

Tsinghua University

Advanced Network Management Project: MicroService System
Troubleshooting

Group Project Report, Fall 2020



Group Name: ANM 小组

Students: Chua Khang Hui, 2020280442 (Leader)

Tee Kah Hui, 2020280402

Yoke Kai Wen, 2020280598

Abstract

The growing of size and complexity of modern microservice-based application had increased the difficulty in system troubleshooting. A fast and accuracy trace anomalies is needed to ease the troubleshooting process. We design a 2 stages root cause detection for microservice system on top of TraceAnomaly and BDSCAN. The system is able to score 153 in the first round of final testing and ranked 6th out of 11 other approaches.

Contents

| | | |
|----------|--|-----------|
| 1 | Problem specification | 1 |
| 1.1 | Context of microservice system troubleshooting | 1 |
| 1.2 | Training data provided | 2 |
| 2 | Data analysis | 2 |
| 2.1 | ESB data | 2 |
| 2.2 | Trace data | 3 |
| 2.2.1 | Trace structure | 4 |
| 2.2.2 | Incomplete traces | 5 |
| 2.3 | KPI data | 5 |
| 2.3.1 | Large number of KPIs | 6 |
| 2.3.2 | Highly irregular KPI time-series patterns | 6 |
| 2.3.3 | KPIs update frequency | 6 |
| 2.4 | Characteristics of ground truth root causes | 6 |
| 2.4.1 | Relationship between docker and os | 7 |
| 3 | Background research | 7 |
| 3.1 | Root cause localisation | 7 |
| 3.2 | Time-series anomaly detection | 8 |
| 4 | Methodology | 8 |
| 4.1 | Host detection | 9 |
| 4.1.1 | Trace Anomaly | 10 |
| 4.1.2 | Trace Pre-processing | 10 |
| 4.1.3 | Homogeneous Service Trace Vector | 11 |
| 4.1.4 | Result Post-processing | 11 |
| 4.1.5 | Result Aggregation | 11 |
| 4.2 | KPI detection | 12 |
| 4.2.1 | DBSCAN | 12 |
| 4.2.2 | Magnitude of Change | 13 |
| 5 | Results | 13 |
| 6 | Conclusion | 13 |

1 Problem specification

In this project, we design an online algorithm to do (1) anomaly detection and (2) root cause troubleshooting of a microservice-based software system, and then deploy it a Tencent Cloud Virtual Machine for real-time testing. The goal is to give a list of predicted root causes for every failure in the microservice system in real time, i.e. we should give the troubleshooting results within 10 minutes of the failure occurrence, in the format of a list with elements of length 2: [[host ID 1, KPI name 1], [host ID 2, KPI name 2] ...].

1.1 Context of microservice system troubleshooting

Recently, microservice architecture has increased in popularity for large-scale software systems in web-based services. This architecture decouples a web service into multiple microservices, such that each microservice can be individually upgraded and maintained. Figure 1 shows multiple calls to microservices involved in one particular user request to a microservice-based web service. Such services contain complex call relationships between different microservices, leading to difficulty in microservice system troubleshooting.

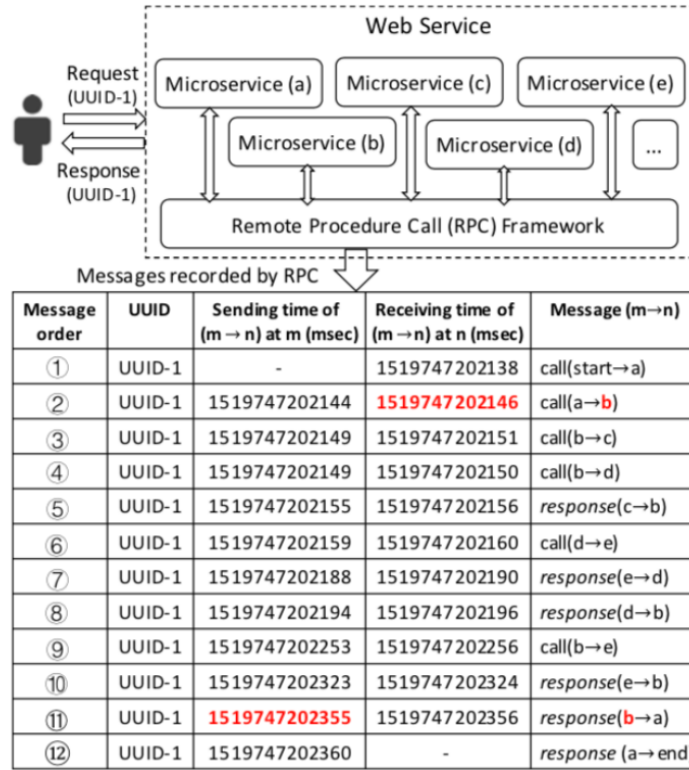


Figure 1: Microservices calls involved in specific user request to a microservice-based web service

1.2 Training data provided

In this project, the microservice system deals with only one kind of service request, and associated microservices are deployed in virtual machines (hosts). The microservice system performance metrics are quantified by 3 kinds of data: (1) ESB business indicator; (2) Trace data; (3) Host KPIs data.

