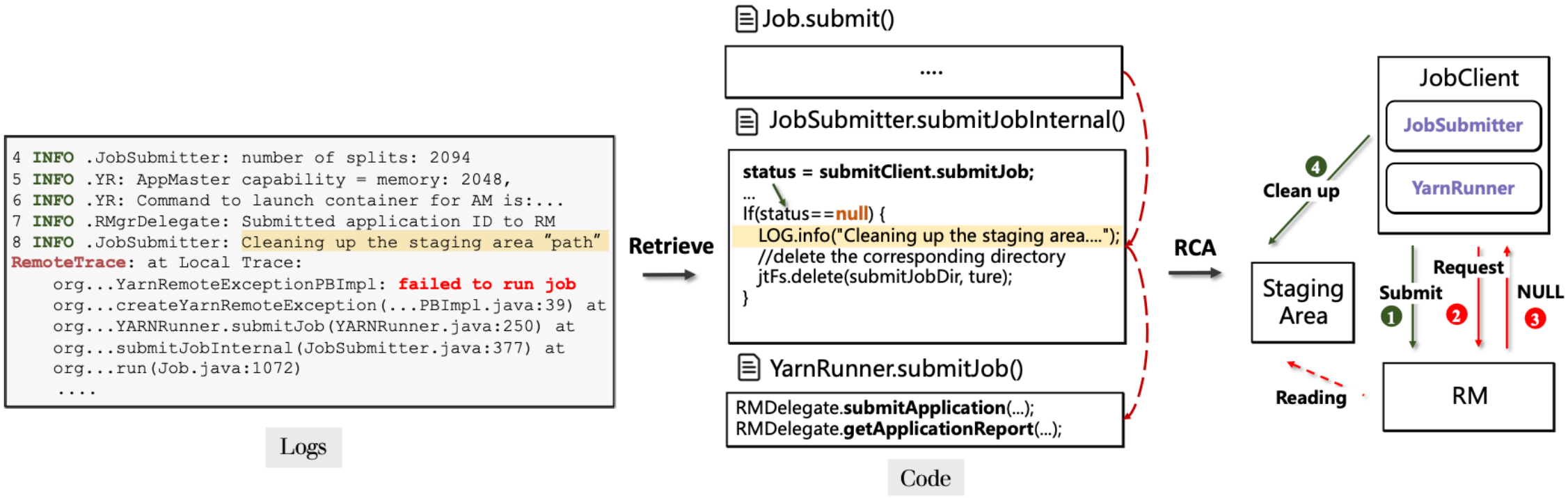
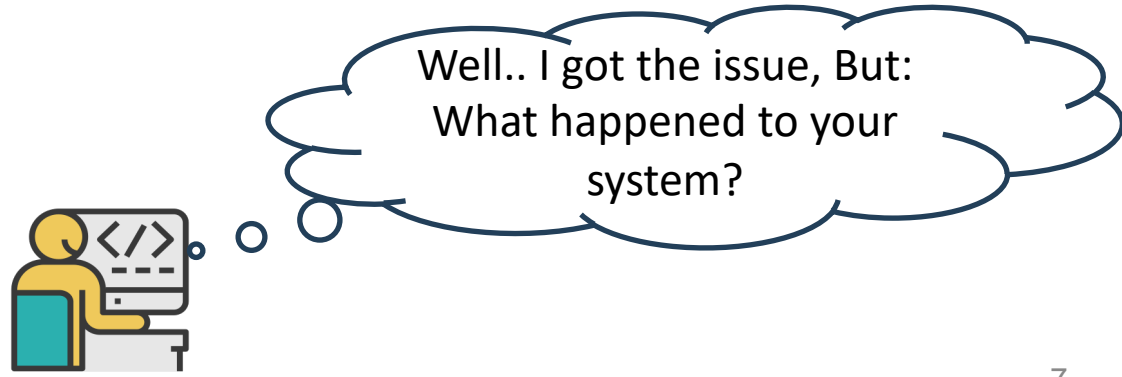


Fig. 2: A motivating example: MapReduce-2953.

Challenges

Challenges

- Challenge 1: Sourcing Log Messages
- Challenge 2: Reconstructing Execution Paths
- Challenge 3: Profiling Failure-Related Code Snippets.



