CZECH TECHNICAL
UNIVERSITY
IN PRAGUE

F3

Faculty of Electrical Engineering
Department of Computer Science

Master's Thesis

Scalability of Car Sharing System

Bc. Filip Ravas
Open Informatics, Software Engineering

May 2019
Supervisor: doc. Ing. David Šišlák, Ph.D.

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Ravas**	Jméno: **Filip**	Osobní číslo: **434955**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Studijní obor:	**Softwarové inženýrství**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Scalability of Car Sharing System

Název diplomové práce anglicky:

Scalability of Car Sharing System

Pokyny pro vypracování:

This diploma thesis addresses the scalability aspect of Car Sharing System with respect to the number of users and their car shares.
1) In the existing system identify scalability bottleneck.
2) Propose proper application performance metrics to evaluate scalability of the system.
3) Implement appropriate simulation where the system can be loaded by varying number of users and resulting car shares.
4) Design and implement changes in the system which leads into better scalability of the system.
5) Perform empirical evaluation of the existing and proposed changes in the system using developed simulation.
6) Discuss results and propose further improvements of the system.

Seznam doporučené literatury:

[1] Gormley C., Tong Z.: Elasticsearch: The Definitive Guide. O'Reilly, 2015.
[2] Chodorow K.: MongoDB: The Definitive Guide. O'Reilly, 2013.
[3] DeCapua T., Evans S.: Effective Performance Engineering. O'Reilly, 2016.
[4] Jain R.: The Art of Computer Systems Performance Analysis. Wiley, 1991.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. David Šišlák, Ph.D., centrum umělé inteligence FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **18.02.2019** Termín odevzdání diplomové práce: **24.05.2019**

Platnost zadání diplomové práce: **19.02.2021**

doc. Ing. David Šišlák, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Ing. Pavel Ripka, CSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání	Podpis studenta

Acknowledgement / Declaration

I would like to express my special thanks and gratitude to doc. Ing. David Šišlák, PhD. for providing professional guidance, experience and valuable knowledge that I acquired during consultations.

I hereby declare that the thesis here submitted is original except for the source materials explicitly acknowledged with respect to the Ethical Guidelines for University Final Theses. I also acknowledge that I am aware of legal regulations related to the thesis publication, particularly the Act N.o 121/2000 Coll. of the Czech Republic, the Copyright Act, and especially that the Czech Technical University in Prague is eligible to conclude a fixed-term licensing contract applied to the thesis as a school work in accordance with § 60, Article 1 of the Copyright Act and with respect to the duration of the Contract.

The handling of the thesis is governed by the Contract of cooperation between the Czech Technical University in Prague, ŠKODA AUTO a.s. and Smart City Lab s.r.o. managing the research Project „Carsharing for university students“. The Contract is published in the national registry of contracts and has ID 5546159.

I am obliged to the non-disclosure agreement (NDA) governing the third-party access to the information I have obtained or created within the Project.

With accordance to the statements above, the work cannot be publicly accessible until the NDA agreement is valid, explicitly until May 31st 2023.

In Prague, 23. 5. 2019

....

Abstrakt / Abstract

Predmetom diplomovej práce bolo vytvoriť a implementovať platformu pre výkonnostné vyhodnotenie existujúceho car sharing systému, za účelom identifikácie úzkeho miesta škálovateľnosti a následne navrhnúť zmeny pre zaistenie lepšej škálovateľnosti a stability systému v budúcnosti. Prvé štádium zahŕňalo definovanie metrík pre vyhodnotenie výkonnosti, nájdenie vhodného nástroja pre zber dát a automatizáciu celého procesu. Dôkladné testovanie odhalilo, že ukladanie všetkých telemetrických dát do jednej tabuľky v relačnej databáze je neefektívne. Navrhnuté riešenie spočívalo v ukladaní len posledných dát a presmerovaní príjmu telemetrie do vhodnejšieho NoSQL úložiska dát. Vyššie zmienené zmeny v systéme mali za následok významný nárast výkonnosti a stability pod veľkým zaťažením a pripravili ho pre ďalšiu expanziu projektu. Okrem toho testovacia platforma poskytuje užitočný základ pre vylepšenia systému v budúcnosti.