1. **ESB:** The ESB data records the request information every minute, thus providing a high level view of the overall counts, durations and success rates of user requests per minute.
2. **Trace:** A trace corresponds to a user request and has a unique traceID. Each trace contains the sequence of microservice calls involved in processing the user request, and records four useful attributes associated with each microservice call: start time, elapsed time, host and microservice name.
3. **KPI:** Host KPIs data are time-series data with format (timestamp, value). Each host has an associated set of KPIs, with each KPI quantifying a particular aspect of the host. Also, KPIs are updated at varying frequencies (60s, 300s, 3600s etc).

We are provided with 15 days of such data with ground truth anomaly root cause labels. There is a total of 81 root causes, all occurring in 11 out of the 15 days.

2 Data analysis

The greatest challenge in this project is understanding the various forms of data associated with the microservice system, and how they can be used to identify and localise faults.

2.1 ESB data

The ESB data records the request information every minute, including the average time spent processing a request, the number of submitted requests, the number of submitted requests which are successfully completed, and the success rate, with sample records shown in Figure 2.

| | service_name | start_time | avg_time | num | succee_num | succee_rate |
|---|--------------|---------------|----------|-----|------------|-------------|
| 0 | osb_001 | 1606838820000 | 0.5742 | 374 | 374 | 1.0 |
| 1 | osb_001 | 1606838880000 | 0.6431 | 421 | 421 | 1.0 |
| 2 | osb_001 | 1606838940000 | 0.6040 | 405 | 405 | 1.0 |
| 3 | osb_001 | 1606839000000 | 0.6037 | 402 | 402 | 1.0 |
| 4 | osb_001 | 1606839060000 | 0.7301 | 411 | 411 | 1.0 |

Figure 2: ESB sample data

Intuitively, we expect faults to occur when **success rate** < 1 , and when average time is abnormally long. The number of submitted requests could also be indicative of fault (i.e. premature termination and re-submission in event of failure). We plot the numerical fields of the ESB data, comparing their correlation at the corresponding timestamps, particularly focusing on timestamps with low success rate in Figure 3.

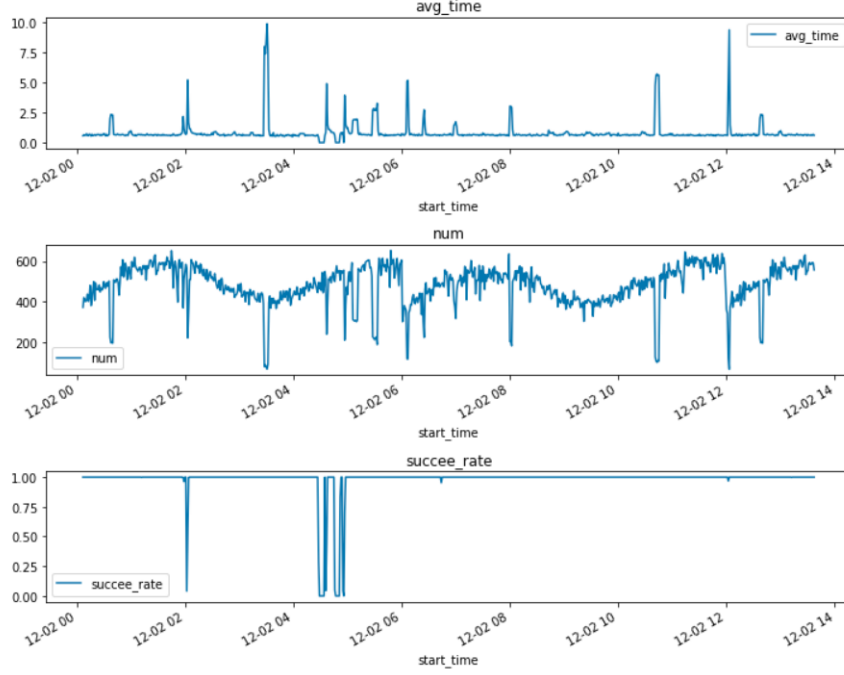


Figure 3: ESB graphs with occasional low success rates: (TOP) average time; (MIDDLE) number of requests; (BOTTOM) success rate

We notice that there are very few timestamps with **success rate** < 1 , and indeed at around these timestamps, **average time** and **number of requests** show abnormal patterns (e.g. jitters, sudden spikes/dips). However, there are also many timestamps with abnormal spikes in **average time** with **success rate** $= 1$. This suggests that we could use **success rate** < 1 and a time-series anomaly detection algorithm on **average time** to detect anomalous timestamps.

