FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Atte Jauhiainen

DATA-ORIENTED APPROACH FOR SYNTHETIC NETWORK TRAFFIC GENERATION

Master's Thesis
Degree Programme in Computer Science and Engineering
January 2022

Jauhiainen A. (2022) Data-Oriented Approach for Synthetic Network Traffic Generation. University of Oulu, Degree Programme in Computer Science and Engineering, 40 p.

ABSTRACT

A significant portion of dynamic websites, called web applications present dynamic information, which can be fetched from a backend pre-emptively or on-demand. For popular sites and applications, the increased volume of network calls and connectivity creates high performance requirements for the server side implementation. A failure to fulfill these requirements might lead to security vulnerabilities, high latency, logical malfunctions, and other issues.

Identifying and quantifying performance weaknesses is not trivial. Even if measurements are collected, they might not originate from a stable environment which allows truthful comparison between software versions. Often application programming interface (API) performance issues are found by accident while using the software, but a few more automated methods to find the issues have been created as well, such as automated end to end API tests.

In this diploma thesis, a case of web application software is studied with the goal of increasing its performance testing capability. A data-oriented approach to create synthetic network traffic for performance testing is prototyped. The approach consists of recording incoming network traffic at the server, finding user sequences within the traffic, which are then used to create a model of the users. The model can then be utilized to generate synthetic network requests, which have the same characteristics as real network traffic. The synthetic network requests are then used to generate load in a performance test environment. The approach is evaluated by its applicability for the use case.

Keywords: Performance testing, user modelling, application programming interface, API, cloud, latent dirichlet allocation

Jauhiainen A. (2022) Datapohjainen lähestymistapa verkkoliikenteen syntetisointiin. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 40 s.

TIIVISTELMÄ

Suuri osa dynaamisista verkkosivuista eli verkkosovelluksista esittää käyttäjälle dynaamisesti tietoa. Tarvittava tieto haetaan tyypillisesti palvelimelta (jatkossa "backend"), jonne voidaan myös lähettää dataa. Suosituilla verkkosovelluksilla voi olla valtava määrä samanaikaisia käyttäjiä, mikä aiheuttaa suuren kuorman backendille. Mahdollinen suuri kuorma johtaa korkeisiin suorituskykyvaatimuksiin. Epäonnistuminen suorituskykyvaatimusten täyttämisessä voi johtaa tietoturvaongelmiin, pitkiin vasteaikoihin, loogisiin virheisiin tai muihin ongelmiin.

Verkkopalvelun suorituskykyheikkousten löytäminen ja tunnistaminen ei ole triviaali tehtävä. Vaikka ohjelman suorituksesta kerättäisiin tietoa, se ei välttämättä pohjaudu vakaasta ja toistettavasta ympäristöstä mitatulle datalle. Usein backendien rajapintojen suorituskykyheikkoudet löytyvät vahingossa rajapintaa käytettäessä, mutta muutamia automatisoituja keinoja heikkousten löytämiseksi on olemassa.

Tässä diplomityössä tutkitaan yhden verkkosovelluksen käyttäjien datapohjaista mallinnusta, ensisijaisena tavoitteena nostaa sovelluksen suorituskykytestauksen tasoa. Työssä kokeillaan datalähtöistä lähestymistapaa verkkoliikenteen mallintamiseksi ja syntetisoimiseksi. Lähestymistapa koostuu verkkoliikenteen nauhoittamisesta, käyttäjäsekvenssien tunnistamisesta verkkoliikenteen joukosta ja käyttäjien mallintamisesta. Mallinnettujen käyttäjien avulla voidaan luoda synteettistä verkkoliikennettä, joka muistuttaa ominaisuuksiltaan alkuperäistä nauhoitettua verkkoliikennettä. Synteettistä liikennettä käytetään suorituskykytestaukseen tarvittavan kuorman luomisessa. Lopulta lähestymistavan onnistumista ja sovellettavuutta arvioidaan.

Avainsanat: suorituskykytestaus, käyttäjän mallinnus, sovellusrajapinta, API, pilvipalvelu, latent dirichlet allocation

TABLE OF CONTENTS

FOREWORD

This thesis was partly conducted while working for Uros Oy. It took a few months to find a suitable topic, but model-based network traffic synthesis eventually came to mind in late 2020. The idea of automating the complete process all the way to parameterization was quickly deemed too broad and the scope was narrowed to its current form. The spring of 2021 was the most intensive time of working with the thesis, as that is when the I completed the traffic recording and most of the data processing software. The pace of work gradually slowed as the thesis got closer to being complete but finally in the Christmas of 2021 the thesis was complete. I can, without a doubt, say that this was the most challenging and time consuming but also the most fulfilling endeavour I've completed during my studies.

This thesis would not be complete without the help of all the lovely people helping me on this journey. First and foremost I would like to thank my wife, Kia, for the seemingly endless support she gave me. I would also like to thank my thesis adviser Teemu Kanstren at Uros for his patience and guidance, and all other colleagues who helped me to make the right decisions. And finally, I thank my supervisor Aku Visuri for his valuable advice and insight.

Helsinki, January 3rd, 2022

Atte Jauhiainen

LIST OF ABBREVIATIONS AND SYMBOLS

API	application programming interface
LDA	latent Dirichlet allocation
AWS	Amazon Web Services
EC2	Elastic Compute Cloud
OSI	Open Systems Interconnected
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol (secure)
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
SOAP	Simple Object Access Protocol
MTU	maximum transmission unit
IP	Internet Protocol
VPC	virtual private cloud
SUT	system under test
HWPC	hardware performance counter
REST	representational state transfer
SVM	support vector machine
HMM	hidden Markov model
SAM	sequence alignment method
DNA	deoxyribonucleic acid
CI	continuous integration
KMS	Key Management Service
ALB	Application Load Balancer
HAProxy	High Availability Proxy
TLS	Transport Layer Security
IGW	Internet Gateway
VXLAN	Virtual Extensible LAN
LAN	local area network
URL	Uniform Resource Locator
K-S	Kolmogorov-Smirnov
F	cumulative distribution function
eCDF	empirical cumulative distribution function
n, m	size of dataset
D	maximum distance

1. INTRODUCTION

Various information systems such as web applications are used daily by people all around the world. As a business that provides these systems, it is important to ensure the systems are of high quality and work as intended. One common method of ensuring the quality is automated testing. There are many methods to test an information technology system, but it could be simplified as applying a set of predetermined scenarios to different parts of the system and then observing and comparing the results. Usually the goal of testing is to gather information about the quality and characteristics of different components and the system as a whole.

One field of testing is performance testing, which deals with the performance of some system. A company ordered this thesis for a web application, which is not thoroughly performance tested. The goal is to improve the testing capability for that application by creating realistic synthetic network traffic. However, it is not trivial to create credible synthetic network load.

In this thesis I investigate an approach to synthesize network requests using a network traffic log. Network request synthesizing is the process of creating fake requests, which simulate user behaviour. The goal of this approach is that the synthesized request chains would be statistically similar to the original traffic. In this thesis I describe and justify the design and implementation of tools to achieve this goal and evaluate their functionality.

The approach developed in this thesis consists of 4 distinct phases: network traffic collection, sequence analysis and modeling, request synthesizing, and finally applying the synthesized requests in performance testing. The effectiveness of this approach is then assessed in the evaluation phase. Sequence modeling is aided by a machine learning model called Latent Dirichlet Allocation (LDA). All the implementation tasks such as modeling, request synthesis and performance testing is done through Python scripting.

I have two research questions I try to answer in this thesis:

- **RQ1**: How effectively can Latent Dirichlet Allocation (LDA) be used to categorise user sequences in network traffic?
- **RQ2:** How effective are synthetic requests in conducting controlled performance tests for a back end application?

2. BACKGROUND AND RELATED WORK

Web applications, network traffic, application testing, performance testing and sequence modelling are the key aspects of this thesis. In the background section I talk about web applications, software testing and sequence modeling.

2.1. Web Application

Web applications and web services are distributed computer systems which are built to provide some functionality, such as providing access to a set of remote resources [1]. Web service is a client-agnostic interface and its infrastructure, whereas web applications consist of a web service and a web application which is usually used within some kind of a web browser. Web applications are usually quick to take into use, compared to desktop applications which might require more setup, such as installation and configuration.

2.1.1. Web Application Architecture

Web applications and other distributed systems often consist of multiple interconnected devices, i.e. servers, clients, and other machines. A large portion of systems required for work, financial services, social platforms, leisure, such as games, etc. rely on these services [2]. For the user, web applications may appear as a dynamic website or a mobile application, but under the hood there are many components that provide the functionality. A static website is not considered as a web application, unlike a website that displays dynamic data fetched from a remote source.

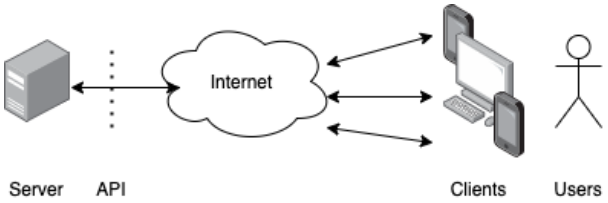

Figure 1. Simplified end to end web application.

A typical mobile or web application consists of a server and clients as shown in Figure 1. The client executes on the end users device such as a mobile phone or a web browser, whereas the server is executed on a remote device often described as a server or the cloud. Cloud platforms offer servers which are machines as any other, except they are managed by somebody else and paid per usage. Both the client and server components may have performance issues and highly benefit from testing in order to verify functionality of the integration and sufficient quality of operation. Client software may be atomic in the sense that it is not dependant on other clients. Unless we take peer-to-peer software into account, the client is often dependant on the server side software. The virtually unlimited number of possible simultaneous client devices and the client-server dependency results in a requirement to manage a highly scalable and difficult to predict network traffic volume.

2.1.2. Backend

Because of the strong coupling between clients and the backend, and the "single point of failure" nature of backends, their high availability is important. A backend that fails to respond might result in the whole application being unusable. Businesses with high availability services often prefer to host their applications in a cloud environment, such as the Amazon Web Services (AWS), Azure, Google Cloud Platform or some other vendor instead of having an on-premise hardware to run the server. These cloud environments offer tools to automatize deployments, backups, updates and other maintenance tasks.