Challenge 1

Challenges

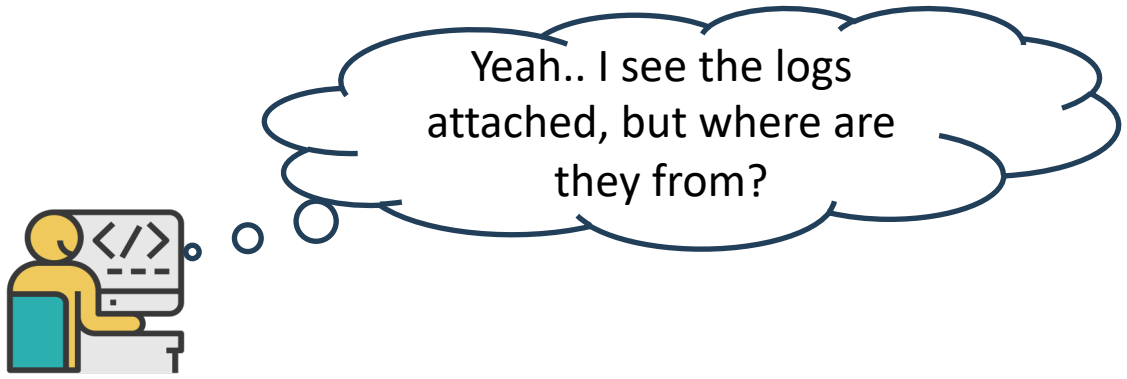
- Challenge 1: Sourcing Log Messages
- Challenge 2: Reconstructing Execution Paths
- Challenge 3: Profiling Failure-Related Code Snippet

Logging Statements

```
# Logging statements from Spark (spark/storage/BlockManager.scala)
logError(s"Failed to report $blockId to master; giving up.")
logDebug(s"Putting block ${blockId} with replication took $usedTimeMs")
logInfo(s"Writing block $blockId to disk")
```

Log Message

```
17/08/22 15:50:46 ERROR BlockManager Failed to report rdd_5_1 to master; giving up.
17/08/22 15:50:55 DEBUG BlockManager Putting block rdd_0_1 with replication took 0
17/08/22 15:51:02 INFO BlockManager Writing block rdd_1_3 to disk
17/08/22 15:51:24 DEBUG BlockManager Putting block rdd_2_2 with replication took 0
17/08/22 15:52:36 ERROR BlockManager Failed to report rdd_3_3 to master; giving up.
```



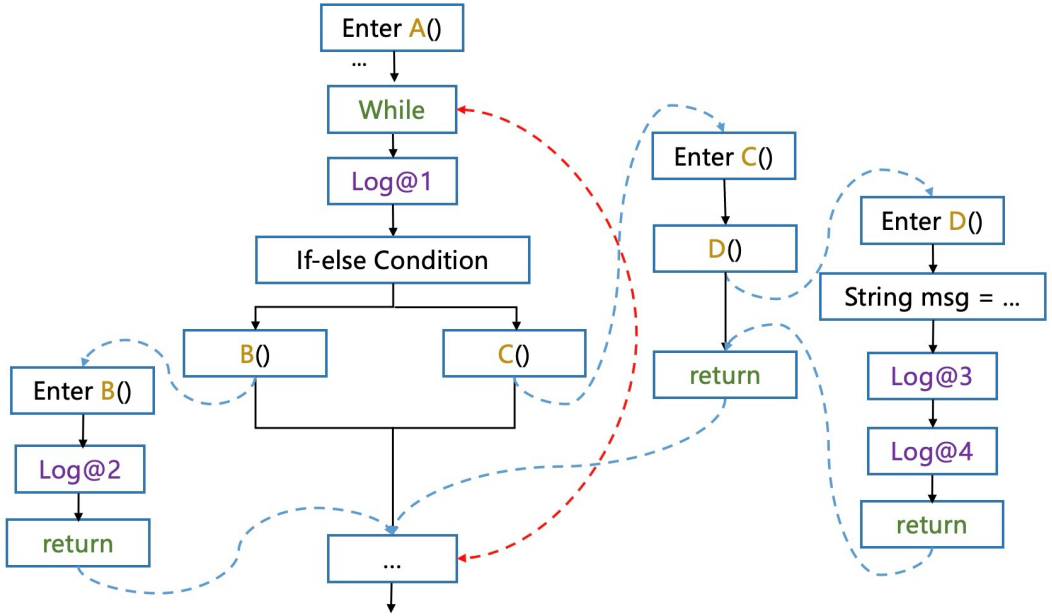
Poor Maintainer

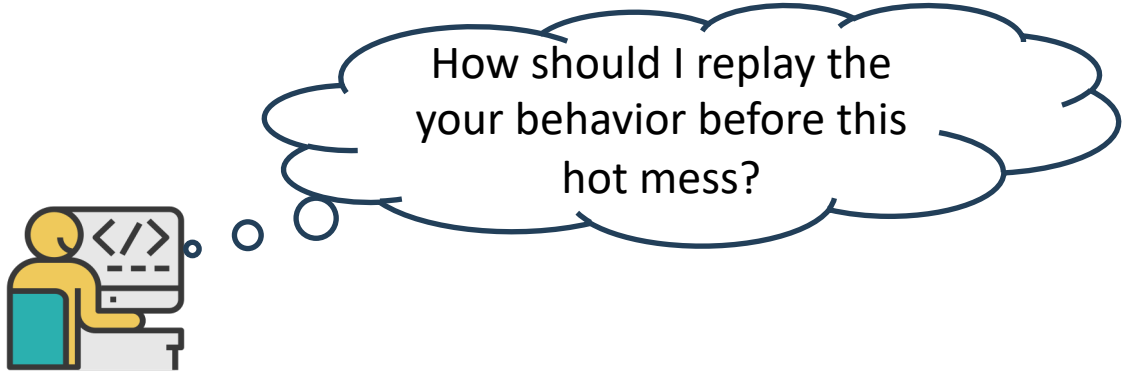
Source line references are occasionally absent from log entries.

