Business Information Systems project

Zucchetti Use Case

Davide Cazzetta 976585

2021

Contents

1 Scenario

Zucchetti offers a range of products in Italy and Europe, allowing customers to gain major competitive advantages and to rely on a single partner for all their IT needs. Software and hardware solutions, innovative services designed and developed to meet the specific needs of (i) companies in any sector and of any size, including banks and insurance companies; (ii) professionals (public accountants, labour consultants, lawyers, bankruptcy liquidators, public notaries, etc.), classified associations and tax advisory centres; (iii) local and central public administrations (municipalities, provinces, regions, ministries, public corporations, etc.).

A multi-disciplinary approach is used to identify the needs of customers and to coordinate, manage and create projects, which allows the Zucchetti Group to develop products and services of the highest quality. To get this goal continuous monitoring of the performances of Zucchetti IT infrastructure is required.

Some key applications are based on an infrastructure including three main components: an *Nginx*[1] frontend dispatches calls to a *Tomcats Cluster*[2] that directs queries to an *SQL Database* (see figure 1). These three components generate separated log files that must be unified to develop a consistent performance analysis. In general the goal of the organisation is detecting the overall performance of the system (through the the log generated by each component).

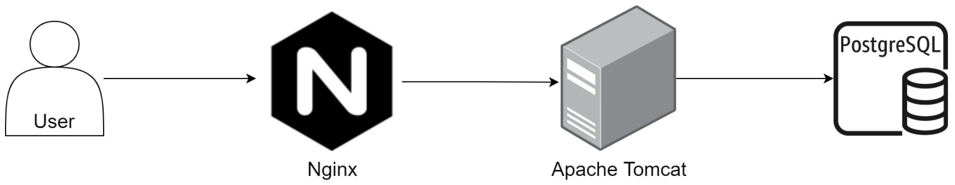

Figure 1: Example of Zucchetti infrastructure

[1]Nginx is a web server that can also be used as a reverse proxy, load balancer, mail proxy and HTTP cache.[6]

[2]Apache Tomcat is a free and open-source implementation of the Java Servlet, JavaServer Pages, Java Expression Language and WebSocket technologies. Tomcat provides a "pure Java" HTTP web server environment in which Java code can run.[4]

2 Goals and Challenges

Relative to the three components, each of them generates separated log files that must be unified to develop a consistent performance analysis. Relating Nginx and Tomcat is quite simple because time, *IP* address, and the *URL* of the calls identify unique cases. Relating Tomcat and the SQL Database is more complex because Tomcat aggregates connections for *READ* operations in pools (*WRITE* operations have a reserved connection), moreover the SQL log is truncated, excluding queries with low execution time.

To reach the goal of detecting the overall performance of the system a performance analysis is needed, and to this aim variant analysis can support these operations, in particular to identify the segment that capture the common behaviour of the system and that can be obtained by verifying which percentile is below a performance score that is considered safe.
Before doing all these operations there is one more thing to do: outlier cases should be excluded from the analysis as they can significantly shift the statistics without having a real impact on the system performance experienced by the users.

Another goal that can be achieved through performance analysis is to identify the possible causes of performance drops. To this aim variants that are strongly correlated to performance drops must be isolated to assess if significant dependences with performance drops can be verified.

3 Knowledge Uplift Model

Table 1 represents the Knowledge Uplift Model.

	Input	**Acquired Knowledge**		**Output**
		Analytics/Models	**Type**	
Step 1	Nginx + Tomcat + SQL log files	Find strategy to unify all log files	Descriptive	Single unified event log
Step 2	Step 1	Performance Analysis & Data Visualization	Descriptive	Most significant performance drops and outliers
Step 3	Step 1 Step 2	Remove outliers	Descriptive	New log file without outliers
Step 4	Step 3	Performance Analysis & Variant Analysis	Prescriptive	Percentile below a safe performance score
Step 5	Step 4	Variants comparison	Descriptive	Dependencies between variants and performance drops

Table 1: Knowledge Uplift Model

Since log files are generated separately by the three components, the first step is find a strategy to unify all log files and generate a single unified log file.

At this point it is possible to compute every process mining techniques to achieve all goals of the organisation: to detect the overall performance of the system a Performance Analysis and Data Visualization have to be computed in order to identify the most significant performance drop and outliers.
These latter have to be excluded for the next performance analysis because they can significantly shift the statistics without having a real impact on the system performance experienced by the users.