2.2 Trace data

Each trace corresponds to a user request and has a unique traceID. The trace data is given in the form of individual microservice call records, with each record containing information about the **callType**, **serviceName**, **dsName** which identifies the microservice being called, **traceId**, **id**, **pid** which helps us derive the parent-child relationship of different call records, **elapsedTime**, **success** which quantifies the behaviour of the microservice call and is thus helpful for detecting anomalies, and importantly **cmdb_id** which indicates the host running the microservice be-

ing called. Naturally, since each microservice call is normally short (ms), the timestamps in the trace data are also separated by uneven intervals in the order of ms. Figure 4 shows sample microservice call records in the trace data.

| | callType | startTime | elapsedTime | success | traceld | id | pid | cmdb_id | serviceName | dsName |
|---|----------|---------------|-------------|---------|----------------------|----------------------|----------------------|---------|-------------|--------|
| 0 | CSF | 1586534660852 | 361.0 | True | d9c4817164d5baee6924 | 9cb7c17164d5baf46931 | 77d1117164d5baee6925 | os_021 | csf_001 | NaN |
| 1 | CSF | 1586534660198 | 212.0 | True | 493c017164d5b85f5792 | 32e6e17164d5b8655795 | 8834517164d5b85f5793 | os_022 | csf_001 | NaN |
| 2 | CSF | 1586534659897 | 246.0 | True | 21d7417164d5b7335890 | 058fb17164d5b7395893 | 0e70f17164d5b7335891 | os_022 | csf_001 | NaN |
| 3 | CSF | 1586534657789 | 1834.0 | True | 3a5c717164d5aef76746 | 9264717164d5aefd6749 | f304f17164d5aef76747 | os_021 | csf_001 | NaN |
| 4 | CSF | 1586534657773 | 1753.0 | True | dcb9917164d5aee66882 | c797a17164d5aee66885 | 4ff4117164d5aee66883 | os_021 | csf_001 | NaN |

Figure 4: Trace sample data

2.2.1 Trace structure

To observe the structure of each trace, we can group each microservice call record by their `traceId`, and construct the hierarchical parent-child call relationships using the `id`, `pid` fields (see Figure 5). We noticed that each complete trace consists of 58 microservice calls, with each trace starting with `callType=osb`, followed by multiple calls to `callType=csf/remoteprocess` (with `csf_001` associated with `app1`, and `csf_002`, `csf_003`, `csf_004`, `csf_005` associated with `app2`), then finally ending with `callType=local/jdbc` which are associated with the Oracle `db` hosts. The physical significance of the trace structure is clear, starting with a service request to one of the two applications and ending with the retrieval/adding of information to the database.

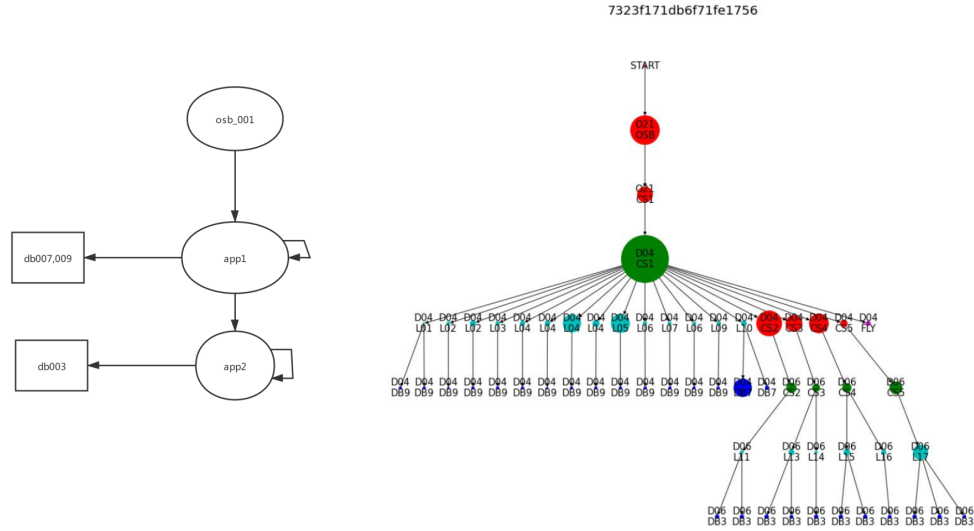


Figure 5: Left: general trace structure; Right: Constructed sample trace, each node being the microservice called, identified by `callType + serviceName + dsName`

2.2.2 Incomplete traces

We notice that there are also occasionally incomplete traces (i.e. traces that do not obey the structure in Figure 5 and contain very few nodes), and the timestamps of these incomplete traces correspond to those in the ESB data with `success_rate < 1`. We infer that these are the traces that correspond to failed user requests which are presumably related to faults in some hosts. Nevertheless, there will also be many cases where faults occur even though traces are complete, in which case we have to rely on other features such as `elapsedTime`, `success` to detect the faults.