Kľúčové slová: diplomová záverečná práca, škálovateľnosť, performance testing, car sharing, SQL, NoSQL, MongoDB, Elasticsearch

The subject of this diploma thesis was to create and implement performance testing platform for existing car sharing system, in order to identify scalability bottleneck, followed by design changes that would ensure better scalability and stability of the system in the future. The first stage involved defining metrics for performance evaluation, finding a fitting tool for data collection and automatization of the whole process. Thorough testing revealed that saving all of the telemetry data into one table of a relation database is too inefficient. The proposed solution was to save only the latest data and redirect telemetry ingestion into more suitable NoSQL data store instead. The aforementioned changes in the system resulted in major increase in performance and stability under heavy load and prepared it for further expansion of the project. Additionally, the testing platform provides a useful baseline for other improvements of the system in future.

Keywords: Master's thesis, scalability, car sharing, performance testing, SQL, NoSQL, MongoDB, Elasticsearch

Contents /

Chapter 1
Introduction

1.1 Shared economics and car sharing

Consumption of goods is the driving force of the economy and our society as we know it. Consumption was always associated with ownership, however, the increased popularity of services such as Netflix, Spotify, Uber, Rekola or Lime in recent years shows a change in the way we approach things. All of these services offer an alternative to ownership - so-called access-based consumption [1]. Providing access to goods without owning them for a fee based on the amount of the goods or the length of the contract. This opens up a lot of new options for the consumers as ownership can be impractical.

Ownership of a car is a great example. First of all, the prices of new vehicles are too high for young people who just started working or are still at university. The value of a brand new car drops significantly immediately after its purchase. The other option is buying a used car, but that carries a lot of risks, such as uncertain condition of the car, short or nonexistent warranty or even unethical and illegal practices by car dealers e.g. lowering the mileage. The car also has to be serviced - tires changes, refueling, cleaning, annual technical inspections/checkups not only cost money but more importantly take up time. As the car gets older it also tends to break down more frequently and the repair costs go up. Uber/Taxi services provide an alternative, but because the cost also involves paying a driver it doesn't make sense economically in some situations like going shopping, going on a trip, etc. because it gets too expensive. They also carry the risk of bad drivers or potential scammers. This is why car sharing services have become more and more popular in recent years as they provide the best of both worlds.

Vehicles of multiple car sharing companies can be observed in the streets of Prague. Consumers can either access cars owned by a company (CAR4WAY, AJO) or they can choose a peer-to-peer sharing model (HoppyGo) in which the vehicles are provided by other users of the service.

The reported number of students in Prague for the year 2018 was 90 758[1]. Students typically can't afford to buy a car and cover the expenses that come with it. They are dependent on public transport which isn't practical for moving to or from dormitories, shopping for furniture or other large objects. Before starting the project, we sent out a questionnaire to the students. They showed great interest in the service and also often mentioned that if there was an option to use a shared car, they would use it to go on weekend trips outside of the city.

1.2 Uniqway project

The Uniqway project was created as a result of joint efforts of three Prague universities (CTU, VSE, CULS) to bring car sharing to students. What is really unique about the project is that the creators of the project are exclusively students under the supervision

[1] https://dsia.msmt.cz//vystupy/vu_vs_f1.html

of mentors from the universities. They are responsible for all of the technical solutions, marketing, economy, social media interactions and all the other duties involved in the upkeep of such a large project.

Responsibility of team from CTU is to create a technical solution for this service. The platform consists of a backend server and multiple client appliactions: a smartphone app for end users (native Android and iOS), a web application for car park administrators and a hardware module located inside of the car. The tech team includes around 15 people from the Faculty of Electrical Engineering, Faculty of Information Technology and Faculty of Mechanical Engineering.

1.2.1 First stage

When the project started the software team consisted of only three fairly inexperienced students with limited resources and only one car to take care of. As such the technolgy used matched the small scale of the project and served more as a proof of concept rather than actual full scale infrastracture.

1.2.2 The execution of plans

The first real plans involved six Skoda Fabia vehicles provided by ŠKODA AUTO DigiLab to establish the service and start raising awareness among the students of the involved universities. During this part of the project the backend infrastructure created in the early stages was still sufficient and showed no significant problems with it's functionality.

1.2.3 The expansion

The car park currently consists of 17 cars. There are circa 900 pre-registered users, around 400 of them are activated, meaning they can actually use the service. On average the service reaches approximately one hundred car shares every week, resulting mileage of around 3000 kilometers. These rising numbers proved to be problematic for the old infrastructure and it started developing bottlenecks. A solution was needed, as well as developement of new ways to stresstest the system.