Now through a variant analysis I can verify the percentile that is below a safe performance score.

The next step consist in verify dependencies between specific variants and performance drops can be considered statistically significant, so it is possible to identify variants that could cause performance drops.

The last step is not mentioned in the knowledge uplift model table because it is based on personal opinions and experiment: the goal is to study if control-flow patterns could be used to identify performance drops and if predictive analytics could be tested.

3.1 Data Flow Model

A Data Flow model showing all steps can be viewed in figure 2, in particular three level are represented:

- Context diagram
- Level 0 Diagram
- Level 1 Diagram: that describe the 1.1 Performance Analysis process

With regard technological aspects in the design of the Data Flow Model I have considered these four:

- **Stream vs Batch processing**: in this case we can speak about batch processing because the program takes a set of data files as input (the three log files), process the data, and produce a data file as output (for example the new unified and filtered log file)
- **ETL vs ELT**: here the project refer to ETL because once the data is extracted, it's transformed into some usable format and then stored.

- **Consistency vs Availability**: with respect to the Zucchetti architecture consistency is the right aspect since there is relational database (like PostreSQL [3])

- **Pseudonymization vs Anonymization**: in this project we have no anonymization since there is the IP address.

[3]PostgreSQL is a powerful, open source object-relational database system with over 30 years of active development that has earned it a strong reputation for reliability, feature robustness, and performance.[3]