Another important factor for cloud deployments is network delay. Each request has some delay between sending a request and getting a response. Larger cloud vendors provide multiple datacenter locations around the world, which allows a company to host multiple geographically distributed instances of their application. Usually the closer a server is to the client within the internet topology, the less significant is the network delay. Other important factors are scalability and the pricing model, as users only pays for usage and not the hardware. The high volume of computing power owned and sold by a cloud vendor allows the consumers to allocate the required resources and quickly ramp up computing heavy tasks without spending on hardware up front. In the case of web applications, the automatic scaling of the number of instance nodes is sometimes referred as a dynamic deployment [3].

In this thesis AWS virtual servers, called EC2 instances (Elastic Cloud Computing) play a key role. They are simple and cheap virtual servers that offer any kind of general purpose computing. The user has total control of the networking, processing and everything else within a virtual server. Virtual servers are a typical tool for hosting web applications and other software in a cloud environment. In this thesis EC2 instances are used for network traffic proxying and traffic monitoring. The preexisting Java-application which act as the network traffic source is also deployed in an EC2 instance.

2.1.3. Performance Considerations

Each instance of an application server has a limited number of API calls it can serve in a given time frame. This threshold is called the throughput of the server and is an important performance metric. If the number of concurrent users surges dramatically, a server will likely experience an unusually high load. If the number of requests is higher than the maximum throughput, unpredictable side effects may arise. Application throughput is a combination of the software itself and the environment it is running in. Environment consists of multiple aspects such as the hardware running the software, network connections and other software running within the same machine. Even the most performant software runs slowly in a terrible environment and vice versa.

In a centralized distributed system performance issues are typically less severe on the client software, but they can be observed regardless of the system wide workload. Figure 2 illustrates how in a simplified system the server side response time of an application is proportionally more significant when the number of concurrent users increases. The client side software performance remains constant for each user regardless of other users. However, the client side is still impacted negatively by

back end slowdown as response times may get longer. The system wide performance is the combination of all components where the lowest performing component is the limiting factor. Typically the components with high concurrency, throughput, number of connections, or heavy computing are the most prone to being the most critical components performance wise.

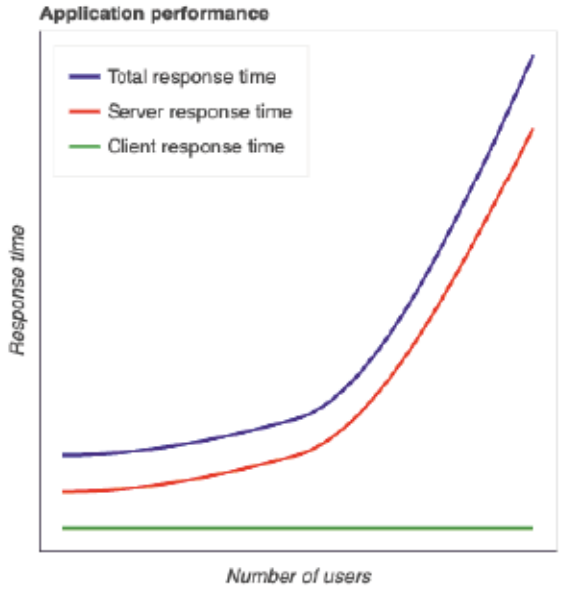

Figure 2. Client and server performance.

Lets consider a case where an end user registers a new account, logs in on the new account and then proceeds to purchase a new device from the store section of the application. If the server was targeted by a surge of simultaneous requests, the request to purchase the device could be timed out. This sort of performance issues could potentially be costly for the business as users might for example change the product because of such problems. Another performance issue example is a video streaming service, which might not be able to provide all viewers with consistent bit rates because of high number of concurrent watchers. This could be observed as stuttering, low video quality and buffering issues.

The performance of a piece of software can mean many things, such as memory usage or execution time relative to the input data or throughput. In the context of web applications performance is often measured in average response times (in e.g., milliseconds), number of requests or sessions within a time frame or request buffer length. All of these measurements can also be inspected relative to hardware utilization, where a higher physical resource usage per throughput is considered as worse efficiency. For example, if two pieces of software both serve one request per second but other reserves 50 per cent more memory, it most likely handles physical resources less efficiently. Software performance is interesting for the application in focus of this thesis as performance issues have arisen during usage and performance regressions have been introduced by accident. An automated performance testing routine would make it easier to find the issues as early as possible. Regular performance testing would also make it easier to discover performance limitations

of the software, and to draw conclusion about how the software and its environment handles current and future user base.

2.1.4. Web Traffic

Network monitoring is a method to inspect and collect network traffic, which is often used to gather some information of the system usage as a whole or some individual users. To create a network traffic monitoring setup, some background information about network traffic is required.

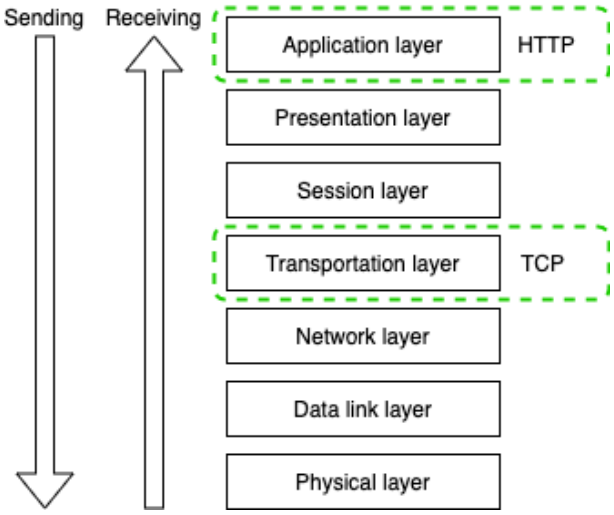

Figure 3. OSI (Open Systems Interconnected) model.

The connection between a server and a client typically uses some sort of application layer protocol on top of a transportation layer protocol, such as TCP (transmission control protocol) or UDP (user datagram protocol). Figure 3 shows how the application layer and transportation layer position within the OSI model (Open Systems Interconnection model). Commonly used application protocols are HTTP(S) (hypertext transfer protocol (secure)), WebSocket and SOAP (simple object access protocol) to name a few. The client and server are tightly coupled through a network connection and each resource access requires a network request. OSI model in Figure 3 describes internet architecture on a protocol level. It describes the layered nature of internet traffic in which application layer protocols build on top transport layer protocols.

The network traffic I need to have as an input for sequence modeling is HTTP requests. The tools I use to record the traffic output TCP packets which transport HTTP requests. Each HTTP request may be split into multiple TCP packets if the HTTP request is larger than the MTU (maximum transmission unit) and will not fit into one packet. This is shown in figure 4 and is called IP fragmentation which must be taken into account when working with a raw TCP dump.

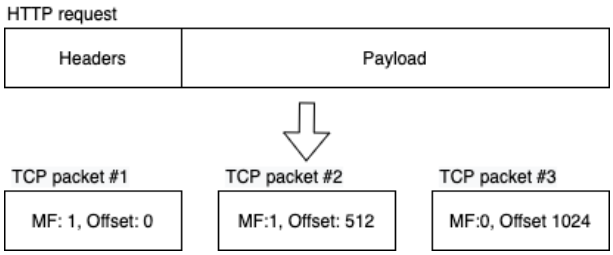

Figure 4. IP fragmentation: a HTTP request split into multiple TCP packets.

HTTP requests typically contain headers and metadata that allows the client and server to distinguish entities from each other. For example, authorization tokens, user agents, source IP addresses, or some other combination of these identifiers may provide the identifying information. With the aid of this identifying information one may rearrange a network traffic log to reveal all actions done by a single client. The user specific rearranged section of the log will be called a used sequence. A user sequence is thus a chronological list of API operations performed by a single entity.

A network traffic dump is one low level choice for analyzing application usage. The good thing about a network traffic dump is that it should show the requests as they are without any modifications. Multiple methods for implementing data capture exist. For example, one may increase logging coverage at the application level to also log individual requests. This could have a significant probe effect, as the logging will have an effect on the application performance [4]. Probe effect is a phenomenon where the process of data collection has an effect on the measured attribute. For example, in this case the increased logging could degrade performance. Logging could also be implemented at some level in the network infrastructure, but this could have same problems as application level logging. In the scope of this thesis a good solution is AWS VPC mirroring, as it is easy to apply on our existing applications. VPC Mirroring is cloud network tool specifically in the AWS cloud, that allows cloning traffic from one network interface to another with little effect in performance. VPC mirroring may have an effect on the traffic, but it should be negligible compared to application level logging [5].

A data set consisting of network traffic will usually not be very useful unless it is processed further. The chapter 2.3 focuses in how network traffic dumps consisting of network requests can be processed to create models or to find information. Finding information about the users of a system can also be helpful in developing or testing a system.

2.2. Testing

Testing is a collection of processes applied to a piece of software or a system, with the goal of finding bugs, weaknesses, and vulnerabilities in the system - and to verify that the software implements the desired functionality. Software testing can be conducted on many levels from a *unit test* to an end-to-end *integration test*. The scope of a unit test should be an atomic function that can be assessed individually, where as the end-to-end test might test one use case of the entire system - spanning over different components. Different test methods address different kind of threats and operate in a

different scope [6]. These different levels of testing are illustrated in figure 5, which portrays a linear model for application development. Performance testing would fit under *System testing* or *Acceptance testing*.