2.3 KPI data

Host KPIs data are time-series data of the format `[timestamp, value]`. There are five host types (quantity of each host type indicated in brackets): `linux os` (22), `oracle db` (13), `redis` (12), `container`(2), `docker` (8). Each host type is associated with a unique set of KPIs (number of KPIs indicated in brackets): `os` (52), `db` (47), `redis` (22), `container` (1), `docker` (9) (see Figure 6). The KPIs update at varying frequencies e.g. 60s, 300s, 3600s.

```
os KPInames 51
['Disk wr kbs' 'Disk io util' 'Processor load 15 min'
 'Incoming network traffic' 'Num of running processes' 'ICMP ping'
 'Memory free' 'Agent ping' 'Cache used' 'ss total' 'CPU user time'
 'Outgoing network traffic' 'Disk avgqu sz' 'FS total space' 'Disk svctm'
 'Processor load 1 min' 'CPU iowait time' 'System wait queue length'
 'Disk wr ios' 'Disk rd kbs' 'Disk await' 'Memory used pct' 'Buffers used'
 'CPU system time' 'FS max util' 'Send total' 'Memory available'
 'Disk rd ios' 'Processor load 5 min' 'System block queue length'
 'Memory used' 'Memory available pct' 'CPU util pct' 'FS max avail'
 'CPU idle pct' 'Page po' 'Recv total' 'Swap used pct' 'Page pi'
 'Sent queue' 'Received queue' 'Sent packets' 'Sent errors packets'
 'Shared memory' 'Received packets' 'Received errors packets'
 'Num of processes' 'Zombie Process' 'FS used space' 'FS used pct'
 'Memory total']
db KPInames 47
['DbTime' 'DFParaWrite Per Sec' 'Hang' 'Proc User Used Pct'
 'LFSync Per Sec' 'On Off State' 'Session pct' 'SctRead Per Sec'
 'Redo Per Sec' 'Proc Used Pct' 'User Commit' 'Exec Per Sec'
 'Sess Connect' 'Sess Active' 'Logic Read Per Sec' 'AWS' 'SeqRead Per Sec'
 'MEM Total' 'Login Per Sec' 'TempTbs Pct' 'Call Per Sec'
 'Physical Read Per Sec' 'Row Lock' 'LFParaWrite Per Sec'
 'tnsping Result time' 'TPS Per Sec' 'AIOS' 'ACS' 'MEM Used Pct'
 'MEM real util' 'CPU free pct' 'CPU Used Pct' 'MEM Used' 'Asm Free Tb'
 'Sess Used Undo' 'PGA used total' 'PGA Used Pct' 'Sess Used Temp'
 'UndoTbs Pct' 'Tbs Used Pct' 'Tbs Free Gb' 'New Tbs Free Gb'
 'New Tbs Used Pct' 'Used Tbs Size' 'Total Tbs Size' 'SEQ Used Pct'
 'DbFile Used Pct']
redis KPInames 20
['evicted keys' 'rejected connections' 'instantaneous output kbps'
 'total connections received' 'redis ping' 'used memory peak'
 'used cpu user' 'keyspace hits' 'used cpu sys' 'expired keys'
 'Redis key count' 'blocked clients' 'instantaneous ops per sec'
 'connected clients' 'mem fragmentation ratio' 'keyspace misses'
 'total commands processed' 'instantaneous input kbps' 'used_memory'
 'redis load']
container KPInames 1
['container fail percent']
docker KPInames 9
['container thread running' 'container thread used pct'
 'container cpu used' 'container mem used' 'container fgct'
 'container_fgct' 'container_thread_idle' 'container_session_used'
 'container_thread total']
```

Figure 6: KPI names by host type

2.3.1 Large number of KPIs

In total, there are slightly over $22 \times 52 + 13 \times 47 + 12 \times 22 + 2 \times 1 + 8 \times 9 \approx 2000$ KPI time-series to be examined (at suspicious timestamps) to detect the root cause KPI, which is clearly unreasonable given the real-time requirement. Therefore, we need to find a quick way to narrow down the KPIs to focus on, for instance, by first narrowing down the host machines using the trace data, then only looking at KPIs associated to these hosts, or by using other heuristics.

2.3.2 Highly irregular KPI time-series patterns

Some KPIs remain at zero most of the time; some fluctuate randomly; some spike suddenly with no pattern. As a result, it is difficult to apply a standard time-series anomaly detection algorithm to detect the root cause KPI out of the 2000 over possible choices. This again points to the need for understanding and analysing each KPI by their meaning to come up with some sort of heuristic.

2.3.3 KPIs update frequency

Majority of the KPIs update every 60s which is quite reasonable for our goal of detecting the root cause KPI within ten minutes of fault occurrence, but a significant number of KPIs also update every 300s which might prove a challenge to detect given the small number of points in a ten minute window. There are also some KPIs that update with frequency even lower than that, which should just be ignored entirely since it would be impossible to detect anomalies on those.