Challenge 2

Challenges

- Challenge 1: Sourcing Log Messages
- Challenge 2: Reconstructing Execution Paths
- Challenge 3: Profiling Failure-Related Code Snippets





Poor Maintainer

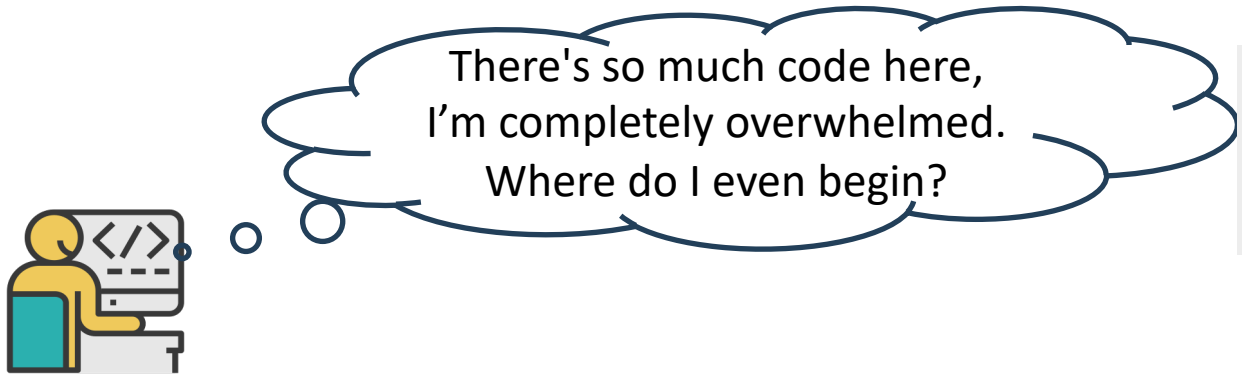
Logs are sparse and reconstructing the path based on logs are hard.

Challenge 3

Challenges

- Challenge 1: Sourcing Log Messages
- Challenge 2: Reconstructing Execution Paths
- Challenge 3: Profiling Failure-Related Code Snippets.

System	SLOC*
MapReduce	721K
HDFS	706K
HBase	912K
Cassandra	1.1M
ZooKeeper	184K

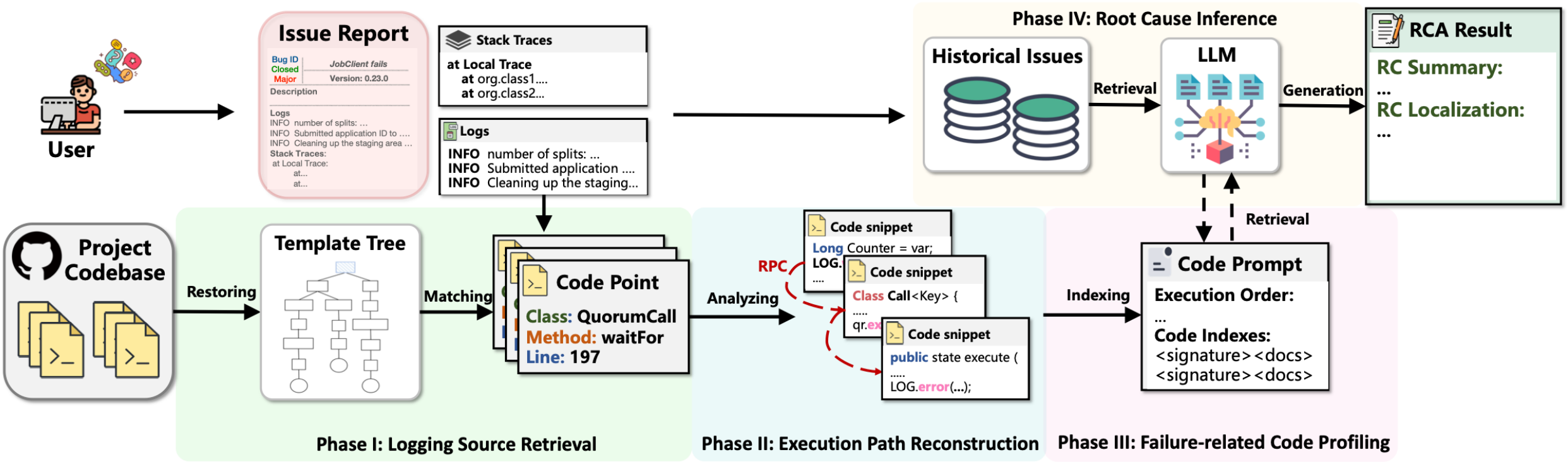


Poor Maintainer

Log-related code is long & Codebase is large

Methodology

COCA : the first **co**de knowledge enhanced root **c**ause **a**nalysis approach



Methodology: Phase 1

Phase 1: Logging Source Retrieval

- Restoring Logging Statements
- Log Template Matching

```
String msg = String.format(
  "Waited %s ms (timeout=%s ms) for a response for %s", waited, millis, operator);
if (!successes.isEmpty()) {
      msg += ". Succeeded so far: " + String.join(",", successes.keySet());
}
if (successes.isEmpty() && exceptions.isEmpty()) {
      msg += ". No responses yet.";
}
LOG.warn(msg);
```

hdfs/qjournal/client/QuorumCall.java

Fig. 4: An example of logging statement restoring.