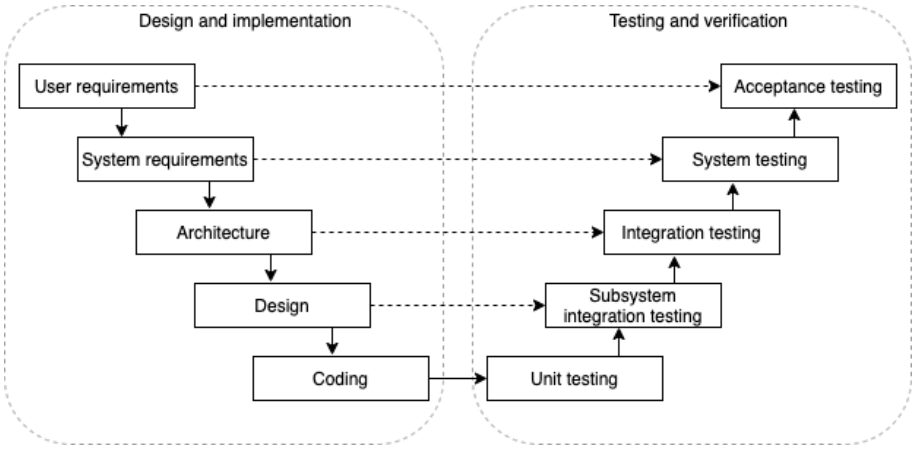

Figure 5. Life cycle of an application.

Different types of bugs are typically introduced to the code base when any changes are made, which warrants testing for every version of the system. This iterative degrading of system quality is called a regression. Software regression can be functional, which means that some output of the software is wrong in a newer version of the software. It can also be a performance regression which means that the software performance is worse after a change. Typical regression test suites try to keep the system quality on the same level as it was by identifying software regressions in existing features from newer versions of software. The ability to identify and solve the root cause causing the issue is often the goal of testing efforts. Regression testing gives tools to help this process in regard of regression issues. [7]

2.2.1. Performance Testing

Performance testing is a non-functional testing paradigm. It means that performance testing is not concerned with the functional correctness of the software, but rather about its performance characteristics. Often the goal for performance testing an application is to find performance regressions, which means finding the change in performance to previous software versions. Performance testing is also important for new features, which are not within the scope of regression resting yet. Performance testing, or performance benchmarking, could also give deeper insight about how the software functions under strain. Performance testing is also a valuable tool in finding the load thresholds at which the software performs badly. Performance, as mentioned in section 2.1.3, can mean many things based on the application. In the case of performance testing a web application, it is usually execution times or concurrency limits.

Web application performance can be observed by generating load against the application in a specific environment. The load should represent actual production application load, but it could be difficult to model production-like load reliably. It has been shown that production environment traffic can be utilized to model the synthetic load [8].

To make any assumptions about the performance of a software, some metrics are required. The metrics can be measured either from within the machine running the software under test, or from an external device, such as a web client. Performance telemetry can be collected from hardware within the machine running the system, e.g. from a processor load, cache hits and misses, or disk usage. External software would experience the performance as request response times. These times do, however, include the delays that are induced from the connection between the system under test (SUT) and the tester device. Modern hardware components; processors, motherboards, memory modules and other components provide a method to inspect their performance. These are called hardware performance counter (HWPC), and they can be fine tuned to track the system performance very accurately [9]. In the scope of this paper the performance of the SUT is viewed in the perspective of the client.

Web application performance testing can be divided into different types of performance testing plans, which are: stress testing, spike testing, load testing and soak testing [10]. Stress testing may find the performance limits of the system, as stress testing is about creating extreme loads. Spike testing is about finding how positive surges in load, known as spikes, affect the performance. A weak system will fail when a drastic change in volume is introduced. Load testing is the most "natural" performance test for an application, as it is about verifying how the software behaves under an expected load, such as the production environment. Load testing should be the easiest testing plan for performant software, but it can be difficult to model and reproduce a real-world load in the testing environment. Finally, the soak test is about finding the issues relating to prolonged use of the software. Difficult to identify issues such as memory leaks may come up when any piece of software is run for long time. Other testing plans exist, such as configuration testing or internet testing but they are not within the scope of this paper. [11, 10]

A test environment and a synthetic load is required to test for the web application performance. To generate predictable load some knowledge of the actual use cases of the software are required. This could be a formal description such as some software specifications but those might not exist or be available. It can be difficult to model the user sequences, but even more difficult is to pull these sequences without any data about actual traffic. By using production environment traffic data to create the models, one aims to achieve a strong relation between modeled users and production environment network traffic. This similarity would mean that the synthetic load is very similar to that of the production environment. This similarity can be evaluated by comparing the properties of the synthetic load to the load in the production environment.

One major difficulty in testing software performance is the imitation of production environment in all aspects, such as database state, usage profile and network requests. Also sometimes the environment replication is difficult because the application is deployed dynamically and the load is very high. In these extreme cases it makes sense to scale down the performance test setup from the production setup. A scaled down performance test setup would have cheaper and less performant hardware but also a smaller load. In the scaled down version it is easier and cheaper to isolate performance metrics and their correspondence to the software. The scaled down test environment is not a silver bullet though, as some new problems could arise as the test environment differentiates from the production environment. One particular problem is that some

of the performance issues could be hidden altogether as some components such as autoscaling have different configurations. The actual test environment should be built with a lot of thought and care.

Other difficulty in replicating the production environment is privacy. The production environment will almost always contain information of actual users, which is subject to privacy legislation and is not available for testing purposes. When replicating production network load, a one way obfuscation process could be applied to anonymize any identifiers, thus allowing us to utilize the relationships and characteristics of the production data set, while anonymizing the actual data. This method of replicating production data can be used to create a test environment that represents the production environment. Testing should not be done in the actual production environment, although production environment is ultimately the environment of which we're interested in. [12]

Network traffic volume is not the only predictor for application performance, but it is what this thesis focuses in. Client and server are not the only components to be considered in a distributed application, as there is usually some hosting infrastructure, such as routers, proxies, load balancers, databases and other devices and software utilized in conjunction with the client and the server. For testing purposes it makes reasonable sense to isolate a component and inspect its performance individually so that the number of measured variables can be reduced and the root causes are easier to identify. Although, both end-to-end and single-component performance measurements can be useful to benchmark the system as a whole. Typically both testing approaches are utilized to pinpoint the reasons that are causing issues. [11]

Often the coupling between a server and a client is defined and implemented in an application programming interface (API). API is something that the backend describes and implements as can be seen in figure 1. Typically an API is accessed with HTTP requests, but alternatives exist too. The most common web application API is a so called Representational State Transfer (REST), which is a set of methods to access and modify hierarchical resources [13]. Through API design one may choose to expose only a very specific set of resources and their actions per user. This granularity of access control allows the developers to cater for different kind of needs and users with just a single API. The wide range of uses, users and clients could make the API more complicated, which in turn makes it more difficult to simulate client behaviour. In this thesis I focus on the backend part of web application.

2.2.2. Test Definition

Methodologies exist for defining a single performance test. The formalized nature of a test definition helps in producing a replicable environment for the test, and results too. The test case could be split into 7 steps which are:

1. Definition of SUT and the environment.
2. Definition of test parameters.
3. Definition of failure thresholds.
4. Deployment of SUT.
5. Initialization of SUT (like database, network, etc.)

6. Conducting the test, collecting performance data.
7. Result analysis & visualization.

The *definition of SUT* is essentially the system and its version that are to be tested and the environment it is run in during the test. *Test parameters* are the scenarios the SUT is exposed to, such as the number concurrent simulated users using the service and their tasks. The definition of *failure thresholds* goes hand in hand with the test parameters; what sort of behaviour is acceptable within the parameters of the test. *The deployment* and *initialization of the SUT* are also closely related, as they describe how the SUT should be executed in the test environment and what sort of state should the whole system be in to start the test. This could include steps such initialization of persistent storage, in memory storage and the application state. It also includes the configuration of the test environment, such as network devices, firewall, remote access control and the physical hardware that the software runs on. Finally, when the configuration is all done, the parameterized *test can be executed* on the well defined and initialized SUT that is deployed in the desired environment. As the test is executed, different metrics are collected, which are then automatically aggregated, *analyzed and visualized*. [14]

One of the more challenging aspects in creating a good performance test definition is finding the right *test parameters*. In the case of web application backends the number of different possible types of user a system could have is virtually endless. One way of solving this issue and finding the most crucial sequences of user requests that can be used as an input for the test is sequence modeling which is introduced in section 2.3.

2.3. Sequence Modeling

Sequence modeling is about manipulating a set of sequences into a model that enables information extraction or generation. In its core sequence modeling is about clustering, abstraction, and model synthesis. However, it could also be a classification problem where the groups might not be known before the clustering. These models may then be utilized to synthesize new sequences that have same properties as the original set of sequences used to create the model. Describing user behaviour with a model and then using that description to generate network traffic is what sequence modeling is used for in this thesis.

If the sequence model describes behaviour of a user using a web application, the sequence can likely be modeled as a decision tree [15]. Decision tree is a model which represents all the actions a user can make as transitions and the states those actions lead to as nodes. A decision tree should cover all the states for the specific application. Decision tree models can be used in recommendation systems and other similar systems where user behaviour needs to be described [16]. The simplest of decision trees might be too simple model if the states can be cyclic. A cyclic state means that a transition returns the user to the same state as it was in before. The cyclic nature in a model could also be observed over multiple states which form a loop. In those cases, a directed cyclic graph could be more suitable. Both the decision tree and graph may also be viewed as Markov chains. A Markov chain is a probabilistic model which describes the relation of states by having a likelihood of transitions. For

example, one may say that "if it rains today, there is a 30 per cent chance it rains tomorrow, and a 70 per cent chance the sun will shine." In that model, the probability of a transition from state "it rains" to "it rains" would be 30 per cent and the transition from "it rains" to "sun is shining" is 70 per cent. A real world model would likely have many more states, but the idea stays the same.

Sequences of web traffic can be used as an input for testing a system that handles network traffic. In this thesis a sequence of web traffic is useful as a method of simulating client behaviour, but the traffic could simulate other sources of traffic as well. Testing which requires actual network requests is often some system level end-to-end test and in the scope of this thesis it is performance testing.

