DOI: 10.15439/2022F172
ISSN 2300-5963 ACSIS, Vol. 30

Discovering interactions between applications with log analysis

Łukasz Korzeniowski
Nordea Bank Abp SA , Satamaradankatu 5, FI-00020
Helsinki, Finland
Email: lukasz.korzeniowski@protonmail.com

Krzysztof Goczyła
Gdańsk University of Technology, Faculty of
Electronics, Telecommunication and Informatics,
Narutowicza 11/12, Gdańsk, Poland
Email: kris@eti.pg.edu.pl

***Abstract*—Application logs record the behavior of a system during its runtime and their analysis can provide useful information. In this article, we propose a method of automated log analysis to discover interactions taking place between applications in an enterprise. We believe that such an automated approach can greatly support enterprise architects in building an up-to-date view of a governed system in a modern, fast-paced development environment. Our contribution is the following: we propose a new method for log template generation called SLT (Simple Log Template), we propose a method of extracting knowledge about application interactions from logs, and we validate the proposed methods on a real system running at Nordea Bank. Additionally, we collect statistical information about application logs from the real-life system, based on which we formulate some observations that support our method.**

I. Introduction

ONE of the challenges faced by enterprise architect teams in large organizations is to ensure an up-to-date overview of the systems they are governing. Traditionally, the governance process relies on manual updates to the system representation stored in the architecture repository. However, in the age of microservices and cloud deployments, where significant changes to architecture can take place overnight, this is barely sufficient. There is a clear need for a better, more automated way of maintaining knowledge about systems across an enterprise.

Automation of system knowledge discovery is a big help for enterprise architects, especially in recent times, when new applications are created at an increasing pace and their architecture changes rapidly. Manually updated architecture repositories no longer keep up with the reality of systems deployed to a production environment. This problem is further emphasized by long, heavy, and manual processes of introducing changes to architecture repositories, which do not fit the time-to-market expectations of the business stakeholders. Having a decent representation of current production deployment allows for reasoning about the architecture, detecting non-compliance with internal or external regulations, and helps the development of the enterprise architecture in a planned direction. It also improves building always up-to-date overview of the enterprise system which is beneficial not only for architects but for developers, analysts, and testers for building a mental model of the system they are working with. One of the potential approaches is to utilize application logs which provide a common, always up-to-date view of applications during their runtime.

Application logs have several compelling advantages as compared to other sources of knowledge about the system (e.g., documentation or source code). Firstly, logging is the most widespread way of tracking system behavior that is present in software development since its beginning. Thanks to that we can assume that nearly every software system, even legacy, implements some form of logging. In the reality of many enterprises, logs may prove to be the only common source of knowledge about systems still running on mainframe platforms. Secondly, logs (which traditionally are stored as text files) are usually human-readable and therefore are suitable for processing by text tools. Additionally, they are a mixture of technical information and narrative. Lastly, logs usually follow the changes in the source code of applications, so they contain the historical aspect of software evolution. All of that makes application logs a rich source of data for various analyses. Because of our focus on automation of knowledge discovery about systems, we are only interested in automated log analysis. According to [1], automated analysis of application logs is a widely studied discipline with growing interest among researchers in recent years. The most obvious usage of log analysis lies in the *operations* area (*intrusion detection*, *monitoring*) but its potential in reasoning about the *business domain* or the *design* of the software is also explored.

In this paper, we try to derive knowledge about enterprise systems (specifically about interactions between applications) from application logs. Our work fits in the *design/component dependency inference* area of the landscape of automated log analysis proposed by the authors of [1]. We perform experiments on a real-life system from the banking industry, hosted at Nordea Bank. We contribute to the body of knowledge in the following ways:

- we perform statistical analysis of log files of a subset of logs from a real-life system deployed at Nordea Bank,
- we formulate some general observations about the way logs are typically created by developers,

This work was supported by Gdańsk University of Technology

- based on the defined observations, we propose a new method for log template extraction called SLT (Simple Log Template),
- we propose a workflow for automated discovery of application interactions from their logs,
- we verify our method with logs from a real-life system deployed at Nordea Bank.

The remainder of this paper is organized as follows. Section II discusses related work. In Section III, we present a formal definition of the problem. In Section IV, we perform a statistical analysis of application logs from a subset of applications deployed at Nordea Bank and we formulate observations related to how developers place their log statements in enterprise systems. In Section V, we describe our approach for component dependency inference based on log analysis, while in Section VI, we evaluate this method against the real-life system deployed at Nordea Bank. Section VII presents conclusions from our experiments and outlines the future work.