Chapter 2
Existing system description

The existing software system of the service is based on HTTP server that is providing API for client applications. There are three main client types that are interacting with the system:

- **Carsharing user:** Or our customer - student. For successful usage of the service, user needs to be able to choose from available vehicles, reserve the one that is the most suitable for him (car type, its location, etc.), physically find it and be able to lock or unlock it at will. Finally, the user has to end the reservation and pay for the service. Nowadays we can assume that the majority of students own a smartphone, so we decided to implement the client application in form of smartphone app. We started with development for Android systems (Android smartphone are generally more accessible in terms of price and they make up most of the smartphone market) and now we also have a full-featured app for mobile devices with an iOS operating system.
- **Car park administrators:** Group of users with special privileges, that are working in the project, who need to have admin access to the system. They require access to the car locations, reservations, users and other general information for the purpose of maintenance and customer support in case of unexpected events. API endpoints for these users are different and to access them, logged user needs to have assigned administrator role.
- **Car module:** Hardware unit embedded in a car that has multiple purposes. It has several parts: RFID card reader that is serving for two-factor authentification, GPS module and connection to on-board diagnostics for data gathering. The module has a connection to the Internet and acts like HTTP client that sends telemetry to the backend. The module is also critical for (un)locking of the car.

2.1 Used technologies for backend system

While choosing technologies to use for backend, we had to keep in mind that the team developing the project consist exclusively of students. This means we should mainly consider technologies that are part of the university's curriculum. Because of that, we choose Java as the main programming language for backed implementation. During studies, the students are introduced to Java Enterprise Edition, but because of the complexity of the whole platform we decided not to use it. Instead of Java EE we decided to go with Play - an open source framework which follows MVC architectural pattern. Some reasons behind this decision are:

- **Good documentation quality:** Documentation of the framework/platform is a fundamental property because it is the main source of the information, therefore critical for the usability of the selected technology

- **Active community:** Community is a vital part of open source projects, it's really important to have a large and active community that is often contributing to the framework - bug fixes, features requests and overall development.
- **Modern stack trace and error handling**: Also a feature that makes development smoother - compilation error is printed in a browser (while the application is running, thanks to in-time compilation) together with code snippet
- **Previous experience with framework**: Previous experience of multiple project developers with Play Framework was also a factor in the decision process

For persisting of a state of the system, we could choose from the number of available technologies. We decided to use a traditional relational database, that we've had experience with from university - PostgreSQL. It is widely used and suitable for well-structured data.

2.2 Deployment infrastructure

Every software system has to have an underlying infrastructure where it's deployed. Because we don't have the capacity and resources to maintain the infrastructure from bottom up, we outsourced taking care of the hardware. Uniqway has a contract with the Technical University of Ostrava that is providing virtual machines for the project. The operational environment consists of three virtual machines:

- **api.uniqway.cz:**
  - 2 CPU, 2 GB Memory
  - virtual machine with Nginx that is used as a proxy that forwards HTTPS requests to proper backend application and is used by both production and staging environments
- **vsb-environment-core-1.uniqway.cz:**
  - 2 CPU, 16 GB Memory
  - virtual machine running backend application
- **vsb-environment-db.uniqway.cz:**
  - 4 CPU, 20 GB Memory
  - virtual machine running PostgreSQL database server

Since Uniqway is a students' project, we simply don't have enough resources and finances to pay for expensive enterprise software and licenses. That is why we decided to use mainly open source software. When we were deciding which tools to use for a specific purpose (monitoring, build management, etc.) we took into consideration factors like community size and it's activity. How frequent are contributions? Are bugs being fixed in a reasonable time? Are feature requests taken into consideration? And of course, an important factor is also previous experience with given tool - either from university or work. When it comes to system infrastructure, our goal is to have a platform as simple and as unified as possible. The operating system of all of the virtual machines is CentOS, currently in version 7. For the purpose of unifying deployment of the backend application, build management server, and other necessary applications we decided to use open source container platform `Docker`[1].

[1] https://www.docker.com/

For minimizing operations work and manual system changes we apply infrastructure as a code concept. For system management we use `Puppet`[1], every Docker deployment is defined using `Docker Compose`[2]. For the build management we use `TeamCity`[3]. It's also necessary to monitor the infrastructure. For this purpose we use `ecosystem of Elastic Stack`[4]. Our infrastructure for monitoring consists of `Elasticsearch`[5] and `Kibana`[6] deployed on separate virtual machine. We use `Metricbeat`[7] for collection of system resources utilization data and their exporting to Elasticsearch.