Specification mining is a type of process mining which aims to infer likely specifications by observing system behaviour [17]. Network data can be used as input data for specification mining processes to extract system information. Process mining a web log has been used to find probabilities for the users next possible clicks in website navigation [18]. Intrusion detection within various control systems such as home automation systems is one field where specification mining is applicable. Other field is network forensics in general, which includes all networked appliances. Specification mining can be used in identifying malicious users, by using a host-based algorithm to detect connection chains [19]. Network traffic in a control system is quite predictable so anomalies stand out. In the case of home automation systems intrusion anomalies are often unauthorized resource accesses. [20]

There are more methods to classify sequences by their features, such as support vector machines (SVM), hidden markov models (HMM), and sequence distance model [21]. Hidden Markov model is an extension to the Markov model described earlier, in which the real states are not examinable. One may not view the state itself, but instead only some secondary phenomenon, i.e. how the state affects something else. The conversion of sequence of events to a vector that describes the sequence and its features, and then using those vectors to classify the original sequences shows promise in making the process less time consuming [22]. The difference between two distinct sequences can be found out for example by calculating the non-euclidean distance between two sequences with sequence alignment method (SAM) [23]. The sequences are more alike the closer they are, which could be useful information when modeling sequences.

Sequence classification has also been used extensively in other fields, such as classifying proteins [24, 25] and categorizing DNA sequences [26]. Process mining, which closely resembles sequence classification has been used extensively in business processes [27], which typically have complicated states and transitions between them. A couple of prominent free tools exist for analyzing processes and process mining, such as ProM [28].

3. IMPLEMENTATION

The purpose of this thesis is to enhance performance regression testing capability by providing a tool to synthesize parameterizable network load. This web application backend has been in the focus of performance testing earlier, but the performance testing environment had trouble in imitating production load for the SUT. The target of the network traffic which I am trying to imitate is the SUT. It is a Java backend which serves different kinds of clients and applications. This synthetic load generated in the process can be utilized in a performance testing setup, where the backend is put under stress to reveal its performance characteristics.

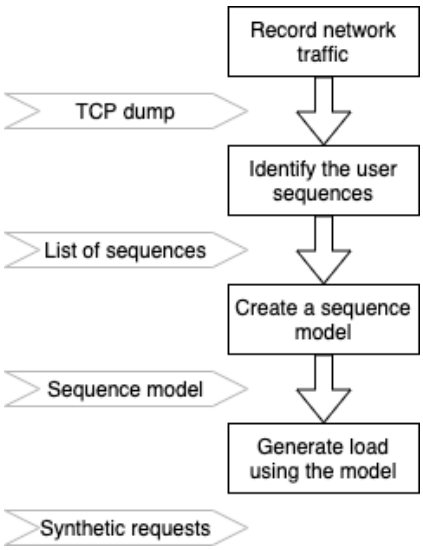

Figure 6. The 4-step process used in the thesis to synthesize network traffic. The text on the left describes the inputs and outputs of the phases.

This thesis consists of four sequential steps which are illustrated in Figure 6 and split into detail below:

1. **Recording:** In the first step web application network traffic is recorded.
2. **Sequencing:** Network traffic is then parsed in the second step. Parsing the TCP traffic outputs HTTP requests with its metadata. HTTP requests are then grouped into sequences, where each sequence contains requests by one user.
3. **Modeling:** In the third step the user sequences are analyzed statistically and a model is created to represent groups of user sequences. This is a clustering method, where action similarity is used to measure similarity for grouping. Each modeled group is then identified as a type of user, and a typical user sequence within that group is parameterized to represent the group as a whole. Parameterization is the process of defining parameters for every request within the sequence.
4. **Synthetic traffic generation:** The fourth step is about creating synthetic requests using the parameterized user sequences from step 3. The synthetic requests can then be used to create load, for example, in a performance testing environment.

This implementation and the 4 steps should not be completely unique to this web application. The approach I am using to model traffic should be usable for all web

applications. There are many application specific implementation details though which decrease code portability. Step 1 might be the most specific, as the network data collection is unique to AWS and will not work for other cloud solutions. The method of collecting data should apply for all use cases though. Step 4 is also quite application specific, as in that step the modeled users are parameterized to match the API so they can be used for synthesizing network traffic. On a generic level this approach should be applicable to virtually any web application. However, some application specific tuning is required for the approach to produce meaningful output.

3.1. Traffic Recording

The environment in which network traffic is recorded is a staging environment. Staging environment is identical to production environment infrastructure and code wise, but the user base is different. Production environment handles paying customers, where as staging serves developers, internal testers, sales and other similar personnel. Using staging environment is sufficient for implementing and demonstrating the process used in the thesis, but is obviously a bad choice for analyzing customers. In this sense, the recorded traffic describes the behaviour of developers, testers and so forth.

The first step is about recording inbound network traffic coming to a web application backend. Network traffic in this case consists of individual HTTP requests, which are sent by individual clients and rerouted through a proxy to the backend, as seen in Figure 7. There are other components within the infrastructure, but they are left out for the sake of clarity. The content of the HTTP packets is REST API requests. The network traffic is captured within a predetermined time frame, which is a week in our case. A weeks worth of requests is approximately 300 000 requests in a 750 megabyte packet capture file.

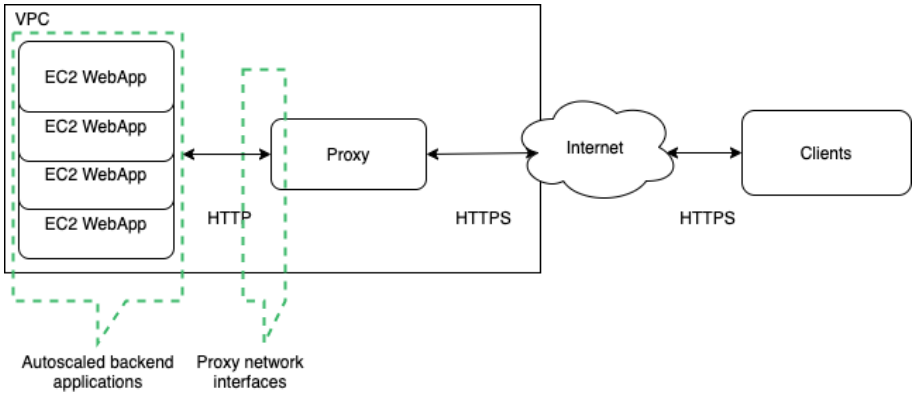

Figure 7. Existing application architecture.

The system under test which is the component in focus, is a Java-based web application backend. As it is developed further, changes to code happen daily and new versions are deployed on a weekly basis. Each of the new versions go through automated continuous integration (CI) pipelines which verify that no logical regressions are introduced to the code base. This is a completely automated step in the process with no human interaction.

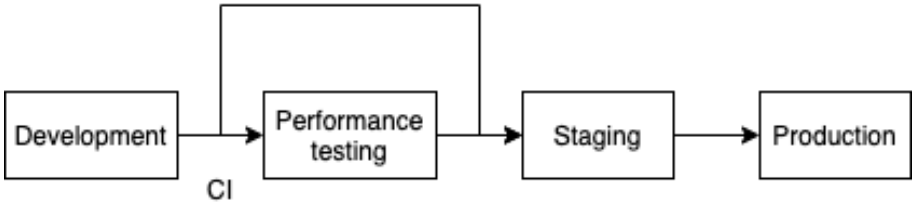

Figure 8. Application life cycle environments.

The application versions go through a life cycle of testing before putting to production as seen in Figure 8. After the automated test efforts, the software is published to a staging environment that imitates the production environment very closely, where it can be tested manually. Alternatively, the software can be put up for performance testing. The performance testing environment, like the staging environment, mimics production environment as closely as possible. The easy part is to mimic the deployment and the configuration of the production environment, but the difficult task is to imitate production-like load in the form of API calls and user actions. One way of putting the performance testing SUT under a load would be to duplicate the production traffic as is. The problem with that approach is the unpredictable nature of real world traffic, resulting in unrepeatable tests. Also, the back end state could be inconsistent with the incoming traffic. Another point is privacy. Production network traffic contains private information, which should be kept out of all testing environments to not weaken security.

The simplified architecture of the backend deployment is visualized in Figure 7. The most important parts in this thesis; the application instances and the proxy network interfaces, are circled in green. The proxy network interfaces circled in green are the points in the network where the network traffic is recorded in step 1. The function of the proxies in normal usage is to spread API calls between the backend applications and to serve some static assets. For network traffic collection the proxies also act as an aggregated source, which routes all traffic. By using the proxy network interfaces as a traffic mirroring source, one may use just one mirroring session to collect the traffic going to multiple backend applications.

Other components that depend on the backend are different types of clients, such as web clients and mobile clients. In staging and production environments the application is deployed in an AWS EC2-stack as illustrated in Figure 7. Below is a simplified list of the components that are required in the backend deployment.

- EC2 instances running the backend software in an autoscaling group
- Proxies that act as load balancers and route requests to available backend nodes
- Internet gateway (IGW) that connects the virtual cloud (VPC) to the internet
- Other AWS service integrations, such as KMS (key management service) etc.
- Route tables, routes etc. that allow the right traffic to pass to the nodes

The cloud deployment architecture of the backend application relies on AWS virtual servers called EC2 (elastic computing cloud) instances. Each EC2 instance hosts a single backend application, as illustrated in Figure 7. The number of EC2 instances running the backend applications is determined by an automated auto scaling group, which is an AWS mechanism to create or destroy virtual servers based on usage

thresholds. This means that there could be a different number of the same application running at any given time, which is called horizontal scaling. The HTTPS traffic coming from the internet is routed to two different proxies, Application Load Balancer (ALB) and HAProxy, which act as load balancers distributing traffic to the backend instances. HTTPS is terminated at the proxy level, so backend only receives plaintext HTTP traffic. As the TLS encryption is already decrypted at this stage, the captured traffic is easier to use without any further processing.

The monitoring account has relatively similar core infrastructure as the production account, however it does not contain the backend application nor its dependencies. It contains a VPC that contains EC2 instances which run *tcpdump* and record incoming packets. The monitoring account has similar networking configuration as the production account, but lacks all the overhead, such as databases and load balancing.