II. Related Work

Component dependency inference using automated log analysis is rather a niche topic, but some notable recent work is worth mentioning. [2] analyzes web service interactions and tries to correlate web service invocations using IP addresses and invocation statuses found in logs. The authors identify two specific types of interactions: composition (one service orchestrates a series of calls to other services) and substitution (service is called as part of an error-handling scenario after a failed call to another service). The authors assume the availability of IP address information in service logs, which (according to [1]) is true mostly for access logs and network logs that may not be available for applications other than web services. Additionally, basing the analysis on IP address correlation may be very challenging in cloud-hosted applications, where services are replicated, and IP addresses can change dynamically. In contrast, our method puts minimum assumptions on log content and bases the analysis on log messages. The authors of [3] perform a statistical analysis of web service logs to identify a correlation between web services. The analysis includes time correlation, call frequency correlation, and analysis of service response times. The authors define three types of web service interactions: dependency on the data source (multiple web services try to access the same shared resource), hierarchical dependency (one service is invoked by another), and serial execution dependency (one service orchestrates a series of invocations). Similarly, as in the previously mentioned work, the focus is on web services and other types of applications are not considered. [4] describes a Bayesian Decision Theory-based approach to the identification of component dependencies. The authors describe each log message with a key and a set of parameters which are determined by some empirical knowledge and common string extraction. The authors also identify some observations related to logging practices that form the foundation for their algorithm: co-occurrence observation (logs of dependent services are time-correlated) and correspondence observation (logs of dependent services often contain some identical parameter). The proposed method is validated using the Hadoop dataset. In our work, we take these ideas further by removing any empirical knowledge needed to extract parameters. We also empirically confirm and further extend the authors' observations regarding logging practices based on application logs from a real-life system at Nordea Bank. Furthermore, we validate our method using a dataset of logs from Nordea Bank, which is expected to be less homogeneous than logs of any shared service platform like Hadoop.

Workflow discovery is a similar research area that aims in recovering whole processes from logs. Although it is not in the scope of this paper, workflow discovery is part of our future work and therefore notable work on this related topic is worth mentioning. [8] uses a process mining approach to discover recursive processes from event logs. The authors of [9] propose a method for triaging production failures that analyses service interactions to identify the failed workflows. The authors of [10] present a method for recovery of Communicating Finite State Machine model that represents service interactions. The proposed approach requires, however, users to input knowledge about the structure of a log file for it to be able to be processed.

Our work also aims to identify real-life logging practices applied by software developers. Similar efforts were presented in [5] where different categories of logging statements used by developers are defined. The authors take the source-code perspective analyzing logging practices from the point of view of a single application. [6] presents a statistical analysis of logging practices in open-source projects and [7] repeats this study for java-based open-source projects. The authors analyze factors such as log density, how meaningful log extracts are for bug-fixing, what are the typical changes to logging code, and how often they occur. In our work, we use logs from a real-life system at Nordea Bank to identify higher-level observations regarding logging statements that represent the intent to specifically track interactions between different applications and thus are useful for analyzing application dependencies.

Log template generation is another area of research, related strictly to log analysis, which aims in discovering templates that describe individual lines in log files. Being able to identify static and variable parts of log entries allows for better reasoning about the log content and is considered the basis of any log analysis task. [14] performs a comparative analysis of various log template generation algorithms. LogCluster [11] and DRAIN [12] are two of these algorithms that, according to [14], present respectively the lowest accuracy span and the top accuracy levels over the sample data sets. We picked these algorithms as a benchmark for the method proposed by us.

III. Problem statement

We aim at inferring knowledge about enterprise systems from application logs. The goal is to provide enterprise architects with a good enough representation of the system that reflects the reality of the production environment. The derived model of the enterprise system needs to support architects' activities related to reasoning about the architecture. An example of such activities is data governance which concentrates on aspects like usage of proper data sources by applications, understanding the semantic relationships between data stored and exchanged between applications, or ensuring assumed data flow. Apart from data governance, common enterprise architects' activities concentrate also on ensuring that processes implemented inside the system fulfill both internal and external regulations. These may include measuring process performance or ensuring certain regulatory constraints are obeyed.

We can assume that the set of applications in the enterprise system is known with a high level of confidence. On the other hand, knowledge of application inter-connections (interactions between applications) from the enterprise perspective is where the confidence decreases. Having hundreds of applications deployed in a bank, this confidence is only as good as the diligence of teams in updating a common architecture repository. Furthermore, basing decisions related to the architecture of an enterprise on human declarations, rather than facts, can lead to false conclusions and not-optimal choices. We consider the problem of increasing the confidence of knowledge about the actual application interactions as the core problem in architecture reverse-engineering.

Let $G(S)=(A,C)$ be an undirected graph representing a real system S, where A is a set of applications constituting S and C is a set of edges representing interactions between the applications. We say that applications A_1 and A_2 are interacting with each other if some data exchange takes place between them. Let $L(S)=\{l_1,l_2,\dots,l_n\}$ be a log of activities collected within system S consisting of n messages. We describe each line of the log by a tuple (t,a,m) where t denotes the date and time of log message creation, a represents the application that created the message, m is the actual log message text.

We define the problem of application interaction discovery from logs as finding an approximate graph $G'(S)=(A,C')$, where C' is an approximation of C, based on the system's log $L(S)$. $G'(S)$ is a graph representing an approximation of system S.

The presented problem definition is extendable and allows inference of other architecture properties on top of the applications' interaction graph. For any property of enterprise system $P(S)$, the problem of architecture property discovery from logs can be defined as finding a function F, such that $P'(S)=F(G'(S),L(S))$ approximates $P(S)$. In this paper, we do not cover solutions to the extensions of the core problems, leaving this for further work.