2.4 Characteristics of ground truth root causes

Other than guessing what a root cause for a fault could look like, it is useful to examine the ground truth root causes given to us, and their associated characteristics. Table 1 shows that all 81 faults in the given ground truth labels belong to merely a small set of hosts and KPI names.

| Fault description | Host | KPI name |
|---|--|---|
| <i>CPU fault</i> | docker (001 to 008) | [container_cpu_used] |
| <i>network delay</i> <i>network loss</i> | docker (001 to 008) os (001, 009, 017, 018, 020, 021) | [none], [Sent_queue; Received_queue] |
| <i>db close</i> | db (003, 007) | [On_Off_state; tnsping_result_time] |
| <i>db connection limit</i> | db (003, 007) | [Proc_User_Used_Pct; Proc_Used_Pct; Sess_Connect] |

Table 1: Table showing all ground truth fault description, host and KPI names

If the test dataset is anything similar to the training dataset, it is likely that the root cause hosts and KPIs will be part of the small pool of candidates highlighted in Table 1, i.e. we only have to focus on os, db and docker types of hosts, and eight KPI types. This solves the problem of having too many KPIs to examine. In fact, this table suggests that once we determine the root cause host, a random guess at the KPI has a high chance of being correct.

2.4.1 Relationship between docker and os

It is also noteworthy that os (017 to 020) each hosts two dockers. This suggests that when both dockers under the same os are identified to be anomalous, it is likely that the hosting os is the root cause.

3 Background research

We mainly focused our research on two areas: (1) techniques for root cause localisation for microservice systems; (2) techniques for time-series anomaly detection.

3.1 Root cause localisation

TraceAnomaly [1] and MonitorRank [2] both exploit the microservice call graph structure to localise the root cause. While MonitorRank [2] represents the trace data into a graph network and applies a random-walk based algorithm to identify anomalous microservices, TraceAnomaly [1] eschews graph-based algorithms altogether, and instead encodes the microservice invocation trace path and response times into a feature vector, which is then fed into a VAE network to predict whether the trace is anomalous or not, assuming that such an unsupervised high-capacity model will be able to effectively learn the hundreds of response time distributions conditional on their invocation paths. To localise the root cause hosts involved in the anomalous trace, the deviation from mean of the response time of each dimension in the trace feature vector is checked. TraceAnomaly reports extremely promising results in its paper, and seem particularly apt for our use case as well.

On the other hand, FluxRank [3] does not refer to the trace structure at all, and instead, only looks at the KPI data of each host to rank root cause machines and KPIs by their anomaly scores. In fact, FluxRank adopts a lightweight heuristic-based approach mimicking manual methods - it uses Kernel Density Estimation to quantify and compare changes of a large number of diverse KPIs around a specific time point, construct a feature vector for each host machine comprising of their quantified KPI changes, and apply DBSCAN [4] and a logistic regression based ranking algorithm to rank the root cause machines according to anomaly score. FluxRank is simple and interpretable because it makes many assumptions about characteristics of anomalies, which we could also try applying to our use case. Further, it could serve as a second level of check to TraceAnomaly to confirm whether a certain detected anomalous host is indeed anomalous.

3.2 Time-series anomaly detection

Unsupervised time-series anomaly detection is quite well-established. Since this project involves plenty of time-series data such as ESB and KPI data, it is worth trying a variety of standard anomaly detection algorithms, such as simple deviation from mean and DBSCAN [4]. Robust Random Cut Forest (RRCF) [5], an unsupervised ensemble method for anomaly detection on on-line streaming data, also seems particularly apt for our real-time use case.

Other than standard universal anomaly detection algorithms, we also looked at OmniAnomaly [6], which is specially designed to identify anomalies in multivariate time-series associated with an entity. OmniAnomaly captures the temporal dependence and stochasticity of multivariate time-series (e.g. different KPI time-series on a host machine) with a stochastic recurrent neural network, and determines whether an input multivariate time-series is anomalous or not based on the reconstruction probabilities of its constituent univariate time-series. This is potentially useful for identifying root cause KPIs of a particular host machine.

4 Methodology

We did not use the ESB data for identifying anomalous timestamps as it only updates once per minute and is far too slow, whereas it would be more efficient if we directly observed from the trace data when a fault is occurring. Overall, we use the trace data and the KPI data as input. We use a TraceAnomaly-like algorithm to localise the root cause host from the trace data, and then employ a combination of DBSCAN and a rule-based approach to identify root cause KPIs associated with the identified host. The host localisation and KPI identification form a 2 stages root cause detection algorithm.

The first version of the system design, as shown in Figure. 7, will submit a new root cause once the system detect a different host in the first stage. This results in too many submissions to the judging server. Also, the system usually detect some false detection at the end of the error. The later wrong submission will overwrite the previous. To solve this, we implement a confidence and lock mechanism to constraint the submission. The overview of improved system is shown in Figure. 8. Once the host detection provide the same host, we increase the confident. We make use of the nature of error only last for 10 minute in this project, and implement a lock to stop submission once reached certain level of confident.