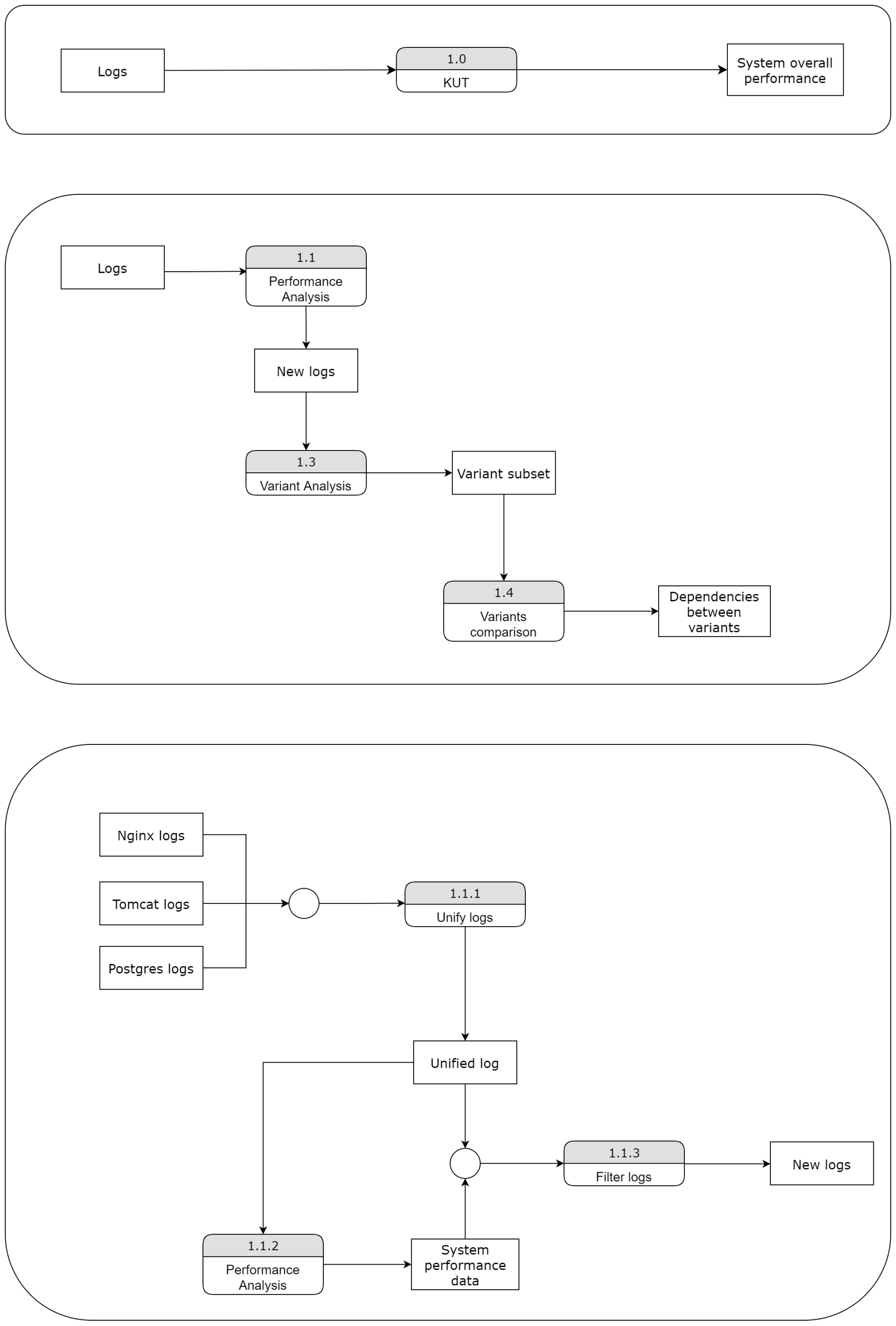

Figure 2: Data Flow Model

4 Materials and Methods

The dataset available consist of three log files with *.log* format:

1. WEB_SERVER.log: where we can find user requests composed by IP, group ID, timestamp, API request, status code and URL.

```
- 172.19.2.151 - - 1622791258 [04/Jun/2021:09:20:58 +0200] "GET /
   infinity41_sp27p/jsp/index.jsp HTTP/1.1" 200 5086 "https://
   orca.aulla.zucchetti.it/infinity41_sp27p/" "Mozilla/5.0 (
   Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
   Gecko) Chrome/91.0.4472.77 Safari/537.36" "-"
- 172.19.2.151 - - 1622791258 [04/Jun/2021:09:20:58 +0200] "GET /
   infinity41_sp27p/fonts/OpenSans.css HTTP/1.1" 200 1007 "https
   ://orca.aulla.zucchetti.it/infinity41_sp27p/SpTheme_Fusion/
   portalstudio.css" "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
   AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.77
   Safari/537.36" "02F71B36557BF32B24868642884ABF38"
- 172.19.2.151 - - 1622791258 [04/Jun/2021:09:20:58 +0200] "GET /
   infinity41_sp27p/servlet/SPPrxy/koe7qxbk/stdFunctions.js HTTP
   /1.1" 200 59861 "https://orca.aulla.zucchetti.it/
   infinity41_sp27p/jsp/login.jsp" "Mozilla/5.0 (Windows NT 10.0;
    Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome
   /91.0.4472.77 Safari/537.36" "02F71B36557BF32B24868642884ABF38
   "

...
```

2. APPLICATION_SERVER.log: this log file contains all logs generated by Tomcat

```
172.19.2.151 - - [04/Jun/2021:09:20:58 +0200] "GET /
   infinity41_sp27p/jsp/index.jsp HTTP/1.1" 200 206
172.19.2.151 - - [04/Jun/2021:09:20:58 +0200] "GET /
   infinity41_sp27p/fonts/OpenSans.css HTTP/1.1" 200 6274
172.19.2.151 - - [04/Jun/2021:09:20:58 +0200] "GET /
   infinity41_sp27p/servlet/SPPrxy/koe7qxbk/stdFunctions.js HTTP
   /1.1" 200 216931

...
```

   In this file we have info similar to the previous one: the request's IP, the API request and the status code.

3. DATABASE_SERVER.log: here you can find all database read and write operations

```
2021-06-04 09:20:58.651 CEST [14552] postgres@infinity41_sp27p
    LOG:  esecuzione di <unnamed>: select * from persist where
    code='PlanCheckDate'
2021-06-04 09:20:58.674 CEST [14552] postgres@infinity41_sp27p
    LOG:  esecuzione di <unnamed>: select count(*) as nNum from
    cpusers c
2021-06-04 09:20:58.675 CEST [14552] postgres@infinity41_sp27p
    LOG:  esecuzione di <unnamed>: select count(*) as nNum from
    cpusers c where code = 1

...
```

This file contain info about database operation: timestamp, database name and operation description.

The software tool used are Pandas[4] and PM4PY[5].

5 Solution

The code is available on my personal Github account: `https://github.com/Davydhh/Esame-BIS/blob/master/solution.py`.