IV. Nordea Bank Data Set

Our work presents an experience of applying log analysis to one of the systems at Nordea Bank, which we will further refer to as NDEASYS. For our experiment and proposed method to be as generally applicable as possible, we picked an application with a relatively big integration part, such that logs created by interacting applications are the most representative. We applied the following criteria:

- team diversity – applications built by teams of different sizes, experiences, locations,
- application diversity – we included both dedicated business applications and shared service platforms (e.g., storage or communication services),
- time diversity – applications built in different periods,
- integration diversity – applications communicating using different interfaces and exchange formats.

Nordea Bank does not enforce any strict, centralized rules on how applications should create their logs (for non-regulatory logging), which additionally removes any accidental correlation between logs because of applications being created in the same company.

The NDEASYS1 dataset consists of logs of six applications whose properties are summarized in Table I. The logs were collected over 12 hours of operation on a random business day in an integrated test environment.

We distinguish between three types of applications that output logs in our data set:

- dedicated – application implementing logic specific to a single business domain,
- technical – application with a minimum amount of business logic, usually providing a support function, e.g., data or interface adaptation,
- shared service – application deployed on a shared platform, following a typical workflow for the platform.

We characterized team diversity by the number of developers and number of locations, they were working from. Time diversity was represented by the development period and the duration of application development. The diversity of integration was characterized by the integration styles used by each application (messaging or remote procedure invocation) and message formats used to exchange data with other applications.

We performed an initial analysis of the NDEASYS1 dataset. Fig. 1 presents a histogram of tokens that are present in the logs of each application. The histogram was created using 1000 bins representing the frequency of token occurrence in a log. We made the following observations with regards to all log files:

- in all the logs there is a clear split between a few very common tokens and a lot of very rare tokens (the histogram is right-skewed, see Fig. 1),
- at integration points, application input is commonly logged,
- shared services tend to log only inputs while dedicated and technical applications – both inputs and outputs.

TABLE I.
CHARACTERISTICS OF THE NDEASYS1 DATASET

App	Log size [MB]	Application diversity	Team diversity		Time diversity		Integration diversity	
		Type	Size	No locations	Dev. period	Dev. duration [months]	Integration style	Format
A	730	dedicated	3	2	2020-2022	24	Messaging, RPI	Swift (ISO15022, ISO 20022), JSON
B	40	technical	1	2	2020	1	Messaging	Swift (ISO15022)
C	100	shared service	2	2	2016-2020	48	RPI	JSON
D	0.2	technical	1	2	2020	6	Messaging	Swift (ISO15022)
E	1600	shared service	3	2	2016-2022	72	RPI	JSON
F	3	shared service	3	2	2016-2022	72	RPI	JSON

We interpret these observations from the point of view of the primary reason for logging, which is failure diagnosis. In the case of application integration, a popular practice is to log identifiers of data exchanged between applications. These identifiers are (to large extent) unique values, which explains the big number of very rare tokens appearing in the log as compared to the few frequent tokens that represent the static part of log messages.

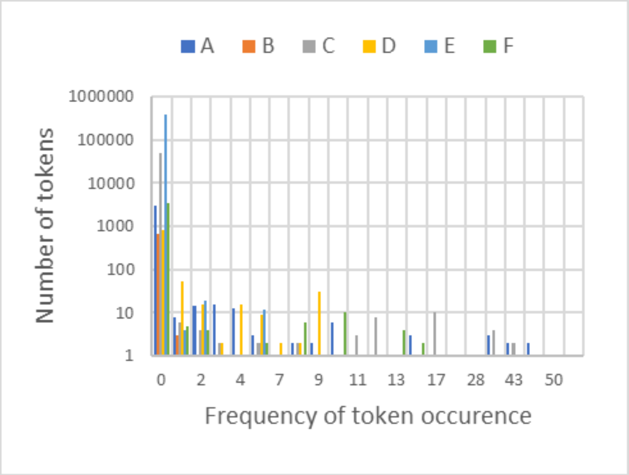


Fig 1. The histogram of token distribution. The horizontal axis represents 1000 bins showing the frequency of token occurrence in each log. The vertical axis is presented on a logarithmic scale and its values describe the number of tokens in each bin.

V. PROPOSED METHOD

Taking the observations presented in Section IV, we propose a method for detecting application interactions from logs leveraging the concept of rare and frequent tokens appearing in log files. Fig. 2. describes the proposed workflow and subsequent sections describe its steps in detail.

A. Log preprocessing

For log files from each application, we perform a minimal level of log preprocessing, which ensures a common view of each log. We introduce minimal assumptions for the content of log files. Each line should contain a *timestamp* and *message* attributes. Additionally, the *source* (the application that created a given line in the log) is determined based on which application the log file belongs to. The process of extraction of the minimum set of information from logs is called log formatting and an example of its output is shown in Fig. 3. In the case of some logs, we also unify data encoding to ensure that logs are comparable between applications. In this step, we perform preprocessing of the *message* attribute by splitting it into individual tokens using a regular expression [:.A-Za-z0-9_-]+. Apart from log formatting, we do not apply any application-specific logic requiring expert knowledge. As a result of the log preprocessing step, each log line is described by attributes: *source*, *timestamp*, and *set of tokens*. Example tokens are shown in Table II.