Algorithm 1: Template Matching Procedure

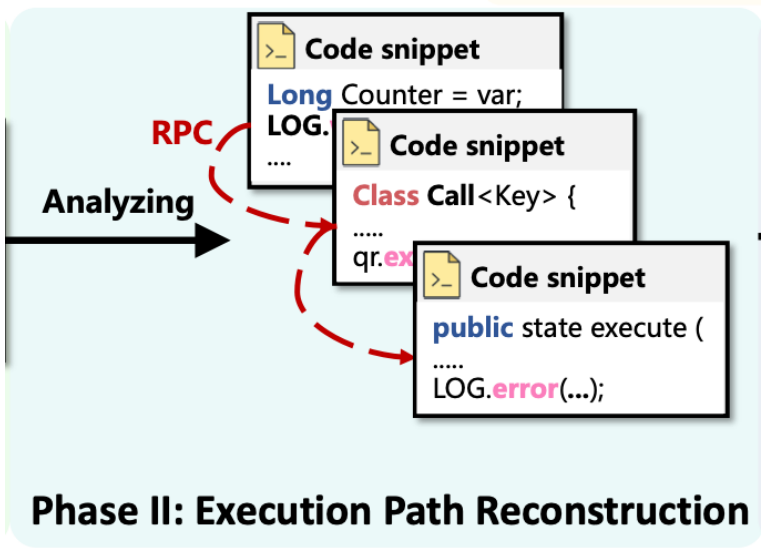
```
1  Input: Template prefix tree; Log message L;
2  Output: Matched template t_m;
3  Function match(root, L, results):
4      if len(L) = 0 and root is a leaf node then
5          add template of root to results; return;
6      if L[0] in root.children then
7          match(root[L[0]], L[1:], results);
8      if <*> in root.children then
9          for i ← 0 to len(L)-1 do
10             if L[i] in root[<*>].children then
11                 match(root[<*>], L[i:], results);
12 root ← root node of the template tree;
13 L, results ← split_tokens(log message), [ ];
14 match(root, L, results);
15 t_m ← most_static_template(results);
16 return t_m;
```