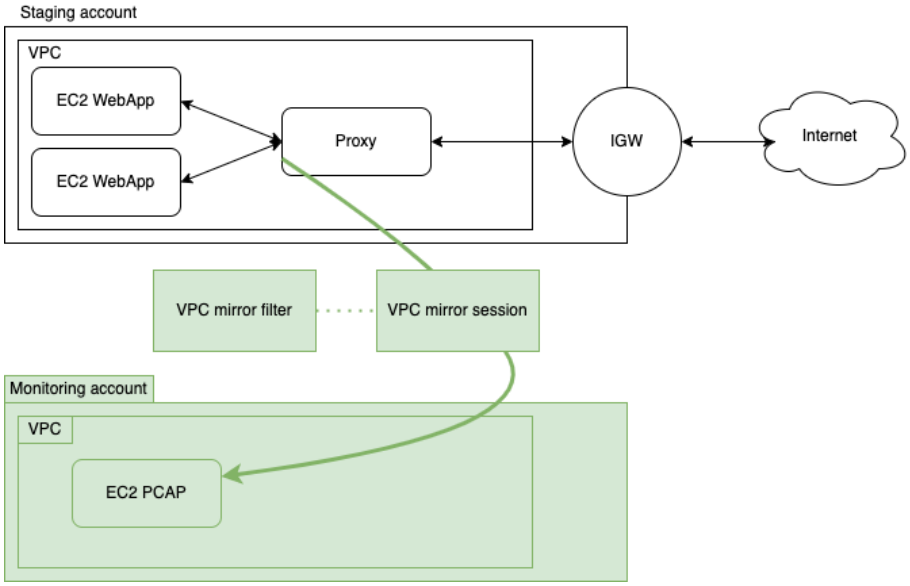

Figure 9. Simplified VPC mirroring & traffic capturing setup. Mirroring and capture components shown in green and existing components in black. EC2 instances connect to the internet gateway (IGW) either directly or through a proxy.

HTTP packets are captured with the system described in Figure 9. The system consists of two AWS accounts: a staging account that existed prior to this thesis, and a monitoring account. None of the monitoring components are on the staging account because of access control and security concerns. Isolation at the account level helps with a secure configuration.

Some AWS tools are required when using the AWS VPC mirroring to copy network traffic. Some of these are visualized in more detail in Figure 9. VPC mirror configuration, which consists of traffic filters and filter rules allow us to filter the routed traffic to consists only of the HTTP requests we are interested in. The mirrored traffic appears just as any other incoming traffic for the target of the VPC mirroring except for the protocol and port. Mirrored traffic uses VXLAN (Virtual Extensible LAN) wrapped UDP (user datagram protocol) as it provides higher throughput than TCP does.

3.2. Sequence Identification

The second step in using network data to synthesize artificial requests (Figure 6) is to identify the user sequences from the recorded traffic. In this process the user sequence identification consists of 3 steps, which are presented below.

1. **Packet reconstruction**: Reconstruct HTTP requests from a list of TCP packets by using TCP segment headers.
2. **Data cleansing**: Remove unnecessary requests, such as those performed by internal diagnostic tools.
3. **Sequence ordering**: Split the HTTP requests into lists by the user who sent the requests. Reorder by time.

Sequencing a data set consisting of large number of requests is processed with the goal of mining user sequences. It is possible to identify the user sequences from the packet capture dump from the access token headers, user-agents, and request origin, that go with each HTTP request. The user sequence also contains the timestamps of the operations, which can be used to calculate the intervals at which each operation is made. It also confirms the order of the requests. In the context of sequence modeling, the combination of HTTP method and request URL can be viewed as the action that client performs in a request. All the user sequences then are then iterated through, and the number of transitions between every two states is counted. This count reveals how prevalent each state is and how likely are the following states after that. The "time spent" in each state or the time between two states is also tracked, as it allows us to calculate the mean transition time and its standard deviation.

The network traffic dump is parsed so that all unnecessary items, such as non-HTTP frames are discarded. Only the interesting content which portrays context or purpose is kept. At this point it is possible to also tokenize any information that attaches request content to an actual user. For example, access tokens, usernames or similar fields could make it possible to view sensitive or identifying data. For this purpose, a one way process should be applied that obfuscates all identifiers into different ones so that mapping a user to a real person is no longer possible. In the case of AWS traffic mirrored requests, UDP decapsulation must be conducted before any other steps, to reveal the underlying TCP packets which contain the HTTP requests. A single HTTP request could also be split into multiple TCP packets based on the size of the request, in a process called IP fragmentation. A single dataframe can only hold as much data as is allowed by the maximum transfer unit (MTU) parameter in the network configuration.

3.3. Modeling

Third step in synthesizing network traffic (Figure 6) is about creating model from the request data that allows us to generate similar requests. In this thesis project the sequences are grouped by their characteristics into so called topics with a machine learning method called the Latent Dirichlet Allocation (LDA). LDA is a probabilistic generative model, which takes a list of documents as an input and outputs a set of topics in which the input documents belong to. The great thing about LDA is that

as a unsupervised method, the topics do not need to be predefined and the input data does not need to be labeled. However, one needs to define the number of topics that is expected to exist. Typically LDA has been used to classify text documents such as news articles into categories. Text documents consist of words, which are the *tokens* that LDA uses to classify a document into a *topic*. In the case of API sequences the *tokens* are formed from a combination of request URL and request method (GET, POST, etc.) and the documents are user sequences, which are then categorised to *topics*. For news articles the topics represent the actual topic of the article, whereas for API sequences the topics represent a type of user.

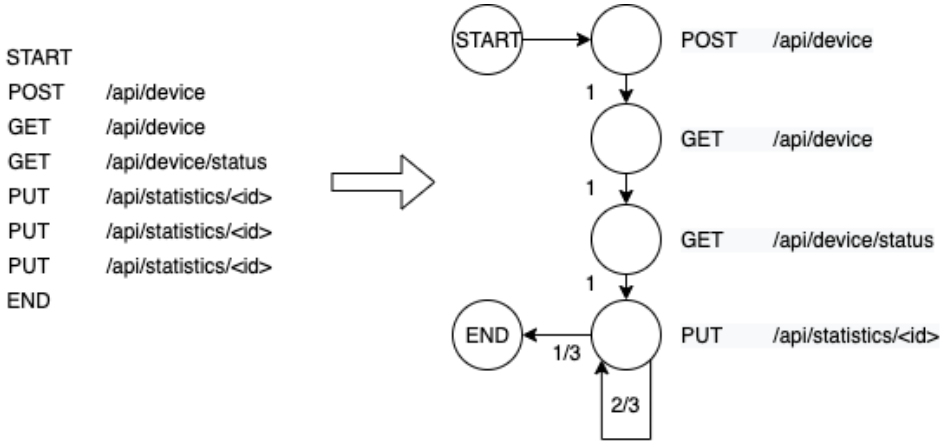

Figure 10. Multiple sequences of requests are processed into a sequence model.

LDA takes in a set of sequences (i.e. documents) which all together are called a corpus, with each of the sequences containing tokens (a combination of a URL and a method). To find the topics within the corpus and that which sequence belongs to which topic, LDA tweaks its generative model so the a new permutation of probabilities is picked. The probabilities mean that how likely is it for a token to belong to a specific topic. LDA then tries to create a new corpus consisting of artificial documents with the probabilities for the topics and words. It then compares this artificial corpus to the input data sequence. The more the artificial corpus represents the original data, the better the LDA performed and thus the generated topics explain the input data. This closeness metric allows us to optimize the generative model so that we get the most likely probabilities for the tokens to belong to a specific topic. [29]

When thinking about how a token can be categorized into topics, it can be beneficial to go through an example. Lets take token "POST-login" as an example. It is a very common token of logging into the application, and every complete sequence must have at least one occurrence of the token. It is a generic token which is just as likely to belong to any topic. Now let's consider "POST-register", which creates a new user. This is a much more specific token that could correspond to a topic "new user", as all new users must register but existing users will not perform this operation. In practise many of the token have a non zero probability to belong to each of the topics. LDA is a purely probabilistic model, with no built in context about the meaning of the documents or the tokens. This means that the model creates topics that carry no meaning other than the similarity between the documents it contains. A tool of some sort or user interaction is often applied to a define semantic meaning for the topics. For example, a

human could go through the topics and give context for the sequences within the topic by looking at the contents.

3.3.1. Dataset

The data used for modeling is the same exact set of lists of requests, output in section 3.2. The lists of requests are varying in length and content but each list has a common factor which is the individual user who sent the requests. The same user means that each list is a so called user sequence. All the sequences are in this dataset are complete, which means that they have a beginning either by logging in or by registering a new user. Sequences can end in any request, which means that the actual user using the application would just stop using the application. Only a couple of sequences ended in a logout request. The dataset used for modeling has 312 sequences in total, which altogether cover 3000 requests.

The reason why the tokens are chosen to be a combination of a URL and a HTTP method springs from REST semantics and how the API is implemented. A URL defines the address name which specifies a resource in the system, and the method defines the type of interaction with the resource: GET, POST, PUT, DELETE and so on. Using both of these identifiers together enables a more fine grained distinction between different types of actions. For example, the user has a different intent when sending a GET request to a resource compared to a PUT request. The former is about retrieving information and latter is about updating information.

3.3.2. Modeling Results

In this thesis project LDA was used to categorize user sequence into topics in which all the users share common key operations that ultimately define the action the user is trying to achieve. None of the LDA algorithms were implemented by hand, but rather through the *lda* python library[1]. The input of LDA categorization was user sequences and output was 10 topics, which had the sequences allocated within them. After the sequences were allocated into topics (T1-T10) and the topics were also named manually. Naming was performed by looking at all the sequences, categorized within the topic and a list of all the requests. The goal was to identify whether the sequences had some common characteristics between them, and if there was any ambiguity between the sequences within a topic. There were some topics which made perfect sense, and were unambiguous in what all the sequences in the topic tried to achieve - for example, to change their account credit card details. There were also some topics which had no clear common purpose, as some of the sequences were short or incomplete or the LDA algorithms just didn't perform well on these cases for some other reason. These are called interpretability issues. Clustering inaccuracy issues where the topics change between classification runs are typically referred to as topic instability.

[1] https://pypi.org/project/lda/

User categories			
Topic number	User topic name	# of user sequences	Avg. sequence length
0	New user	12	11
3	Device buyer	26	15
7	Password reset	3	9

Table 1. Some LDA output topics. Topic name is based on the assumed purpose of the sequence.

Table 1 shows statistics of a couple of of the topics output by the LDA. The user types which are shown later in the performance testing section as well are types 0, 3 and 7 which we named "New user", "Device buyer" and "Password reset" based on the functionality the user type seemed to perform. The total number of sequences per user type is the same as the number of individual users, as there was no multiple sequences of the same type for the same user in the recorded data set. Average length of the sequences in a topic is the number of requests a user of that type typically sends.