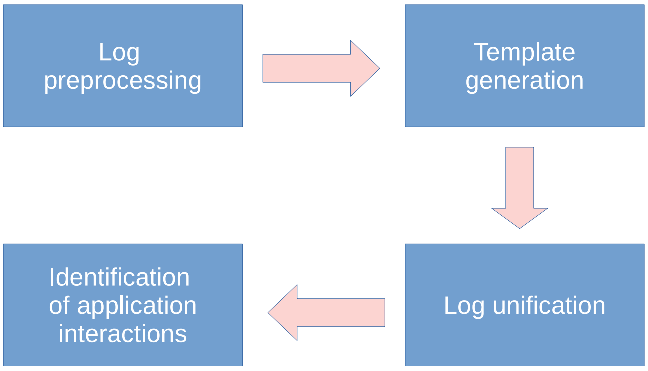
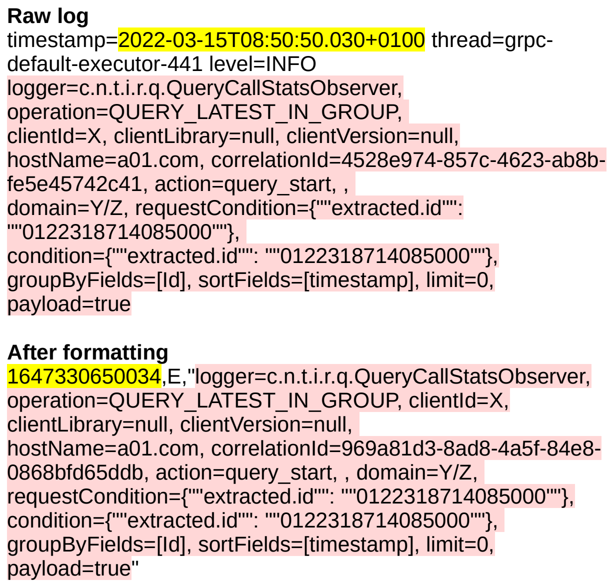


Fig 2. The workflow of log analysis for identification of application interactions.



Raw log

```
timestamp=2022-03-15T08:50:50.030+0100 thread=grpc-
default-executor-441 level=INFO
logger=c.n.t.i.r.q.QueryCallStatsObserver,
operation=QUERY_LATEST_IN_GROUP,
clientId=X, clientLibrary=null, clientVersion=null,
hostName=a01.com, correlationId=4528e974-857c-4623-ab8b-
fe5e45742c41, action=query_start, ,
domain=Y/Z, requestCondition={""extracted.id"":
""0122318714085000""},
condition={""extracted.id"": ""0122318714085000""},
groupByFields=[Id], sortFields=[timestamp], limit=0,
payload=true
```

After formatting

```
1647330650034,E,"logger=c.n.t.i.r.q.QueryCallStatsObserver,
operation=QUERY_LATEST_IN_GROUP, clientId=X,
clientLibrary=null, clientVersion=null,
hostName=a01.com, correlationId=969a81d3-8ad8-4a5f-84e8-
0868bfd65ddb, action=query_start, , domain=Y/Z,
requestCondition={""extracted.id"": ""0122318714085000""},
condition={""extracted.id"": ""0122318714085000""},
groupByFields=[Id], sortFields=[timestamp], limit=0,
payload=true"
```

Fig 3. The outcome of the log formatting phase. Colors denote the respective fragments in the raw and the formatted log.

B. *Template generation*

Template generation is the process of determining some sort of pattern for each log line, which distinguishes the static part of the message from the variable parts. Traditionally, this process aims at providing as precise template as possible, which describes the position of each variable element in a log message. We found such an approach not necessary and giving worse results than the relaxed approach proposed in this paper. A more detailed comparison with commonly used template generation methods is provided in Section V.C.

We introduce the SLT (Simple Log Template) method of template generation, which is a relaxed variant of the LogCluster approach described in [11]. As compared to the original method, we do not extract the exact pattern but rather focus on splitting the *message* into a *key* (which represents the static part) and *identifiers* (representing the variable part) while entirely disregarding the position of tokens in the *message*. Such an approach serves two purposes. Firstly, it minimizes the number of templates. Secondly, it is better suited for generating templates for log statements with a variable number of repeated tokens. A typical example is logging of service input/output values which are documents exchanged between applications. Such documents, usually expressed in the XML or JSON format, very often are built on top of data schema with repeated occurrences of data items. Moreover, XML or JSON documents cannot be treated as a regular text because, depending on the schema, the order of appearance of their elements can also be variable. In such cases, extraction of the exact pattern would result in a potentially big number of complex patterns being generated, depending on the data sample.

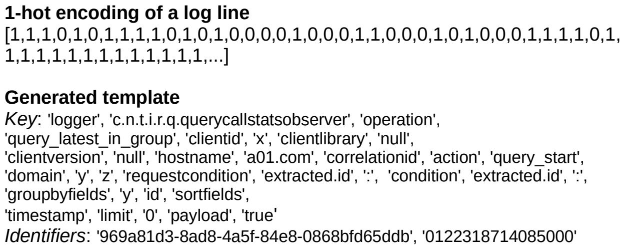
Our algorithm consists of two phases. In the first one, we cluster log lines using the *set of tokens*. We 1-hot encode [13] each token and then 1-hot encode each log line using the encoding of the tokens it contains and then run the K-means clustering algorithm. An example of token encoding is presented in Table II. The encoding of the sample log line from Fig. 3. is presented in Fig. 4. The optimal number of clusters is determined by using the silhouette method. The idea is, based on the observations in Section IV, that the same 1-hot encoded log lines would differ from one another only on a few positions, which should cause them to be assembled to the same cluster. The clustering process is performed separately for each *source*. This is to ensure that we do not find by accident common templates across different applications. Additionally, it reduces the clustering problem significantly.