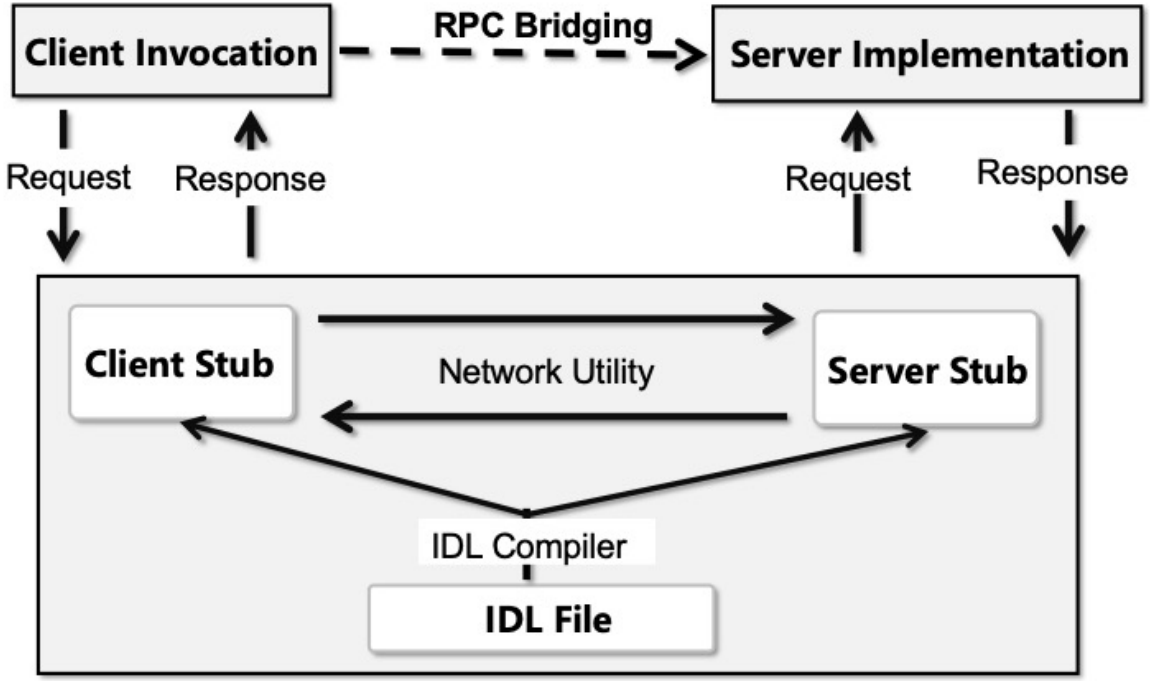
Methodology: Phase 2

Phase 2: Execution Path Reconstruction

- Static Call
- RPC Call

RPCBridge



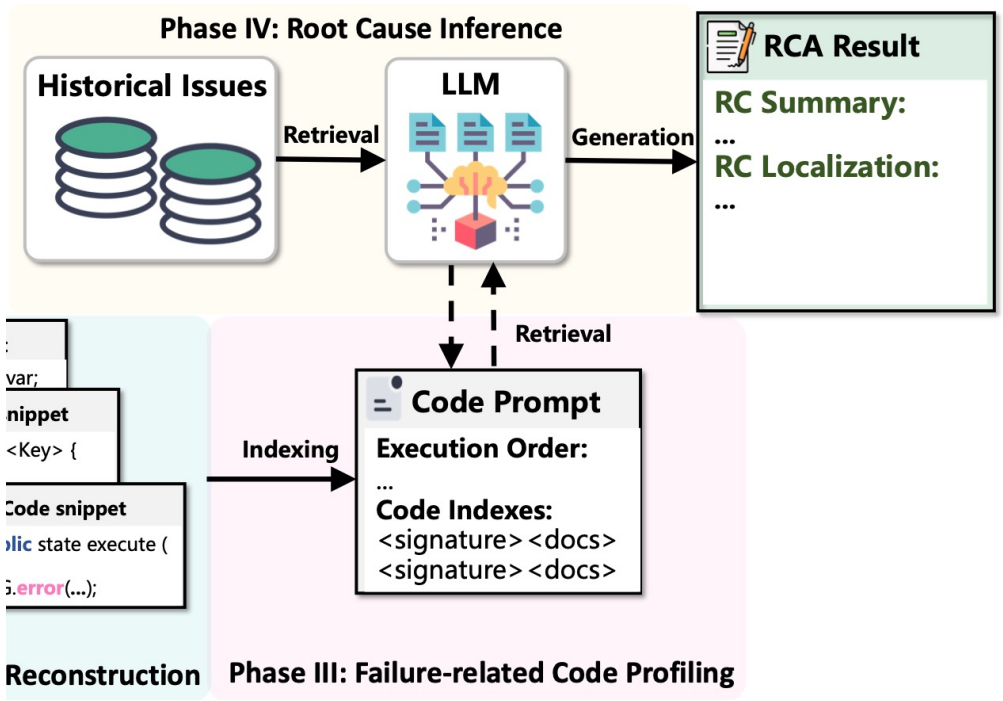


Methodology: Phase 3

Phase 3: Failure-Related Code Profiling

- Code Snippet Indexing
- Code Snippet Retrieval

Phase 4: Root Cause Inference



Code Prompt

Instruction:
Please review the issue report and its execution paths, then identify and return the method signature that requires detailed review for further diagnosis of the issue report.
Issue Report: ...
Execution Paths: Method A -> Method B -> Method C...
Code Indexes:
Signature: <JobStatus submit (Job job, Cluster cluster)>
Docs: /** Internal method for submitting jobs to the system...
* @param....
* @throws **IOException** */
Signature: ...

<Returned method **signatures**>
.....

RCA Prompt

Instruction:
Please infer, localize and summarize the root cause of above input for further issue solving. Specifically, You have two tasks: root cause...
Issue Report: ...
Execution Paths: Method A -> Method B -> Method C...
Code Snippets: JobStatus **submit** (Job job, Cluster cluster)...
Historical Issues: <Issue Report A with diagnosis result>

Evaluation

Dataset: Collected and labelled based on JIRA issue report platform (D-Bug Set).

RCA Tasks:

- Localization (Exact Match and Top-k):
  - Non-code-related components:
    - System resources (e.g., network, RPC)
    - Abstract concepts (e.g., race conditions, deadlocks)
  - Code-related components:
    - Specific classes (e.g., JobClient, ResourceManager)
- Summarization (Similarity and Human Evaluation)

TABLE I: Details of the datasets.

System	SLOC*	Failure Types (#)	# Issues
MapReduce	721K	Hangs (7), Inconsistent state (5), Incorrect result (5), Resource exhaustion (2), Resource leak (1), Unexpected termination (6)	26
HDFS	706K	Data loss (1), Hangs (1), Inconsistent state (1), Incorrect result (8), Resource leak (1), Unexpected termination (3)	15
HBase	912K	Data loss (4), Hangs (8), Inconsistent state (2), Incorrect result (2), Unexpected termination (8)	24
Cassandra	1.1M	Hangs (6), Inconsistent state (5), Incorrect result (20), Unexpected termination (2)	33
ZooKeeper	184K	Hangs (4), Incorrect result (2), Resource leak (2)	8
Total	-	Data loss (5), Hangs (26), Inconsistent state (13), Incorrect result (37), Resource exhaustion (2), Resource leak (4), Unexpected termination (19)	106

Evaluation: RQ1

How effective is COCA in RCA compared with existing methods?