Overview - V1

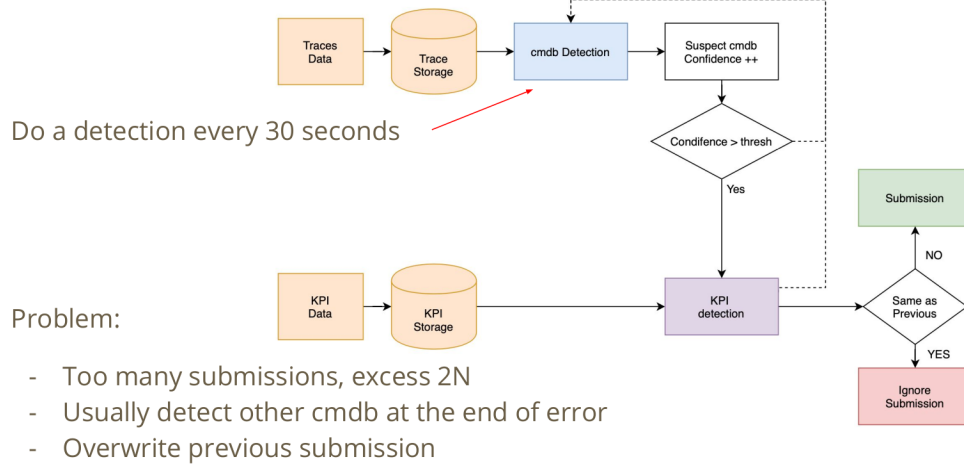


Figure 7: Root cause localisation pipeline version 1

Overview - V2

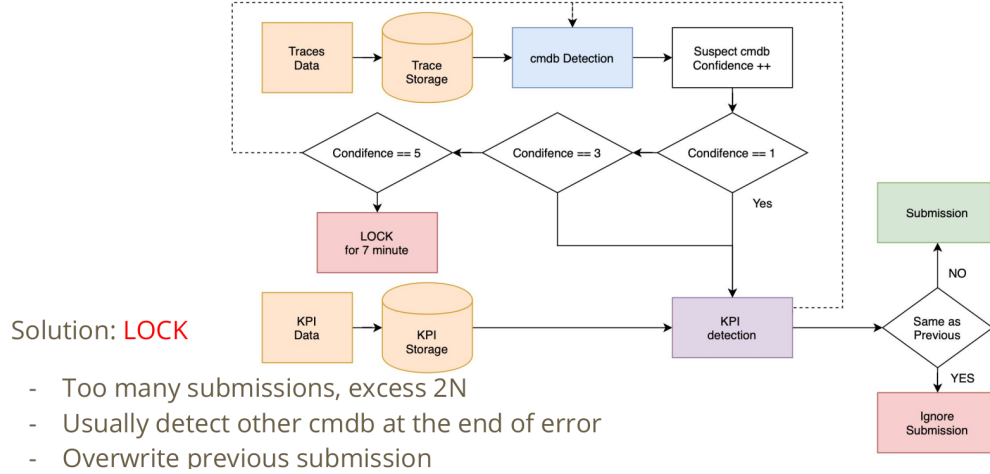


Figure 8: Root cause localisation pipeline version 2

4.1 Host detection

To detect the anomalous host, we borrow the idea from TraceAnomaly [1] and use a Variational Auto-Encoders (VAE) as the backbone. All the traces with the same *trace id* are group together to form a trace group follow a handcrafted structure. Every 30 seconds all the trace group received will be process together. An anomaly score is calculated using VAE for each trace. Next, all the anomalous host detected from the anomalous trace and failed trace are aggregated together to decide the final anomalous host using a voting mechanism. Followed by KPI detection.