TABLE II.
EXAMPLE OF 1-HOT ENCODING OF THE IDENTIFIED TOKENS

Identified token	Word index in dictionary	1-hot encoding
logger	0	[1, 0, ..., 0]
c.n.t.i.r.q.querycallstats observer	1	[0, 1, 0, …, 0]
operation	2	[0, 0, 1, 0, …, 0]
query_latest_in_group	38	[0, …, 0, 1, 0, …, 0]
clientid	4	[0, 0, 0, 0, 1, 0, …, 0]
x	39	[0, …, 0, 1, 0, …, 0]
clientlibrary	6	[0, 0, 0, 0, 0, 0, 1, 0, …, 0]

The outcome of the clustering phase is a set of clusters containing specific log lines for each *source*. Each cluster represents a separate logging statement (log template) and the lines that belong to the cluster – instances of that template. During the second phase, we process each cluster individually and extract *identifiers* from the *message* attribute. We count the frequency of each token appearance in the cluster. We then apply a frequency threshold – tokens appearing less frequently than the threshold are considered the *identifiers*. This is a direct utilization of the observation presented in Fig. 1. Both *key* and *identifiers* are represented as a set of tokens – their order is not considered. An example of such a template is presented in Fig. 4. Although this might result in different templates receiving the same key, we rarely found that to be a case in practice. Usually, logging statements are significantly different from one another which ensures the possibility of precisely locating such a logging statement in the source code of the application during failure diagnosis.

In the end, we combine the identifiers from all the clusters into a single set. The outcome of the template generation process is a mapping of log lines to clusters and a set of tokens that are considered identifiers in each application log.



1-hot encoding of a log line
[1,1,1,0,1,0,1,1,1,1,0,1,0,1,0,0,0,0,1,0,0,0,1,1,0,0,0,1,0,1,0,0,0,1,1,1,1,0,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,...]

Generated template
Key: 'logger', 'c.n.t.i.r.q.querycallstatsobserver', 'operation',
'query_latest_in_group', 'clientid', 'x', 'clientlibrary', 'null',
'clientversion', 'null', 'hostname', 'a01.com', 'correlationid', 'action', 'query_start',
'domain', 'y', 'z', 'requestcondition', 'extracted.id', ':', 'condition', 'extracted.id', ':',
'groupbyfields', 'y', 'id', 'sortfields',
'timestamp', 'limit', '0', 'payload', 'true'
Identifiers: '969a81d3-8ad8-4a5f-84e8-0868bfd65ddb', '0122318714085000'

Fig 4. Example outcome of a log line encoding and the generated template

C. *Log unification*

The goal of this step is to provide a unified view of the log across all applications. We merge all logs and order according to *timestamp*. For each log line, we identify the cluster it belongs to. The cluster identifier becomes a *key* of the log line. From the message attribute, we select only these tokens that are considered identifiers for a given cluster. These tokens form the *set of identifiers* for the log line.

D. *Identification of application interactions*

We identify interactions between applications by tracing the same identifiers appearing in different sources. This leverages the observation related to how developers log inputs and outputs to/from the applications they integrate with. We seek reoccurring identifiers within time windows. The size of the time window is expressed in the number of milliseconds and is configurable. That allows searching for both direct and indirect (distant) interactions between applications.

For a given time window size, we slide the window through the unified log with a configurable increment. Within a given time window, for each identifier, we record the applications (sources), for which it appears in the log line. In that way, we collect the sets of applications sharing the same identifier. We consider these sets as probable application interactions. This approach is continued while the time window slides through the log. We collect the number of occurrences of each interaction during that process.

In the end, we apply a threshold to filter out interactions that are not very common in the log. Choosing a proper threshold appears to be challenging as we both want to filter out the accidental correlation of identifiers and do not want to dispose of true but rare interactions. We noticed that usually the problem with the identification of accidental occurrences of identifiers is related to the common occurrence of short tokens which are falsely classified as identifiers during the template generation phase. An example of such a situation is when processing time is printed in a log as a numeric value. These values tend to be small numbers with a high probability of reoccurring in the log. As a countermeasure, we apply the token length criterion to determine what can be considered a valid identifier. We find this optimization simple but very effective in filtering out false-positively identified interactions.

We start the process of application interaction discovery using short time windows. This is intended to identify short, direct interactions in the first place. We then continually increase the time window size to identify distant interactions which can represent scenarios other than real-time processing (e.g., batches of data delivery, followed by batches of data processing).

VI. Method Evaluation

We evaluated our method using a real system deployed at Nordea Bank and a set of logs described as the NDEASYS1 dataset in Section IV. The subsequent sections describe in more detail the test environment and our approach to method evaluation.