[1] https://puppet.com/
[2] https://docs.docker.com/compose
[3] https://www.jetbrains.com/teamcity/
[4] https://www.elastic.co/guide/en/infrastructure/guide/current/infrastructure-monitoring-overview.html
[5] https://www.elastic.co/products/elasticsearch
[6] https://www.elastic.co/products/kibana
[7] https://www.elastic.co/products/beats/metricbeat

Chapter 3
Application performance metrics

As our service grows and gains more and more users, there is also a bigger demand for system features. With more features, system complexity also increases. One of the challenges that come with the implementation of new features and increased load is to maintain system performance. It is desirable to avoid problems connected with slow website loading, failing requests and other indicators of poor performance. Such system behavior could result in users moving to another platform that is not suffering from mentioned problems. In order to avoid that, metrics data are needed, so that overall performance could be evaluated. Metrics that indicate system behavior may differ and have to be set before any data collection starts.

Selection of such metrics is usually based on system requirements. In our case, the user needs to be able to comfortably interact with the system and be able to execute basic actions connected to the usage of the service - clients lean heavily on response time so that the application doesn't load too long, possibly confusing the user. Besides that, it's also important to process all of the requests from the modules inside of the cars in order to have the most up to date information about their location and another telemetry.

3.1 User Satisfaction / Apdex Scores

`Apdex`[1] scoring system works by specifying a target response time length for a specific HTTP request or a transaction. These transactions are then bucketed into categories: satisfied (fast), tolerated (sluggish), too slow, and failed requests. A simple math formula is then applied to provide a score from 0 to 1, where 0 means that all requests were frustrated and 1 means all requests were satisfied.

$$\text{apdex} = \frac{\text{satisfied count} + \frac{\text{tolerated count}}{2}}{\text{total request count}}$$

Figure 3.1. Apdex formula

3.2 Average and median response time

Response time is the amount of time an application takes to return a response for a user request. Requests are sent to the server, the time between first and the last byte sent is measured for every request. From measured response times average and median response time can be obtained. We usually work with the average response time, however, average metrics are affected by extremes and that is why median response time is collected as well.

[1] http://apdex.org/overview.html

3.3 Error rates

Applications errors are important health metrics of the system, they usually indicate greater problems and it's critical to keep their occurrence as low as possible. We can differentiate types of errors:

- HTTP Errors - Client applications sent a request to the server and server returns status code that starts with the digit 5. Such return code means that the server failed to complete the request and error occurred on the server side.
- Logged Exceptions - We can also count a number of exceptions that appeared in application logs. They indicate unexpected system event or state and they are also often accompanied with HTTP error.

3.4 CPU and memory usage

If the CPU usage on the server is extremely high, it will lead to application performance problems. Monitoring the CPU usage of the server and applications is a basic and critical metric. The same principle applies to memory usage. Tracking of a system utilization metrics can be integrated with mechanisms like auto-scaling when the system automatically spins up a new node that will join the existing pool and will start processing the current system load.

3.5 Metrics collection and evaluation

All of the aforementioned metrics are relevant for the proper functionality of our system and we, therefore, need a performance testing tool, that can not only collect all of these data but also simulate more complicated load than a simple GET request.

In order to access the collected information and use it for its intended purpose, it needs to be easily accessible. The best solution would be to send them to already deployed Elasticsearch, which handles the system load metrics from the virtual machines as well so that we could access them comfortably through Kibana. This would allow easy filtering and creation of visual reports.

Chapter 4
Implementation of appropriate simulation

As mentioned in the previous chapter, the choice of a tool for performance testing depends on two main factors. Firstly, for purposes of scalability bottleneck identification, the tool has to be able to simulate the behaviour of all our client types and secondly the tool has to have an ability to collect metrics mentioned in the previous chapter.

4.1 Research of available technologies

There is a lot of performance testing tools that can be used. We only considered ones that provide an option to write simulation scenarios as a code, which is a must-have feature. There are also tools where one can only define the behaviour of client using UI which is highly impractical and unusable for our use case. This is a list of tools worth mentioning:

- `gatling` [1] : open source tool developed in Scala programming language, it is also used for writing simulation scripts.
- `tsung` [2] : open source tool developed in Erlang programming language, it is also used for writing simulation scripts. Besides HTTP, the tool can also be used for performance test of protocols and technologies like: WebDAV, SOAP, PostgreSQL, MySQL, AMQP, MQTT, LDAP and Jabber/XMPP servers.
- `jmeter` [3] : open source tool developed in Java programming language, which is also used for writing simulation scripts. Like tsung mentioned above, it can be used for various protocols and technologies like :SOAP / REST Webservices, FTP, Database via JDBC, LDAP, Message-oriented Middleware (MOM) via JMS, Mail - SMTP(S), POP3(S) and IMAP(S) and so on.
- `locust` [4] : open source tool developed in Python, programming language.