After the user topics are defined, the next step is to convert the information in LDA topics into some formal description that enables programmatical synthesis of API requests. I call this conversion of topics to request constraints the parameterization of a user model. In a sense parameterization of a user model can be seen as a simulation of client software and an end user, as the real world network traffic and the output of user models should be similar. In this thesis project, some of the edge cases are not interesting whereas the bulk of the user sequences are in the focus. This is because the goal of the performance testing is to find weaknesses in the most common usage flows. Distinguishing the most common flows can be done by comparing the number of sequences in the topics of the LDA output.

The parameterization of a user model is equal to writing to a set of rules which define the order, timing and content of each request for a user type. Large portion of the rules is about how data transforms from request to request in a sequence. For example, some authentication tokens are returned as a user logs in with their credentials. These tokens must then be used in consecutive requests to authenticate the user. The combination of all parameterized user types should define all the requests that are required to reach the goal of the user modeling.

As the sequences within each topic are very similar to each other I decided to parameterize one user sequence within each user group. The decision which user to parameterize was easy, as the users did not have highly differing API usage patterns or number of API calls which are the key features in user parameterization. In a sense the reasoning on which sequence to parameterize was quite vague. In future a more precise method could be used to pick a sequence to represent the whole user group. To increase variety, it would also be possible to parameterize multiple slightly varying users for some or all of the topics.

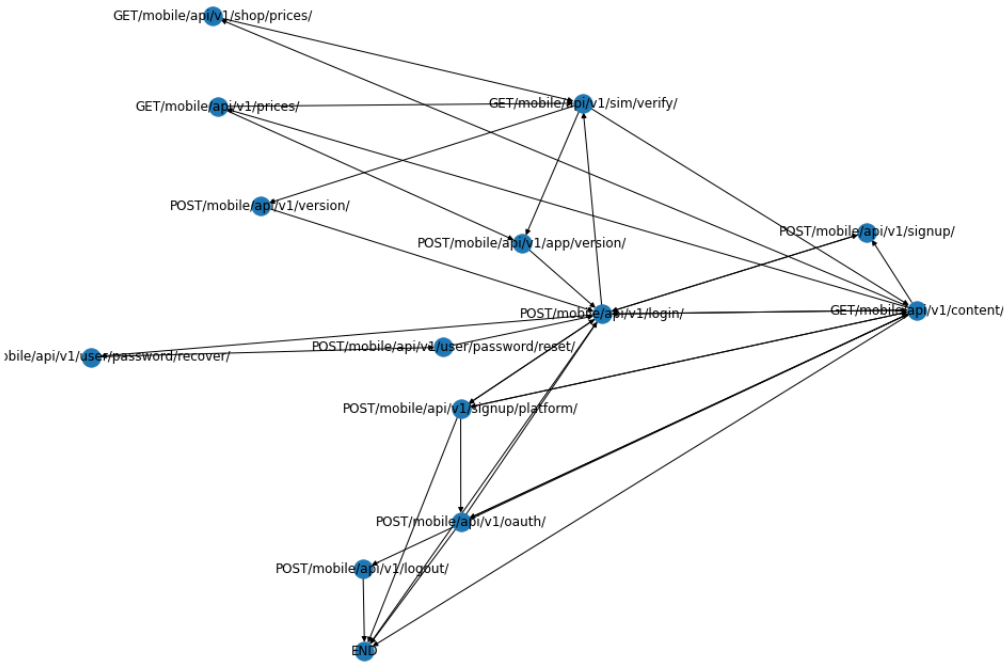

Figure 11. Markov model represented as a directed graph.

3.3.3. Markov Chains

I also experimented with a Markov chain model. As seen in Figure 11, a Markov chain can also be visualized as a directed and weighted graph. The input data for the specific model was a subset of the dataset described in section 3.3.1. Markov chain is a discrete model, in which states may connect to other states and the transitions from one state to another may have varying probabilities, which can be thought as edge weights in graph theory. To construct the model I implemented a python script which utilised pandas[2] and networkx[3] libraries for the data structure and matplotlib[4] for visualization.

In the graph each node represents a URL or a client state. The starting point in the model is the login endpoint and all flows either end at the logout endpoint or an arbitrary "END" endpoint to denote incomplete sequences. The transition from one state to another can be thought as making a request to the URL in the target node. The out edges from each node are weighted by their normalised frequency so that the weights represent the probability at which the transition happens. For example, if a transition from state A to B happens 30 times and a transition from A to C happens 10 times in the dataset, the probability of transition A to B is 75 % and A to C is 25 %. Each edge also has the mean transition time and its standard deviation attached to it. I decided against using a Markov chain model for traffic synthesis, as I couldn't find a well working method of generalizing the most common user types. Markov chain model was used for analyzing the timings of transitions.

[2]https://pandas.pydata.org/
[3]https://networkx.org/
[4]https://matplotlib.org/

3.4. Synthetic Traffic Generation

To generate network requests using the model created in the previous step one needs to create a program that translates the modeled sequence into actual network requests. In this thesis a python script was written which defines the parameters and timing to create synthetic network requests and then measures the response delays for each request.

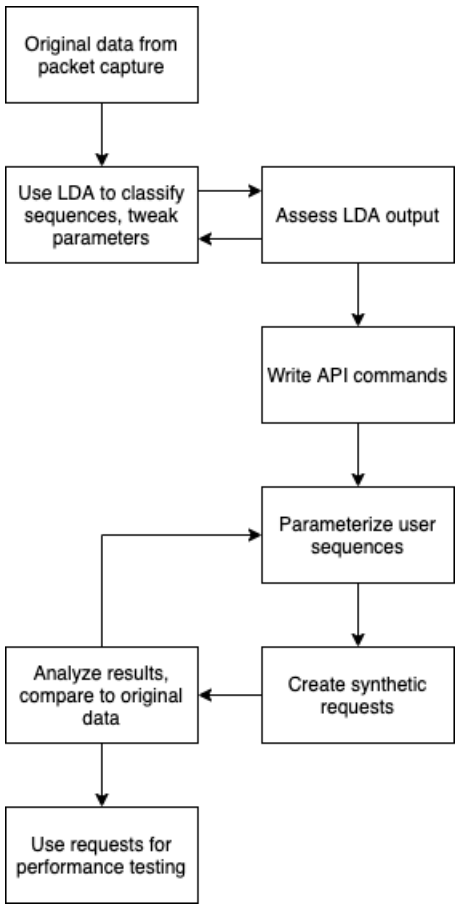

Figure 12. Overall process of modeling users for performance testing.

The performance testing script contains three main sections; API definition class, the parameterized user models, and a main program that ties these together. The API definition class contains an implementation for interacting with the web application interface. The user models contain each of the modeled user sequences so that they define a sequence of API requests which is typical within that topic. A parameterized sequence defines all the required data for each sequence and in this case it is also able to randomly generate data to populate the parameters. The required data consists of fields such as email, password or street address. The main program then creates 3 parallel instances of the user model, which start executing their built-in sequences. As the users go exhaust their sequences, new users are spun up until a time threshold is reached. Output and metrics of each user instance and each request is saved for analysis. The simplified code can be viewed in the appendices.

Wait times between each request in a sequence are drawn from the graph model created in step 3. In the graph each transition between two states contains the average time and deviation it takes to transition from one state to another. With this knowledge

one may create artificial timings by generating a sample of values that conforms to the distribution in the recorded traffic. Each transition has its own timing distribution, which was derived from the model created in Section 3.3.3. Finally a main function spins up a few of the users in parallel and they go through their sequences, recording the response times and other metadata. New users are spun up as the existing users exhaust their sequences. If the model created in step 3 (Section 3.3) is done correctly, the synthetic traffic generated with this process should have similar properties as the recorded traffic.

In Figure 12 is described the process of converting a network traffic dump into parameterizable synthetic network load through user modeling. The first step is using LDA to classify the users into topics, which are then assessed and the classification is repeated after changing parameters in LDA if necessary. Then the required API commands are implemented and user sequences are parameterized. The combination of sequences and API commands is then used to generate synthetic requests. The synthetic requests are analysed and the parameterization is repeated if the requests are not satisfactory. When the results are good enough, one may use the synthetic requests in whatever is the required use case.

Figure 13. Output of one execution of the synthetic request script.

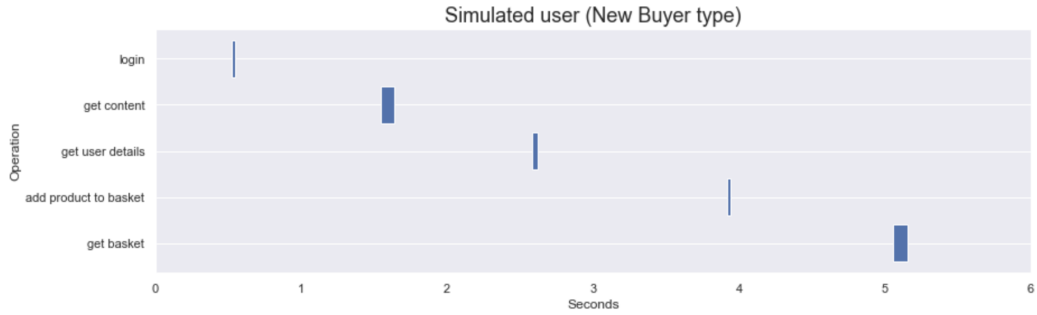

Figure 14. One simulated user sequence. This sequence would be interleaved with other sequences to generate traffic. The bars illustrate the execution of a HTTP request and the space in between is the time when the simulated user is thinking.

Output of one run of the network request creating script is visualized in Figure 13. It shows the response times for each request. Figure 14 shows the response times for each request in one simulated user sequence. The operations performed by the simulated user are on the y-axis and are in chronological order from top to bottom. The x-axis shows the passage of time. These graphs give an overview of how the application performs under stress - in this case 3 simultaneous users. The number of simultaneous users was extracted from the original network traffic data and then finding the average number of users using the application simultaneously. The number of users could easily be scaled up or down, depending on the needs of the use case. From these graphs it is possible to find the slowest methods to return, which could be an indication of performance issues that require more study. One could also average all the requests of certain type and then compare these numbers between software versions. This way it is possible to identify changes in the software performance, i.e. software regressions.