The overview of host detection is shown in Figure. 9

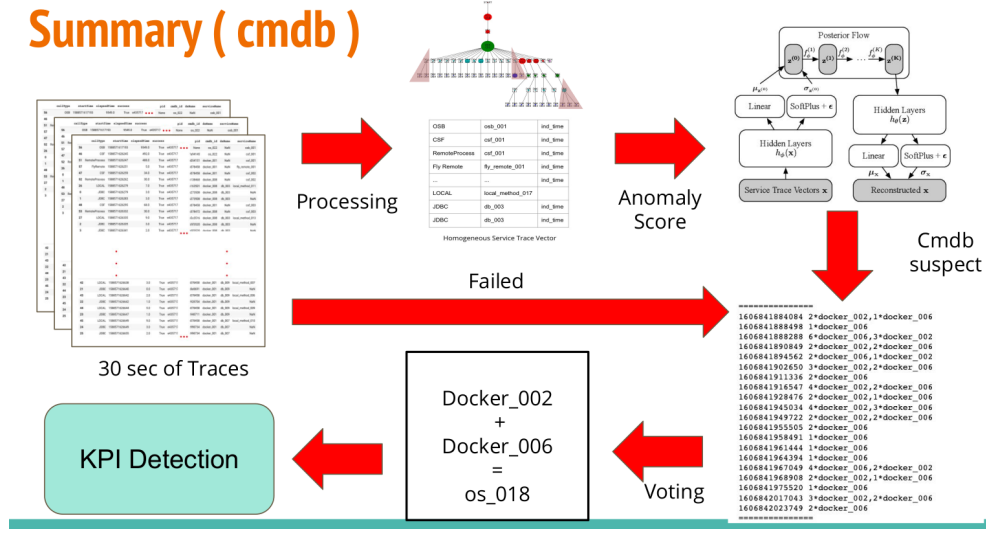


Figure 9: Detailed pipeline

4.1.1 Trace Anomaly

In our approach, we follow similar idea from TraceAnomaly [1] for our first stage of root cause detection. TraceAnomaly using Variational Auto-Encoders (VAE) to encode the data distribution to a latent representation then reconstruct the data back. The distance between the data and reconstructed data will be the anomaly score. The data will be mark anomaly is the anomaly score is beyond a hand-picked threshold. The host detection consists of 3 steps. First, trace data pre-processing. Trace data is grouped and transform to a specific format named service trace vector that encodes both invocation path and individual response time of each service. Second, indicate the anomaly in traces. Every completed trace is analysed and an anomaly score is computed and compare to a threshold. Lastly, anomalous traces is grouped together using a 30 second window and indicate the anomaly services using a voting mechanism. The TraceAnomaly is able to perform in real-time.

4.1.2 Trace Pre-processing

Traces are grouped together by *trace id* to form a complete trace group. For each of the individual trace, an *individual time* is calculated by its *elapsed time* minus the sum of its children's *individual time*. Trace group with more than 57 trace will be use as input for TraceAnomaly; otherwise, is will be treated as incomplete trace group.

4.1.3 Homogeneous Service Trace Vector

After analysing all the trace group, all the normal trace group have length of 57 to 59. Then we handcraft a vector structure of length 60 with the union of all trace groups, named Homogeneous Service Trace Vector (STV). As shown in Fig. 6, the differences in trace group are highlighted in red and service trace vector is constructed using from the trace group. For *CallType* with same *ServiceName*, the trace are sorted by *StartTime*. If any entry in the service trace vector is empty, we filled it in with -1. Finally the STV is standardised using the means and standard deviation before passing to TraceAnomaly.

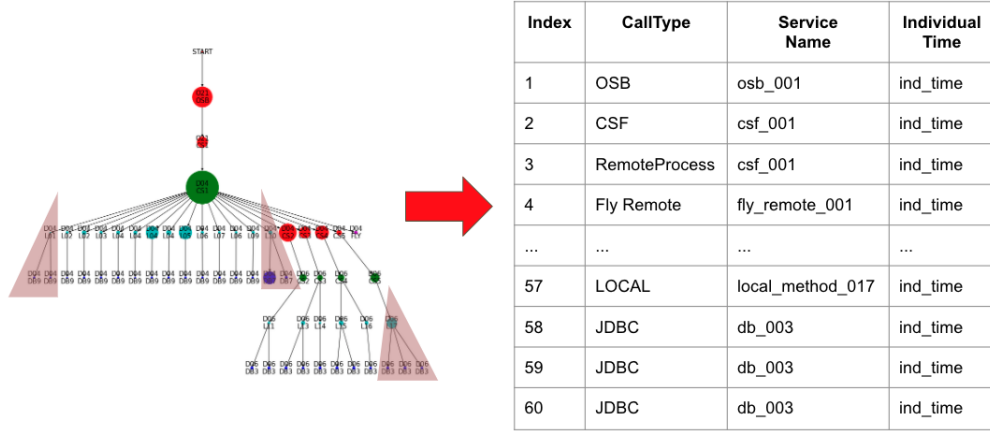


Figure 10: Construction of Homogeneous Service Trace Vector from Different Trace Group

4.1.4 Result Post-processing

Every trace group will get an anomaly score, and we mark it as anomaly if the score excess a handpicked threshold value. The STV of the anomaly trace group will compared with the mean and standard deviation(std). The particular trace will mark as anomaly if it value is outside the range of $mean \pm 3 * std$. The hosts of these anomaly trace are pass to the next stage for anomaly host detection.

4.1.5 Result Aggregation

Together with the host of unsuccessful traces, the occurrence of anomaly hosts are aggregated every 30 seconds. If the most frequent host is more than the second frequent host by a big margin, we assume the root cause in on the host. If this is not the case, we further analyse top 3 frequent hosts. If there is 2 *dockers* in the top 3, we refer to the relationship between *docker* and *os* to decide between multiple root cause from different *dockers* or a single *os* which hosts them. The detected host(s) will send to the next step for KPI detection.

4.2 KPI detection

4.2.1 DBSCAN

Density-based spatial clustering of applications with noise (DBSCAN) DBSCAN [4] is a density-based clustering non-parametric algorithm. It groups together data points by using Euclidean distance and number of neighbouring points. It then marks any point that falls into low-density region as an outlier. There are two key parameters of DBSCAN,

1. **eps**: The distance that specifies the neighbourhoods. Two points are neighbours if the distance between them are less than or equal to eps.
2. **min_samples**: Minimum number of data points to define a cluster.

Based on these two parameters, points are classified as core point, border point, or outlier:

1. **Core point**: A point is a core point if there are at least minimum number of points (including the point itself) in its surrounding area with radius eps.
2. **Border point**: A point is a border point if it is reachable from a core point and there are less than minimum number of points within its surrounding area.
3. **Outlier**: A point is an outlier if it is not a core point and not reachable from any core points.