A. *Experiment setup*

The system we used for evaluation consists of six applications, which were used to capture NDEASYS1 dataset. For the sake of anonymization, in this paper the applications are called with the letters A-F and this naming is consistent with Table I. Fig. 5 presents the high-level architecture of the system.

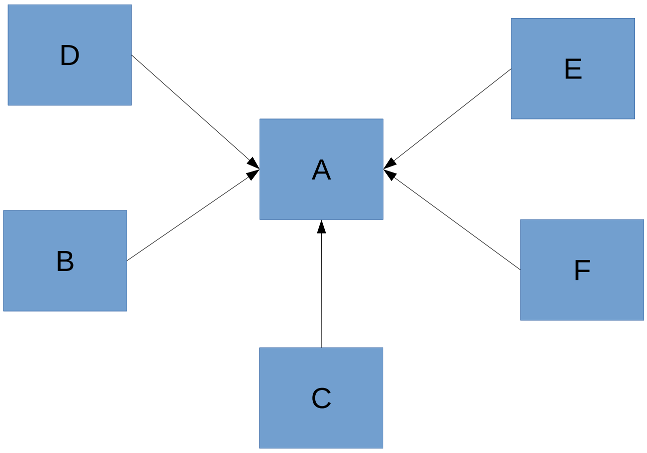


Fig 5. Architecture of the system used for evaluation. Lines denote pairs of applications interacting with one another, arrows denote the direction of data flow.

Applications B and D are responsible for data delivery. Application A is the main application in the system, which performs business logic and coordinates interactions between applications C, E, and F by fetching data from them. Applications C, E, and F are common services in the bank, exposed via a shared platform. Each of them exposes its domain data internally in Nordea Bank using standard interfaces.

The business processes of application A that are in the scope of this paper can be divided into three categories:

- data loading – an asynchronous process of delivering data to the system; once data is loaded, it is stored for further processing,
- data processing – a scheduled synchronous process occurring every 15 minutes that performs processing of previously delivered data,
- daily reporting – a scheduled synchronous process executed once per day which aggregates the processed data and delivers them to downstream applications.

Although this paper focuses on direct interactions between applications, awareness of the processes helps in configuring our algorithm to find interactions that appear in logs in distant lines.

B. Evaluation method

We use this knowledge about the system to validate our method. As a main measure of accuracy of our method we chose the F1 score and use it to compare the set of edges identified by our algorithm with the reference set of edges presented in Fig. 5. We do not consider the direction of edges.

C. Template generation algorithm performance

A part of our algorithm is a proposal of a new template generation system, focusing specifically on the detection of identifiers in log files. We evaluate our algorithm by comparing its efficiency to popular template generation algorithms: LogCluster [11] and DRAIN [12]. Table III presents the outcome of this comparison for different types of logs: the time of algorithm execution for each data set, the number of identified clusters, precision, recall and F1 score.

Since both LogCluster and DRAIN are more generic algorithms than ours, we need to define objective comparison criteria. For the sake of algorithm comparison, we decided to consider the number of identified templates, the F1 score of identifier detection and the speed of the algorithm. Both LogCluster and DRAIN produce a set of templates as an output. We unify these templates as regular expressions, where variable parts of each template are transformed into capturing groups. We then process each line of the log file with all the regular expressions and extract the tokens that were captured. LogCluster and DRAIN are not very consistent in what they consider as a token with our algorithm. To remediate that, we post-process each captured group with the same regular expression that is used in our algorithm for token identification. In the end, we apply the same length-based criterion to decide if a token is a valid identifier. The outcome of this post-processing is, for each log file and a set of templates, a list of identifiers found in the file. The list of identifiers is compared to the ground truth derived from the log dataset using our domain knowledge.

Since all the algorithms can be tuned with hyper-parameters, we measured a wide range of parameter values but for the brevity of presentation, we mention only the best score for each of the algorithms.

To extract the ground truth, for each log file, we collected the list of all tokens and the number of their occurrences. We then traversed the list of tokens from the most to the least frequent applying our domain knowledge to remove all tokens which were not valid identifiers. Part of this process was performed automatically. In this phase, we removed tokens based on their length or type (e.g., dates, IP addresses, or cash amounts were not considered valid identifiers, but are easy to filter out using regular expressions). In the second phase, we performed manual filtering by removing to-

TABLE III.
COMPARISON OF TEMPLATE GENERATION METHODS

Algorithm	Log	Time [s]	Clusters	Precision	Recall	F1 Score
DRAIN	A	95.0	31	0.96	1.0	0.98
	B	10.0	335	0.99	0.94	0.96
	C	39.0	6	0.99	1.0	0.99
	D	1.0	82	n/a	n/a	n/a
	E	342	97	1.0	0.86	0.92
	F	1.0	13	0.97	0.99	0.98
LogCluster	A	20.0	6	1.0	0.93	0.96
	B	1.0	28	2.0	0.018	0.03
	C	8.0	2	1.0	0.79	0.88
	D	0.1	3	0.66	1.0	0.79
	E	34.0	5	1.0	0.85	0.91
	F	0.1	3	1.0	0.95	0.97
SLT	A	185.0	17	0.95	1.0	0.97
	B	1.0	2	0.71	0.99	0.83
	C	46.0	19	0.94	0.99	0.97
	D	0.1	3	0.6	1.0	0.75
	E	380	19	1.0	1.0	1.0
	F	2.0	2	0.76	1.0	0.86

kens that were valid domain-related words. This process was performed for the most frequent tokens (with the number of occurrences higher than 1).