Alternatively to the method that was used to synthesize traffic, one could also generate requests by traversing a random walk within the Markov chain graph, which was the other method of modeling user behaviour that was implemented. This, however, is a more complicated method of generating sequences rather than defining each topic as a sequence in code. The Markov model has a few starting points for the random walk, a login URL, sign up URL, and a fake login URL. To complete each random walk, is a logout URL and a fake end URL, which is always the last node that a random walk visits. The reason fake login URL and fake end URL exist is that incomplete user sequences were imputed with start and end nodes in the sequence identification step in Section 3.2. The random walk algorithm is a recursive function which keeps track of the generated sequence, chooses the next node and then calls itself with the next node as the new start node.

4. EVALUATION

The research questions as defined in Section 1 are:

- **RQ1**: How effectively can Latent Dirichlet Allocation (LDA) be used to categorise user sequences in network traffic?
- **RQ2:** How effective are synthetic requests in conducting controlled performance tests for a back end application.

These questions were answered through prototypical implementation of traffic synthesis and evaluation through simple performance testing. The overall method seems promising in simulating user sequences to create artificial load, and LDA shows promise in classifying user sequences to topics to help in the process.

The statistical similarity of the packet capture and the synthetic requests was analyzed using a two-sample Kolomogorov-Smirnov (K-S) test. The basic K-S test is typically used to answer the question: "is this distribution of tokens drawn from a specific distribution, such as the normal distribution", and the two-sample version of K-S test tries to answer whether any two distributions differ [30, 31, 32]. In this case, the distributions are the original packet capture O with a length of n, and a set of synthetic requests S with a length of m. Distributions O and S contain the number of each type of request in the data sets.

$$F_O(x) = \frac{number\ of\ elements \leq x}{n} \quad F_S(x) = \frac{number\ of\ elements \leq x}{m} \tag{1}$$

The 2-sample K-S test measures the maximum distance between two distribution functions [31]. Distribution functions are functions, which have a fraction of all less than or equal observations of the variable as their value at any given point [33]. The empirical cumulative distribution functions (eCDF) $F_O(x)$ and $F_S(x)$ need to be calculated for datasets O_i and S_j with the Formulas 1. The eCDF is the same as cumulative distribution function except that as an empirical function it is a step function and its values are calculated by observing the values in the distribution [34]. Variables n and m are the total number of elements in the samples.

$$D = max|F_O(x) - F_S(x)| \tag{2}$$

The test for the two-sample K-S is done with Formula 2. In essence the formula finds the maximum distance between the two functions $F_O(x)$ and $F_S(x)$. The test was done with a python implementation of K-S, and scipy[5] [6] package was used. The two-sample Kolmogorov-Smirnov test resulted in a *distance* **D** of **0.38** and *p-value* of **0.09**. The *distance* **D** is low enough and the p-value is high enough ($p > 0.05$) to not reject the hypothesis that the distributions of the two samples are the same. In light of the K-S test, it can be concluded that the original packet capture and the set of synthetic requests are statistically similar.

The statistical test outputs values that indicate statistical similarity between the two distributions. When comparing the two traffic data sets, it is clear that some patterns

[5]https://pypi.org/project/scipy/

[6]https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.ks_2samp.html

are present in both samples and that are some differences. The statistical K-S test likely outputs valid values, but they might not paint a complete picture or could be misleading when judging the usefulness of the approach. Even though the statistical analysis indicates similarity, the method could likely be improved further. One difference originates from the removal of environment specific requests in a preprocessing step in Section 3.2. The environment specific requests are produced by internal diagnostics tools and test procedures, and are not required for testing the application software. The environment specific requests are quite prevalent in the recorded traffic. The preprocessed traffic was then an input for the modeling phase in Section 3.3. As one may guess, the topics only contain sequences and requests that are in the input data. Second reason that explains differences in similarity was that not all user topics were modelled and parameterized. Not all the original users were represented in the synthetic traffic. This skews the distribution of requests away from the recorded traffic.

It is also to be noted that perfect similarity between the original and synthetic load may not be necessary for the synthetic traffic to be useful in performance testing. Performance testing needs typically only require the load to contain the most common or critical user sequences - not necessarily all the corner cases. However, this is a future consideration that the similarity could be improved through better data cleaning, topic clustering, modeling, parameterization, and synthesis.

The value that utilizing LDA brings in the process is the sequence categorization through topics. The process of creating synthetic user sequences could be done without any preexisting information about actual user behaviour, but that would leave room for a lot of guessing about the actual sequences. Guessing the artificial sequences would likely result in more uncertainty about the similarity between the test environment and the production environment. Although the model-based approach aids in finding the user sequences to parameterize, it is wise to take the approach with a grain of salt, as there is some level of uncertainty included. Some other method for model could be used in conjunction or just for verification to have an increased level of confidence in the result.

In the perspective of this web application, the created model and the performance test script are applicable but work on a proof of concept level. The type of output from the performance testing script is exactly what is needed for analyzing the performance of a web application in a load test. For the approach to actually test the production software, some modifications should be made throughout the process. The recorded traffic would need to be either from a different environment or more thoroughly processed and parameterized. In the case of this thesis, the recorded traffic was not representative of the actual usage but rather of some testing scenarios.

4.1. Discussion

User sequence classification with LDA was the most study intensive part of this thesis. LDA is a machine learning classification method with unsupervised training process and it is not known to be the easiest to evaluate. Most important part of determining whether the user models are good seemed to be labeling the topics with experts of the application usage. The labeling was conducted by a software developer working in the project and a lead tester who focuses in the application. The tester had no prior

knowledge of the method for user modeling. They were very quick to realise that most of the user groups seemed to have some sort of theme within them. As the LDA categorises the sequences only by the tokens within each sequence, they lack some context of the complete sequences. It is a useful convention to try to name each user type within the model, to see if it makes sense as a user. It was possible to name most of the output groups, and they seemed to map to a type of user we would presume to exist such as a "new buyer" or "password reset".

Topic instability is another thing to consider. It is something that could lead to erroneous results in the topic clustering, and parameter optimization could improve the stability of the results [35]. In this thesis I did not go very deep into whether the model is good or bad other than by eyeballing the results. There are methods that could compress the model characteristics into a single metric, which helps in fine tuning the LDA training parameters. In the case of LDA these are often coherence metrics and the perplexity measure.

The most time consuming task was to implement the network data collection in the cloud. In an automatically scaling environment it was surprisingly complicated to create the required infrastructure to capture traffic in a secure way. The network data collection is now a capability that we have in the company as all the components and infrastructure required for the capture is written as code to allow for faster setup next time. This part of the process would be much faster in a less complicated environment with fewer constraints to consider.

The basic parameterization of the user model was not very difficult. In fact, I think it was only a matter of putting in a days work to write the parameters and constraints that define the API requests and their sequences. There is not many things that could go wrong with the content or ordering of requests within a sequence, as there is often only one valid way of using an API. It also helps, that I also develop the API so the requests and usage flow is rather familiar to me. A more error prone and difficult task is deciding on which topics to parameterize for traffic synthesis. Also, creating realistic timings between each request is not trivial, as one need to simulate human behaviour to achieve realistic timings. One could tweak with the values of the parameters, but in some cases that gets more into the area of model-based testing. Once the modeling is complete, which is the difficult part, it is a rather straightforward process to get into sending requests and measuring some metrics from them.

Another interesting aspect is whether the traffic generation should be more adjustable or have deterministic qualities. Currently there is no way of configuring the generated requests with any other parameters but the number of sequences. Deterministic generation would be able to create same results each time for a given seed, using the same seed for any randomizing functions within the generation process. This would allow reproducing of the results of a previously used seed. The current implementation falls short on this front, as it is not able to create the same set of sequences twice, unless by completely random chance. The parameters that change between executions are wait times between requests and request parameters such as a username or the number of bought devices. Currently the timing parameter is constrained by the distribution by which it is picked at random for each execution and the other changing parameters are generated with a python library called faker[7]. The

[7]https://faker.readthedocs.io/en/master/

optimization of these parameters could be part of the process to improve the quality of the generated traffic.

Performance test result visualization and analysis could also be an interesting topic for a further study. In Section 3.4 two diagrams were drawn to visualize the output of the performance testing script. Currently the visualization is implemented with a python script and produces static image files. A more interactive process could be implemented, which could also be used to compare the changes in the output between different runs of the performance test software. Visualization and analysis of the results are important topics to consider if this approach was used in large scale for application performance testing. After all, raw data is often not as useful for us humans, compared to processed information.

One more thing to consider when using LDA or any other modeling method is the life cycle of the system. In this thesis the implementation of user modeling takes a snapshot of the application traffic as an input. What is not taken into consideration is the evolution of the interfaces, user flows and consequently the network traffic. In the case of this application, an outdated model would likely result in poor synthetic traffic and unpredictable performance testing results. The quality of the synthetic traffic is a measurement whether it matches the application and how well it simulates its users. The quality of the traffic might not be obvious at first, but would probably cause problems in the long run. The evolutionary model decay could be addressed by updating the models regularly. This is something that should be studied, if the approach was to be deployed in large scale.

4.1.1. Future Work

A thing to study further would be how such user modeling could be used to help in business decisions. As the model helps in distinguishing user groups, it could be very helpful in finding business insight about the user base. The information could help, for example, when making business decisions about which features to prioritise for development. Another thing could be about learning how a specific group of users uses the application. In this case, the models should represent actual traffic exactly, which is not a requirement for performance testing. Some of this information is typically recorded at the client level, such as with Google analytics, but the method I am proposing could help in discovering a more complete picture of the users.

5. SUMMARY

In this thesis a 5-stage approach for creating a synthetic network load for a web application backend was implemented. The approach consists of network traffic recording, user sequence identification, user modeling, sequence parameterization, and finally performance testing.