Since the library PM4PY needs *.csv* or *.xes* files with a delimiter it is necessary to preprocess the three log files:

- WEB_SERVER.log: there isn't a specific delimiter, so I have decided to use the space character as delimiter and then merge the right columns.

```python
df_web_server = pd.read_csv("WEB_SERVER.log", sep=' ', header=None, nrows=25)
df_web_server[5] = (df_web_server[5] + df_web_server[6]).str.strip("[]")
df_web_server[5] = pd.to_datetime(df_web_server[5],
↪  format='%d/%b/%Y:%H:%M:%S%z')
df_web_server = df_web_server.drop(labels=[0, 2, 3, 6, 9, 11, 12], axis=1)
df_web_server = df_web_server.rename(columns={1: "IP", 4: "ID", 5:
↪  "TIMESTAMP", 7: "REQUEST", 8: "CODE", 10: "URL"})
```

[4]`https://pandas.pydata.org/`[2]
[5]`https://pm4py.fit.fraunhofer.de/`[1]

- APPLICATION_SERVER.log: this case is almost identical to the previous one.
- DATABASE_SERVER.log: this file is very critical since it is composed by a lot of lines and also anomalies, for example there are empty lines and lines that wrap, making the automatic processing impossible to compute and forcing the developer to resolve them manually. In particular to find the lines that generate the error (see above some examples) I have used the optional keyword parameter *nrows* of the method by trial and error. About the structure of the file I have adopted an approach very similar to the previous two.

```
Query Text: Delete from ba_umiart001 where UAFLGART=1 AND
   UAKEYART='kddvagacdk' AND UACODUMI='N'  and CPCCCHK='
   pfm6qd1ajv'
  Righe 100970:          Filter: ((ba_umiart001.cpccchk)::text =
      'pfm6qd1ajv'::text)
```

Listing 1: Error 1 example

```
1 - Creazione tabella elenco magazzini:04-06-2021|11:30:31
  2 - Creazione tabella Movimenti magazzino per inventario
     :04-06-2021|11:30:31
  3 - Creazione tabella ultimi costi/listini:04-06-2021|11:30:31
  3.1 - Valorizza tabella Ultcos:04-06-2021|11:30:31
  Valorizzazione ultimo costo eseguita
  4 - Creazione tabella 001 - inventario per magazzino base
     :04-06-2021|11:30:31
  5 - Creazione tabella 002 - inventario per articolo + magazzino
     :04-06-2021|11:30:32
  6 - Creazione tabella 003 - inventario per articolo
     :04-06-2021|11:30:32
  7 - Creazione tabelle  - Valorizzazioni LIFO/FIFO
     :04-06-2021|11:30:32
  Valorizzazione LIFO a Scatti eseguita
  Valorizzazione LIFO Continuo eseguita
  Valorizzazione FIFO Continuo eseguita
  8 - Valorizzazione costo medio ponderato per movimento
     :04-06-2021|11:30:32
  Elaborazione Costo medio ponderato per movimento eseguita
  Elaborazione Costo medio ponderato per movimento eseguita
  Elaborazione Costo medio ponderato per movimento eseguita
  Elaborazione Costo medio ponderato per movimento eseguita
  9.1 - Inserimento testata inventario:04-06-2021|11:30:32', NULL
     , 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 'yequcd4vml')
```

Listing 2: Error 2 example

```
N. 4 record inseriti su 4
  N. 4 dettagli magazzino inseriti su 4
  ', '', 'fyhb4dx5dg')
```

Listing 3: Error 3 example

The second issue is relative to unify the Nginx and Tomact logs since there are logs which are equal, consequently the merge method of Pandas library, by performing the Cartesian product, generates other logs; for example the first four POST operation in the unified log file are sixteen. So my solution is to join the column of the Tomcat log file representing the timestamp in the Nginx file (supposing that there is a one-to-one relationship between the two log files).

The most difficult challenge is unify the Tomcat and SQL log files since there aren't any kind of direct connections between: analyzing the Nginx-Tomcat unified log file and the SQL log file we can see that are respectively long 7019 and 110379, so we can roughly suppose that one HTTP request correspond to ∼16 database operations. Therefore, supposing the JDBC[6] has a FIFO policy, I can assume that the first request is linked to the first sixteen database operations approximately. As a result to calculate the cycle time for each request I've made the difference between the first and the last database operation associated to the corresponding request.

Since there are significant performance drops that can shift the statistics, through a Performance Analysis and Data Visualization, outliers (request that take more than 5 seconds) are identified and then removed from the Dataframe.