LogCluster is the fastest algorithm among the three. It outperforms the rest by an order of magnitude. It also notes the lowest F1 score, especially for log B. It tends to keep the precision very high with lower recall values. Further analysis of false positives returned by this algorithm shows that it often includes frequent tokens, which are business terms appearing in the log. LogCluster returns fewer and more generic clusters than others.

DRAIN shows the best overall results in terms of F1 score with by far the highest number of clusters returned. The high number of clusters is a result of two aspects: DRAIN not being designed for processing multi-line log entries (similar as LogCluster) and not coping well with log entries of variable length. While the first deficiency is easy to overcome, the second poses a challenge for general use for enterprise-wide log analysis. A typical case for log entries with variable content is logging system inputs, which often come in the form of XML/JSON documents with repeated elements. In such cases, DRAIN extracts each log entry with a given length to a separate template. For long log entries, templates are often very big and hard to process. This finding is in line with the conclusions from the real-life DRAIN application presented in [14].

The SLT approach proposed by us is not far from DRAIN in terms of F1 score, with similar timing performance but outputting the number of clusters that is much more on par with the reality. It copes well with log entries of variable length. One deficiency, that results in lowered precision rates, is falsely identifying rare numbers (e.g., cash amounts) as identifiers. With the approach we have taken, they cannot be distinguished from the true identifiers. This problem needs to be handled in the subsequent processing steps that utilize our algorithm. Overall, we think SLT is a reasonable all-around approach for identifying identifiers in log files of diverse format and content.

D. Results

We ran a series of experiments with our approach using different time windows. For each run of the algorithm, we collected the discovered application interactions, together with the set of identifiers that are found in the logs of both interacting applications within the time window. An example of such output is presented in Table IV, and the respective approximate graph G'(S) is shown in Fig. 6. In Table IV, falsely identified interactions are marked in red.

TABLE IV.
EXAMPLE OF DISCOVERED APPLICATION INTERACTIONS

Interaction	Number of occurrences	Example identifier
(E, A)	47828	5435ab2142314bsw
(C, A)	2156	43543612124
(F, A)	1314	2353518
(F, C)	874	200.0000
(D, A)	77	abhgswe0053
(D, E)	14	2022-01-21
(B, A)	7	basdewe2xyz

In some cases, it is hard to distinguish between an accidental correlation of token values and actual but rare interactions. For example, the (F, C) interaction has a higher number of occurrences than (D, A) but the discovered identifiers actually denote the number of records per second processed by services F and C. Such an accidental correlation of tokens representing data of low variety (e.g. small numbers or dates) is the main source of score deterioration in our method.

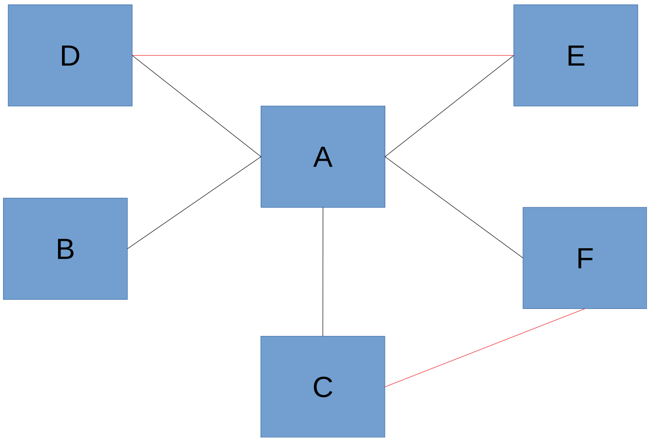


Fig 6. An example of graph G′ output that represents the approximation of the system, at a window size equal to 20 minutes. Edges marked with red represent falsely discovered application interactions.

We found that applying a proper identifier length criterion allows for maximizing the F1 score of the result. Table V shows the maximum scores achieved for each size of the time window. The maximum overall F1 score of our algorithm was 83%, reached for a time window of 20 minutes which was long enough to detect the interaction between the asynchronous data loading process and the scheduled data processing process. Longer time windows allow us to identify distant correlations at the cost of the decrease in precision (the longer the time window, the higher chance of accidental correlation occurring). At the current stage, our approach tends to discover short-time-distance interactions with high precision. The confidence of interaction detection decreases when searching for distant correlations. This is one of the aspects that are subject to improvement in our future work.

TABLE V.
RESULTS OF OUR METHOD FOR DIFFERENT WINDOW SIZES

Window size [ms]	Number of discovered interactions	Precision	Recall	F1 Score
200	3	1	0.6	0.75
1000	3	1	0.6	0.75

5000	4	0.75	0.6	0.67
10000	4	0.75	0.6	0.67
60000	5	0.8	0.8	0.8
1200000	7	0.71	1	0.83

VII. Conclusions

In this paper, we presented an approach to discovering the interactions between applications in enterprise systems. We validated our approach with a real-life system deployed at Nordea Bank. Our method could achieve the 83% F1 score of the identified interactions and is a good base for further extension. The biggest challenge is distinguishing rare, actual interactions from accidental data correlation, which we will address in our further study.