Using network traffic recordings for identifying user sequences in a cloud environment is a data-oriented way of finding input data for modeling users. The data-oriented nature of the process increases the quality of the output, by making the output more similar to the environment that is simulated. The field of web application user modeling is still in development and the solutions require a lot of manual labour to work. This is also the case for the method used in this thesis, for example the parameterization requires manual work, as shown in Section 3.3.2. The next step in automation would be to estimate the parameters for the synthetic traffic automatically, based on the recorded network traffic.

The most interesting part of this thesis is the modeling of the users. This is also the step where there are multiple different ways of accomplishing a similar result. In this thesis a machine learning technique, LDA (latent dirichlet allocation) was used to help in categorizing user sequences. The applicability of LDA to help in clustering sequences shows promise, but a lot of care should be put in the tuning for optimal results. Depending on the tuning, a different output could be observed.

The overall process of data transformations is delicate with many different phases that could go wrong. In this thesis some of the phases such as traffic recording, data preprocessing, and LDA model tuning were not optimal. Regardless of the non-optimal phases, a Kolmogorov-Smirnov statistical test shows similarity between the original traffic data and the simulated traffic. This implies that the 5-stage approach used in the thesis is able to produce simulated traffic, which shares characteristics of the real world network traffic. The overall approach seems applicable for using in this particular case of web application and is likely usable in other applications too.

6. REFERENCES

[1] Papazoglou M.P. & Georgakopoulos D. (2003) Introduction: Service-oriented computing. Communications of the ACM 46, pp. 24–28.

[2] O'reilly T. (2009) What is web 2.0. " O'Reilly Media, Inc.".

[3] Srirama S.N., Iurii T. & Viil J. (2016) Dynamic deployment and auto-scaling enterprise applications on the heterogeneous cloud. In: 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), IEEE, pp. 927–932.

[4] Andrews J.H. (1998) Testing using log file analysis: tools, methods, and issues. In: Proceedings 13th IEEE International Conference on Automated Software Engineering (Cat. No. 98EX239), IEEE, pp. 157–166.

[5] Pomata S., Mirror production traffic to test environment with vpc traffic mirroring. URL: `https://aws.amazon.com/blogs/networking-and-content-delivery/mirror-production-traffic-to-test-environment-with-vpc-traffic-mirroring/`.

[6] Ammann P. & Offutt J. (2016) Introduction to software testing. Cambridge University Press.

[7] Harrold M.J., Jones J.A., Li T., Liang D., Orso A., Pennings M., Sinha S., Spoon S.A. & Gujarathi A. (2001) Regression test selection for java software. ACM Sigplan Notices 36, pp. 312–326.

[8] Shoaib Y. & Das O. (2011) Web application performance modeling using layered queueing networks. Electronic notes in theoretical computer science 275, pp. 123–142.

[9] Alam M., Gottschlich J., Tatbul N., Turek J.S., Mattson T. & Muzahid A. (2019) A zero-positive learning approach for diagnosing software performance regressions. In: Advances in Neural Information Processing Systems, pp. 11627–11639.

[10] Hooda I. & Chhillar R.S. (2015) Software test process, testing types and techniques. International Journal of Computer Applications 111.

[11] Molyneaux I. (2014) The art of application performance testing: from strategy to tools. " O'Reilly Media, Inc.".

[12] De Barros M., Shiau J., Shang C., Gidewall K., Shi H. & Forsmann J. (2007) Web services wind tunnel: On performance testing large-scale stateful web services. In: 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), IEEE, pp. 612–617.

[13] Fielding R.T. & Taylor R.N. (2002) Principled design of the modern web architecture. ACM Transactions on Internet Technology (TOIT) 2, pp. 115–150.

[14] Meier J., Farre C., Bansode P., Barber S. & Rea D. (2007) Performance testing guidance for web applications: patterns & practices. Microsoft press.

[15] Sarker I.H., Colman A., Han J., Khan A.I., Abushark Y.B. & Salah K. (2020) Behavdt: a behavioral decision tree learning to build user-centric context-aware predictive model. Mobile Networks and Applications 25, pp. 1151–1161.

[16] Cho Y.H., Kim J.K. & Kim S.H. (2002) A personalized recommender system based on web usage mining and decision tree induction. Expert systems with Applications 23, pp. 329–342.

[17] Li W. (2013) Specification mining: New formalisms, algorithms and applications. University of California, Berkeley.

[18] Mukhiya S.K. (2016) Predicting the next click with Web log Process Mining. Master's thesis, NTNU.

[19] Almulhem A. & Traore I. (2007) Mining and detecting connection-chains in network traffic. In: IFIP International Conference on Trust Management, Springer, pp. 47–57.

[20] Caselli M., Zambon E., Amann J., Sommer R. & Kargl F. (2016) Specification mining for intrusion detection in networked control systems. In: 25th {USENIX} Security Symposium ({USENIX} Security 16), pp. 791–806.

[21] Xing Z., Pei J. & Keogh E. (2010) A brief survey on sequence classification. ACM Sigkdd Explorations Newsletter 12, pp. 40–48.

[22] Gundersen O.E. (2012) Toward measuring the similarity of complex event sequences in real-time. In: International Conference on Case-Based Reasoning, Springer, pp. 107–121.

[23] Hay B., Wets G. & Vanhoof K. (2004) Mining navigation patterns using a sequence alignment method. Knowledge and information systems 6, pp. 150–163.

[24] Wang D. & Huang G.B. (2005) Protein sequence classification using extreme learning machine. In: Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005., vol. 3, IEEE, vol. 3, pp. 1406–1411.

[25] Wu C., Berry M., Shivakumar S. & McLarty J. (1995) Neural networks for full-scale protein sequence classification: Sequence encoding with singular value decomposition. Machine Learning 21, pp. 177–193.

[26] Chuzhanova N.A., Jones A.J. & Margetts S. (1998) Feature selection for genetic sequence classification. Bioinformatics (Oxford, England) 14, pp. 139–143.

[27] Veiga G.M. & Ferreira D.R. (2009) Understanding spaghetti models with sequence clustering for prom. In: International conference on business process management, Springer, pp. 92–103.

[28] van der Aalst W.M., Bolt A. & van Zelst S.J. (2017) Rapidprom: mine your processes and not just your data. arXiv preprint arXiv:1703.03740 .

[29] Blei D.M., Ng A.Y. & Jordan M.I. (2003) Latent dirichlet allocation. the Journal of machine Learning research 3, pp. 993–1022.

[30] Massey Jr F.J. (1951) The kolmogorov-smirnov test for goodness of fit. Journal of the American statistical Association 46, pp. 68–78.

[31] Lopes R.H., Reid I. & Hobson P.R. (2007) The two-dimensional kolmogorov-smirnov test .

[32] Fasano G. & Franceschini A. (1987) A multidimensional version of the kolmogorov–smirnov test. Monthly Notices of the Royal Astronomical Society 225, pp. 155–170.

[33] Weibull W. et al. (1951) A statistical distribution function of wide applicability. Journal of applied mechanics 18, pp. 293–297.

[34] Drion E. (1952) Some distribution-free tests for the difference between two empirical cumulative distribution functions. The Annals of Mathematical Statistics 23, pp. 563–574.

[35] Mantyla M.V., Claes M. & Farooq U. (2018) Measuring lda topic stability from clusters of replicated runs. In: Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement, pp. 1–4.

7. APPENDICES

```
import json
import time
import random
import uuid

import asyncio
import aiohttp

from faker import Faker

output_list = []
max_input = 0
instances = []
session = aiohttp.ClientSession()

class Apicommands():
    def __init__(self):
        self.apiurl = "http://127.0.0.1:8080"

    def timer(func):
        def wrapper(*args, **kwargs):
            start_time = time.time()
            result = func(*args,  **kwargs)
            total_time = time.time() - start_time
            output_list.append{"function": func.__name__,
                "user": args[2],
                "duration":total_time,
                "start_time": start_time}
            max_input += 1
            return result
        return wrapper

    def save_timing(self):
        output_file_name =  time.strftime("%Y:%m:%d-%H:%M:%S") +
        with open(output_file_name, 'a') as out:
            out.write("method,user,duration,start_time")
            out.write("\n")
            for item in output_list:
                row = []
                for pair in item:
                    row.append(str(item[pair]))
                row = ",".join(map(str, row))
                out.write(row)
                out.write("\n")
```

```
    @timer
    async def post_login(self, session,
        caller, username, password, bearer_token):
        req_url = self.apiurl + "login"
        req_headers = {
            "Content-Type":"application/json",
            "Authorization": "␣Bearer␣" + bearer_token}
        req_data = json.dumps({"email": username,
            "password": password})

        async with session.post(req_url,
            data=req_data,
            headers=req_headers) as response:
                html = await response.text()
                return json.loads(html)

    # TODO: Define all the required api commands here

class User():

    def __init__(self):
        instances.append(self)
        self.api = Apicommands()
        self.faker = Faker()

    async def get_tasks(self):
        tasks = []
        task = asyncio.create_task(
            self.sequence(session))
        tasks.append(task)
        print(tasks)
        await asyncio.gather(*tasks)

    async def schedule(self):
        tasks = []
        async with aiohttp.ClientSession() as session:
            task = asyncio.create_task(self.sequence(session))
            tasks.append(task)
            return tasks

    async def _generate_userdata(self):
        user = {
            'email': self.faker.ascii_safe_email(),
            'password': self.faker.password()
        }
        return user
```

```
    def _create_id(self):
        return str(uuid.uuid4())[:8]

class Returning_Buyer(User):
    def __init__(self):
        super().__init__()
        self.name = self.__class__.__name__ +
            ":" + (self._create_id())

    # Signup, login, get content, get prices
    async def sequence(self, session):
        login = await self.api.login(session,
            self.name,
            "user@instituti.on",
            "secret",
            oauth["access_token"]
            )
        await asyncio.sleep(1)

        # TODO do all the required requests here

        instances.pop()
        return login

# TODO parameterize all user types
# users must have the "sequence" method defined

classes = (Returning_Buyer)
while True:
    print(len(output_list))
    if len(instances) < 3:
        instance = random.choice(classes)()
        print("instances: ", instances)
        tasks = instance.get_tasks()
        asyncio.ensure_future(tasks)
        await asyncio.sleep(3)
    if len(output_list) > max_input:
        break
```