TABLE II: Root cause analysis results from both *summarization* and *localization* dimensions for each system.

System	Model	Root Cause Summarization					Root Cause Localization		
		BLEU-4	ROUGE-1	METEOR	Semantics	Usefulness	Exact Match	Top-3	Top-5
MapReduce	Base model	0.147	0.451	0.354	0.781	0.645	0.346	0.615	0.692
	RCACopilot	0.174	0.503	0.396	0.812	0.740	0.346	0.692	0.808
	ReAct	0.172	0.494	0.384	0.804	0.683	0.385	0.654	0.808
	COCA	**0.203**	**0.525**	**0.445**	**0.867**	**0.795**	**0.423**	**0.731**	**0.923**
HDFS	Base model	0.145	0.430	0.346	0.787	0.627	0.200	0.400	0.600
	RCACopilot	0.196	0.479	0.407	0.835	0.650	0.400	0.667	0.800
	ReAct	0.192	0.486	0.413	0.843	0.733	0.400	0.533	0.733
	COCA	**0.219**	**0.501**	**0.454**	**0.883**	**0.754**	**0.467**	**0.800**	**0.800**
HBase	Base model	0.167	0.487	0.385	0.817	0.683	0.250	0.542	0.917
	RCACopilot	0.171	0.505	0.408	0.873	0.734	0.292	0.458	0.958
	ReAct	0.185	0.496	0.416	0.844	0.729	0.375	0.458	0.917
	COCA	**0.200**	**0.526**	**0.435**	**0.893**	**0.745**	**0.417**	**0.667**	**0.958**
Cassandra	Base model	0.149	0.451	0.340	0.784	0.648	0.364	0.576	0.697
	RCACopilot	0.153	0.466	0.392	0.807	0.661	0.424	0.697	0.879
	ReAct	0.165	0.484	0.407	0.796	0.691	0.515	0.727	0.848
	COCA	**0.208**	**0.522**	**0.441**	**0.877**	**0.830**	**0.606**	**0.848**	**0.909**
ZooKeeper	Base model	0.129	0.429	0.333	0.756	0.588	0.250	0.500	0.875
	RCACopilot	0.144	0.468	0.357	0.863	0.613	**0.375**	0.750	1.000
	ReAct	0.136	0.438	0.342	0.828	0.594	0.250	0.625	0.875
	COCA	**0.185**	**0.497**	**0.381**	**0.894**	**0.625**	0.250	**0.875**	**1.000**

Evaluation: RQ2

What are the impacts of different phases in COCA?

Data Leakage Concern.

TABLE III: Ablation Study of COCA.

System	Setting	Root Cause Summarization					Root Cause Localization		
		BLEU-4	ROUGE-1	METEOR	Semantics	Usefulness	Exact Match	Top-3	Top-5
All	COCA	**0.205**	**0.519**	**0.438**	**0.880**	**0.776**	**0.472**	**0.774**	**0.915**
	w/o Logging Source Retrieval	0.145	0.462	0.338	0.815	0.660	0.387	0.557	0.660
	w/o Execution Path Reconstruction	0.171	0.472	0.403	0.827	0.685	0.349	0.632	0.821
	w/ 1-Degree Execution Path Reconstruction	0.189	0.507	0.373	0.872	0.723	0.443	0.698	0.842
	w/ 3-Degree Execution Path Reconstruction	0.201	0.524	0.419	0.891	0.693	0.349	0.745	0.906
	w/o Failure-Related Code Profiling	0.175	0.507	0.399	0.847	0.734	0.396	0.679	0.792
	only w/ Full JIRA Discussion	0.495	0.712	0.775	0.914	0.895	0.849	0.906	0.934

Evaluation: RQ3

How generalizable is COCA with different backbone models?

TABLE IV: The performance of COCA with different backbone models.