At this point to perform Variant Analysis the Dataframe must be converted into Event Log; in this case the *CASE_ID_KEY* used is the group ID since group some requests together, and the activity name is the column *REQUEST*. After doing that it is possible to compute performance analysis to the event log to get the lead time[7] and the cycle time[8]. In that case because the cycle time is previously calculated I have decided to reuse it for the next points: for example for each variant I have made the sum of each event composing the variant and then calculate the percentile that is below a performance score, that is 3 seconds.
The same things are done also for the filtered log, that is event logs filtered by per-

[6]Java Database Connectivity (JDBC) is an application programming interface (API) for the programming language Java, which defines how a client may access a database.[5]

[7]the overall time in which the instance was worked, from the start to the end, without considering if it was actively worked or not.[1]

[8]the overall time in which the instance was worked, from the start to the end, considering only the times where it was actively worked.[1]

centage; in this project I have used the 50% percentile noticing an improvement in performance.

To verify if dependencies between specific variants and performance drops can be considered statistically significant, as we are considering scalar variable, I have applied variants comparison, comparing the most performance drop variant with fifty random variants: the result is that for each variant the z is always greater than 1.96, so the P-value is less than 0.05 → dependencies are statistically significant.

The last point is verify if control-flow patterns could be used to identify performance drops. The first thing to do is identify a set of pattern: fro example sequence, synchronisation, parallelisation, iteration, and combination. These can be simple or complex; in this use case I have I found this activity really hard as the activities are very context dependent and can be influenced by some random factors like Tomcat/SQL delay.
To support this step, to have a graphical view of the variants, I have generated a Directly-Follows Graph with the cycle time in every arcs. The figure 3 show only a very small portion of the entire graph but we can notice that is more or less easy to find the request with performance drop (in this case 55 seconds).
About predictive analysis at the moment I don't think that could be tested since the mapping from Tomcat to SQL is very general and approximate.

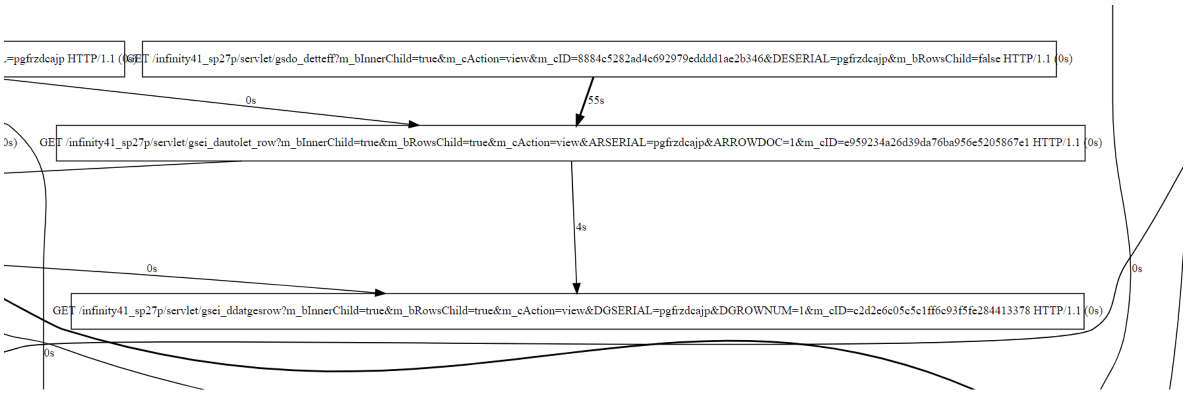
=pgfrzdcajp HTTP/1.1 (0s)
GET /infinity41_sp27p/servlet/gsdo_detteff?m_bInnerChild=true&m_cAction=view&m_cID=8884c5282ad4c692979edddd1ae2b346&DESERIAL=pgfrzdcajp&m_bRowsChild=false HTTP/1.1 (0s)
0s
55s
0s)
GET /infinity41_sp27p/servlet/gsei_dautolet_row?m_bInnerChild=true&m_bRowsChild=true&m_cAction=view&ARSERIAL=pgfrzdcajp&ARROWDOC=1&m_cID=e959234a26d39da76ba956e5205867e1 HTTP/1.1 (0s)
4s
0s
0s
GET /infinity41_sp27p/servlet/gsei_ddatgesrow?m_bInnerChild=true&m_bRowsChild=true&m_cAction=view&DGSERIAL=pgfrzdcajp&DGROWNUM=1&m_cID=c2d2e6c05c5c1ff6c93f5fe284413378 HTTP/1.1 (0s)
0s

Figure 3: Small portion of Directly-Follows Graph

6 Key Contributions

In my opinion there are two interesting parts to this project:

1. The DATABASE_SERVER.log pre-processing activity.

```python
1  my_col = [x for x in range(50)]
2  df_database_server = pd.read_csv("DATABASE_SERVER.log", sep=' ', header=None, names=my_col)
3  df_database_server = df_database_server.fillna('')
4  df_database_server[0] = df_database_server[0] + ':' + df_database_server[1] + "+0200"
5  df_database_server[0] = pd.to_datetime(df_database_server[0], format='%Y-%m-%d:%H:%M:%S.%f%z')
6
7  for i in range(6, 17):
8      df_database_server[5] = df_database_server[5] + ' ' + df_database_server[i]
9
10 del_col = [1, 2, 6]
11 del_col.extend([n for n in range(7, 50)])
12 df_database_server = df_database_server.drop(labels=del_col, axis=1)
13 df_database_server = df_database_server.rename(columns={0: "TIMESTAMP", 3: "ID", 4: "DATABASE", 5: "OPERATION"})
14 df_database_server = df_database_server[df_database_server.DATABASE == "postgres@infinity41_sp27p"].reset_index()
```

In particular the first line allows to get most of one line since every line has different length, while line 7-8 are useful to reconstruct the operation description.

2. The function *performance_analysis* that allows to identify performance drops and percentiles.

```python
def performance_analysis(variants):
    variants_scores = {}
    count = 0
    for key, value in variants.items():
        variants_scores[key] = 0
        for event in value[0]:
            variants_scores[key] += event["TIME_DELTA_DB"]

    performance_drop_variant = max(variants_scores, key=variants_scores.get)

    performance_drop_value = variants_scores[performance_drop_variant]

    print("\n{0:.2g} is the most variant performance drop".format(performance_drop_value))

    plt2 = plt.figure()
    ax1 = plt2.add_subplot(111)
    ax1.plot([x for x in range(len(variants_scores))], variants_scores.values())
    plt.show()

    count = 0
    for value in variants_scores.values():
        if value < 3:
            count += 1

    print("\nPercentile under 3 sec time delta db {:.2f}%".format(count / len(variants_scores) * 100))

    # Remove the variant from the dictionary for the later comparison
    for key, value in variants_scores.items():
        if value == performance_drop_value:
            del variants_scores[key]
            break

    return {0: performance_drop_value, "values": list(variants_scores.values())}
```

This function calculate every variant cycle time by adding the cycle time of each event of which the variant is composed and then find the most performance drop, then show a plot to better identify most performance drop and understand the data distribution and last identify the percentile that is below a safe performance score.

7 Conclusions

The biggest problems with this project are two. The first concerns the structure of the log files since they aren't in *.csv* on in *.hex* format and haven't a well defined structure, from this follow a mandatory pre-processing activity to make the log files compatible with the PM4PY library. The second covers the activity of unifying the Tomcat and SQL log files since there is no a one-to-one mapping between them and we have not a direct connection or reference, this brought me to find a solution that is pretty approximate.
For this reasons this solution is definitely not generalisable to other scenario, first of all for the structure of the log files.

The most important improvement that could be improve the result is surely about the log unification strategy, I think that this activity is crucial for the entire project as in this project all the potential of the variant analysis has not been exploited for this reason.
A possible improvement that I feel like proposing is the inclusion of queuing analysis and simulation, since the arrival times and service times are very variable. For example in this use case we can consider a queuing model like this: P/P/c, where the first P means that the interarrival time distribution is Pareto distribution, the second means that also the service time follow Pareto distribution end c is the number of server.

To conclude, when a much more solid strategy will be found I think that it could be possible to test some predictive analysis to detect in advance some possible performance drops.

References

[1] Fraunhofer FIT. *Cycle Time and Waiting Time.* URL: `https://pm4py.fit.fraunhofer.de/documentation#filtering`.

[2] Pandas team. *Pandas.* URL: `https://pandas.pydata.org/`.

[3] PostgreSQL Team. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* URL: `https://www.postgresql.org/`.

[4] Wikipedia. *Apache Tomcat.* URL: `https://en.wikipedia.org/wiki/Apache_Tomcat`.

[5] Wikipedia. *Java Database Connectivity.* URL: `https://en.wikipedia.org/wiki/Java_Database_Connectivity`.

[6] Wikipedia. *Nginx.* URL: `https://en.wikipedia.org/wiki/Nginx`.