In our approach, we used DBSCAN to detect anomalous KPI for docker and db type anomaly. First, we mapped CMDB_ID and KPI name into respective integer. With that, we obtained a $n \times 3$ feature vector. Then we fit the input into scikit-learn DBSCAN model with maximum distance of 0.3 and set minimum number of samples as 10. Any outlier detected will be appended into a list and sorted in ascending order of timestamp. We used 30 minutes data as the input to our DBSCAN algorithm.

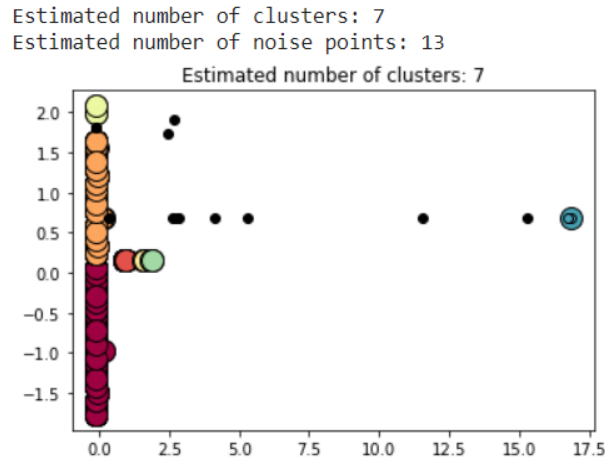


Figure 11: Result from DBSCAN

4.2.2 Magnitude of Change


We used a different approach to locate the root cause of anomaly in OS. The reason is because the value changes are relatively small for OS type of KPI and hence DBSCAN unable to mark them as outliers. To overcome this challenge, we used a statistical approach to detect the anomalous KPI. First, we group the same KPI and sort them in ascending order of timestamp. Next, we calculate the absolute change in value and changes in term of percentage for current log and its previous log. We then mark any data points that has a change percentage above threshold as anomaly. However, this couldn't give us an accurate result due to two reasons.

1. KPI changes are small and the change percentage is below our threshold.
2. Prolonged anomalous value causing the change percentage is low when only compare to previous data.

To overcome these issues, we use a rolling window to smoothen the changes. We calculate the percentage changes in term of rolling mean. This gives us a more accurate prediction.

5 Results

We ranked 6th with a score of 153 on 54 hours of test data, as shown in Figure 12.



| rank | group name | score | highest score |
|------|------------------|-------|---------------|
| 1 | 学堂路车神 | 985 | 200 |
| 2 | meow meow | 624 | 144 |
| 3 | Veritaserum | 535 | 125 |
| 4 | MSSherlock | 369 | 118 |
| 5 | flower group | 161 | 99 |
| 6 | ANM小组 | 153 | 18 |
| 7 | ANMG | 78 | 9 |
| 8 | The Anomalies | 70 | 19 |
| 9 | study group | 0 | 0 |
| 10 | Learning Failure | 0 | 9 |
| 11 | DANM! | 0 | 0 |

Figure 12: Leaderboard score

6 Conclusion

Our application detect root cause using a 2 stages approach. First localise the *cmdb* that cause the problem using TraceAnomaly, then identify the KPI using DBSCAN or magnitude of change on time-series KPI data. This application reached real-time performance in this project with the network traffic around 600 requests per minute.

References

- [1] P. Liu, H. Xu, Q. Ouyang, R. Jiao, Z. Chen, S. Zhang, J. Yang, L. Mo, J. Zeng, W. Xue, and D. Pei, “Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks,” in *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, 2020, pp. 48–58.
- [2] M. Kim, R. Sumbaly, and S. Shah, “Root cause detection in a service-oriented architecture,” *SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, p. 93–104, Jun. 2013. [Online]. Available: <https://doi.org/10.1145/2494232.2465753>
- [3] P. Liu, Y. Chen, X. Nie, J. Zhu, S. Zhang, K. Sui, M. Zhang, and D. Pei, “Fluxrank: A widely-deployable framework to automatically localizing root cause machines for software service failure mitigation,” in *30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019, Berlin, Germany, October 28-31, 2019*, K. Wolter, I. Schieferdecker, B. Gallina, M. Cukier, R. Natella, N. R. Ivaki, and N. Laranjeiro, Eds. IEEE, 2019, pp. 35–46. [Online]. Available: <https://doi.org/10.1109/ISSRE.2019.00014>
- [4] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, “A density-based algorithm for discovering clusters in large spatial databases with noise,” in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, ser. KDD’96. AAAI Press, 1996, p. 226–231.
- [5] S. Guha, N. Mishra, G. Roy, and O. Schrijvers, “Robust random cut forest based anomaly detection on streams,” in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML’16. JMLR.org, 2016, p. 2712–2721.
- [6] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, “Robust anomaly detection for multivariate time series through stochastic recurrent neural network,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 2828–2837. [Online]. Available: <https://doi.org/10.1145/3292500.3330672>