System	Framework	Root Cause Summarization					Root Cause Localization		
		BLEU-4	ROUGE-1	METEOR	Semantics	Usefulness	Exact Match	Top-3	Top-5
GPT-4o	Base	0.151	0.455	0.354	0.789	0.648	0.302	0.547	0.745
	COCA	0.205	0.519	0.438	0.880	0.776	0.472	0.774	0.915
	Δ	↑ 35.8%	↑ 14.1%	↑ 23.7%	↑ 11.5%	↑ 19.8%	↑ 56.3%	↑ 41.4%	↑ 22.8%
GPT-3.5	Base	0.124	0.408	0.309	0.755	0.541	0.264	0.472	0.632
	COCA	0.157	0.461	0.361	0.802	0.592	0.368	0.557	0.764
	Δ	↑ 26.6%	↑ 13.0%	↑ 16.8%	↑ 6.22%	↑ 9.4%	↑ 39.4%	↑ 18.0%	↑ 20.9%
LLaMa-3.1-405b	Base	0.150	0.456	0.348	0.764	0.626	0.283	0.557	0.811
	COCA	0.189	0.514	0.427	0.838	0.702	0.434	0.745	0.877
	Δ	↑ 26.0%	↑ 12.7%	↑ 22.7%	↑ 9.7%	↑ 12.1%	↑ 53.3%	↑ 33.9%	↑ 8.1%
Claude-3.5-Sonnet	Base	0.107	0.441	0.337	0.800	0.652	0.349	0.660	0.868
	COCA	0.148	0.470	0.419	0.854	0.744	0.453	0.764	0.943
	Δ	↑ 38.3%	↑ 6.6%	↑ 24.3%	↑ 6.8%	↑ 14.1%	↑ 29.7%	↑ 15.7%	↑ 8.7%
Gemini-1.5-Pro	Base	0.161	0.460	0.361	0.794	0.633	0.321	0.528	0.764
	COCA	0.224	0.523	0.408	0.859	0.737	0.443	0.736	0.887
	Δ	↑ 39.1%	↑ 13.7%	↑ 13.3%	↑ 8.2%	↑ 16.4%	↑ 38.0%	↑ 39.4%	↑ 16.1%

Case Study

Case Study: HBASE-9821

Root Cause: Collision

of scannerID in HBASE system

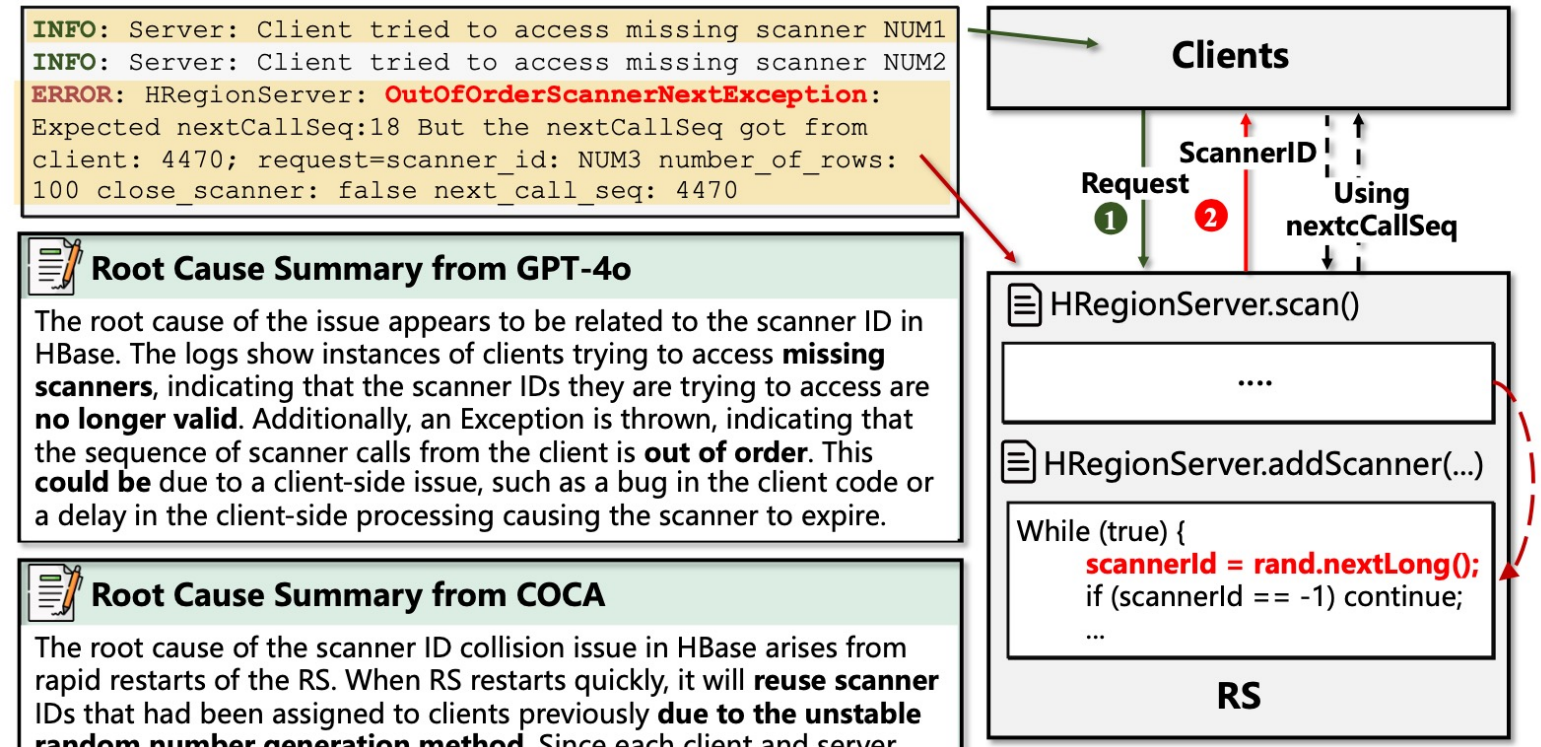
```
INFO: Server: Client tried to access missing scanner NUM1
INFO: Server: Client tried to access missing scanner NUM2
ERROR: HRegionServer: OutOfOrderScannerNextException:
Expected nextCallSeq:18 But the nextCallSeq got from
client: 4470; request=scanner_id: NUM3 number_of_rows:
100 close_scanner: false next_call_seq: 4470
```