As part of our approach, we proposed the SLT method for discovering templates for log entries. We compared this method with other common approaches and found that it provides good-enough F1 score while being able to handle variable log entries, which is one of the main deficiencies of the other methods. We find SLT a good candidate for a template generation method in a general use case when we do not know the exact profile of the logs we are analyzing.

Working on a real-life system allowed us also to provide statistics and conclusions about logs from various types of applications. We found some common patterns of logging activities performed by developers when working with integrated systems which stem from the practical need of developers to be able to perform failure diagnosis. The main conclusion in this area is that logging data identifiers is a common practice in the industry. These observations allow introduction of simplifying assumptions for the general problem of discovering system properties from application logs and help better focus our future work on the problem.

Our future work will focus on two areas: 1) improvement of the current method and 2) working on its extensions to extract other properties of enterprise systems. We see increasing the precision of the identifier detection as the main improvement that would positively influence the F1 score of the overall method. This requires the development of a better way to check if an identifier is valid to decrease the level of accidental correlation of non-identifier tokens. Another area of focus is the performance of our method, improvement of which would allow for the analysis of larger log samples covering longer periods. That would open the possibility to identify very distant interactions (e.g., related to monthly-reporting processes), or validate the method using a larger system.

As for the method extensions, our goal is to discover the fragments of UML diagrams describing the working system. That requires providing means of discovering the system's properties such as:

- processes the system is running (control flow) – scenarios of interactions between applications with their frequency and timing,
- data flow – how data is passed across applications and what are their origins,
- semantic relationships between data – the mapping of data processed by different applications to find a common data dictionary (or even the data model).

We believe that such an overview of the running system would provide enterprise architects with the necessary tools to centrally validate various properties of the system and plan respective actions accordingly.

Acknowledgment

This paper was written in cooperation with the Nordea Bank, which provided the log dataset and an overview of the systems that were subject to this study.

References

[1] L. Korzeniowski and K. Goczyla, "Landscape of Automated Log Analysis: A Systematic Literature Review and Mapping Study," *IEEE Access*, vol. 10, pp. 21892–21913, 2022.

[2] H. Labbaci, B. Medjahed, and Y. Aklouf, "Learning interactions from web service logs," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 10439 LNCS, no. August, pp. 275–289, 2017.

[3] E. U. Aktas, M. C. Calpur, U. U. Yildirim, and E. Yıldırım, "Inferring dependencies among web services with predictive and statistical analysis of system logs," *CEUR Workshop Proc.*, vol. 2291, no. December, pp. 235–244, 2018.

[4] J. G. Lou, Q. Fu, Y. Wang, and J. Li, "Mining dependency in distributed systems through unstructured logs analysis," *Oper. Syst. Rev.*, vol. 44, no. 1, pp. 91–96, 2010.

[5] Q. Fu *et al.*, "Where do developers log? an empirical study on logging practices in industry," 2014, pp. 24–33.

[6] D. Yuan, S. Park and Y. Zhou, "Characterizing logging practices in open-source software," 2012 34th International Conference on Software Engineering (ICSE), 2012, pp. 102-112, doi: 10.1109/ICSE.2012.6227202.

[7] B. Chen and Z. M. (Jack) Jiang, "Characterizing logging practices in Java-based open source software projects – a replication study in Apache Software Foundation," *Empir. Softw. Eng.*, vol. 22, no. 1, pp. 330–374, Feb. 2017.

[8] M. Leemans, W. M. P. Van Der Aalst, and M. G. J. Van Den Brand, "Recursion aware modeling and discovery for hierarchical software event log analysis," *25th IEEE Int. Conf. Softw. Anal. Evol. Reengineering, SANER 2018 - Proc.*, vol. 2018-March, no. March, pp. 185–196, 2018.

[9] G. Qi, W. T. Tsai, W. Li, Z. Zhu, and Y. Luo, "A cloud-based triage log analysis and recovery framework," *Simul. Model. Pract. Theory*, vol. 77, no. August 2020, pp. 292–316, 2017.

[10] I. Beschastnikh, Y. Brun, M. D. Ernst, and A. Krishnamurthy, "Inferring models of concurrent systems from logs of their behavior with CSight," 2014, pp. 468–479.

[11] R. Vaarandi and M. Pihelgas, "LogCluster - A data clustering and pattern mining algorithm for event logs," *Proc. 11th Int. Conf. Netw. Serv. Manag. CNSM 2015*, pp. 1–7, 2015.

[12] P. He, J. Zhu, Z. Zheng, and M. R. Lyu, "Drain: An Online Log Parsing Approach with Fixed Depth Tree," *Proc. - 2017 IEEE 24th Int. Conf. Web Serv. ICWS 2017*, pp. 33–40, 2017.

[13] J. T. Hancock and T. M. Khoshgoftaar, "Survey on categorical data for neural networks," *J. Big Data*, vol. 7, no. 1, 2020.

[14] J. Zhu *et al.*, "Tools and Benchmarks for Automated Log Parsing," *Proc. - 2019 IEEE/ACM 41st Int. Conf. Softw. Eng. Softw. Eng. Pract. ICSE-SEIP 2019*, pp. 121–130, 2019.