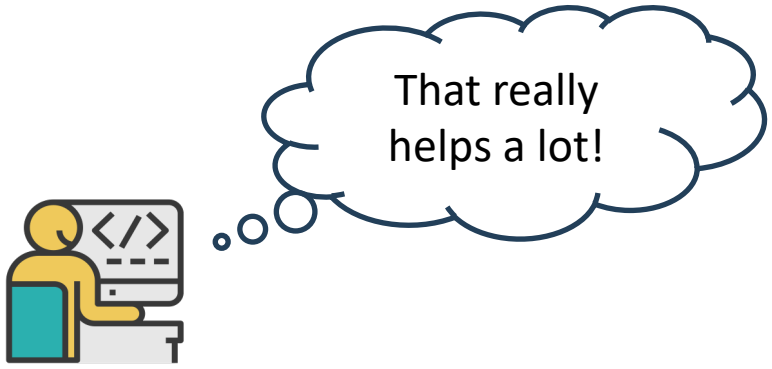
Root Cause Summary from GPT-4o

The root cause of the issue appears to be related to the scanner ID in HBase. The logs show instances of clients trying to access **missing scanners**, indicating that the scanner IDs they are trying to access are **no longer valid**. Additionally, an Exception is thrown, indicating that the sequence of scanner calls from the client is **out of order**. This **could be** due to a client-side issue, such as a bug in the client code or a delay in the client-side processing causing the scanner to expire.

Root Cause Summary from COCA

The root cause of the scanner ID collision issue in HBase arises from rapid restarts of the RS. When RS restarts quickly, it will **reuse scanner** IDs that had been assigned to clients previously **due to the unstable random number generation method**. Since each client and server pairing operates on a unique scanner ID to track data sequence numbers **through variable "nextCallSeq"**, Moreover, if the new RS happens to assign a new client **the same scannerID** as the old one, a nextCallSeq error may occur, causing potential collisions errors.



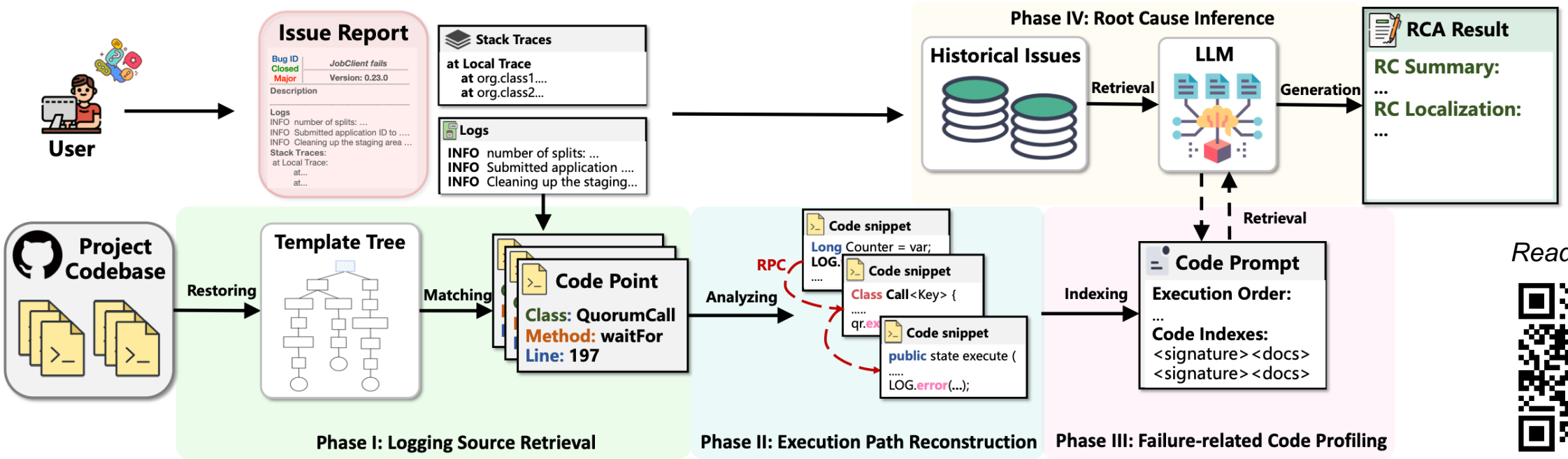


Poor Maintainer

Takeaway

Lessons Learned:

1. Maintainers often face irreproducible user-reported issues.
2. Code knowledge does fill gaps and facilitates RCA.
3. Even code-unrelated issues benefit from code context to build domain understanding.



Read the paper!

Q&A

Thank you sooooo much for the listening!

Read the paper!