After a comparison of selected tools, it was clear that locust was the most suitable and satisfied our requirements best. It has an active community and it's written in Python - a language we are familiar with, it's easy to learn its syntax and also it's simple to implement a simulation of client behaviour scripts. These are key parameters for writing the tests - new members of the team can edit or write new tests easily.

The locust provides a collection of metrics like average and median response time, request response time distribution out of the box. Reports don't contain apdex, but the tool provides a fairly simple way of extending it so we can implement this functionality.

4.2 Locust implementation of system clients

We implemented locust client classes for all three types of clients that exist in the system. We won't be developing complicated simulations that would consist of a process

[1] `https://github.com/gatling/gatling`
[2] `https://github.com/processone/tsung`
[3] `https://github.com/apache/jmeter`
[4] `https://github.com/locustio/locust`

of unlocking a car, but we will focus on use cases that are most common and that generate a majority of the system load.

- **car sharing user:** Most often, this client is getting data from public endpoints that do not require authentication:
  - parking zones where the user is allowed to return a car to the system
  - locations and information about cars that are available for sharing
  - details for a selected car - fuel level, type, engine and the car model
- **car park administrators:** Requests that always require authentication and are available only for administrators - users with special privileges:
  - detailed information about all the cars (including the ones that are reserved) such as last time car sent request to the system, what is the version of module's hardware and software, etc.
  - information about users, their reservations etc.
  - information for dashboard - how many cars are in the system, number of cars that are not responding, the current number of reservations for the day
- **hardware module in car:** Car module is periodically sending requests with telemetry to the backend. The module also sends a request in a case that event of attaching RFID card to a reader placed on windshield occurs, but it rarely happens and no data are persisted, that is why we omit this type of a request.

Every locust client can implement `on_start` method that is run for every client at the beginning of the performance test. Since car sharing users have to be authenticated only for requests that are rarely used compared to the ones on public endpoints, we will ignore them and `on_start` method can remain empty. For car park administrator we will call login to the system in `on_start` method. This method is the most complicated for the car module. We need to dynamically create a car with every attached entity. It consists of multiple car administrator API calls which can take multiple seconds. Additionally, we are going to send one request from the car, so that we are really sure everything was created correctly. That's why we added an additional build step for creating cars and writing car credentials to file and specifying it as a build artifact. In `on_start` method, cars will just read car credentials.

4.3 Parameters of performance testing

For each run of performance test, we need to specify run parameters, such as a number of clients, time of test run and request frequency. We will operate with run time for every test - first test runs, and runs in process of development would be shorter than runs used for comparison of some specific implementations - for that, we would run each test for one hour to get relevant data. Other parameters were chosen for every client type and test run independently.

4.3.1 Carsharing user

Currently, we have 400 activated users (means that they are able to actually use the service - create reservation) and 800 registered users. While searching for a car, the client is doing approximately one request for API every 3 to 9 seconds.

4.3.2 Car park administrator

At the moment there are 50 people with administrator privileges. We will use this number for every run of the test. We do not expect this number to rise with an increasing number of cars and registered users. That's why we will manipulate more with a number of cars for load tests. If you have admin application opened, it's refreshing data every 10 seconds.

4.3.3 Car module

The car park currently includes 17 cars. But in summer we expect the number to rise to 30. As mentioned before, the module is periodically sending telemetry to the backend. That period depends on whether there is an active reservation in the system. In that case, the car is sending telemetry every 5 seconds, if not the interval is 30 minutes. We will simulate the state in which all cars are reserved. Telemetry data that are sent to processing are randomly generated.

#	Cars in the system	Number of admins	Number of clients
1	17	50	400
2	30	50	600
3	60	50	1200
4	100	50	1800
5	150	50	2800
6	200	50	4000
7	250	50	5000
8	300	50	6000
9	350	50	7000
10	400	50	8000
11	450	50	9000
12	500	50	10000

Table 4.1. Proposed numbers of clients for specific test runs.

4.4 Virtual machine resources utilization

Besides response time, for better understanding of system performance under a certain load, we also used basic metrics from virtual machines such as CPU usage, memory usage, disk IOPS, etc. For metrics collection we used Metricbeat[1] from Elasticsearch monitoring ecosystem mentioned before.

4.5 PostgreSQL metrics

For a better understanding of database performance, we also use a module for PostgreSQL metrics. Most useful metric set for us is statements. It's using `pg_stat_statements`[2] extension, that provides a way for tracking statistics of SQL statements. The extension is gathering data from the time it was enabled. Because of that, we are not able to get statistics for a particular load test. That is why we have to invoke `pg_stat_statements_reset()` function before every performance test

[1] https://www.elastic.co/products/beats/metricbeat
[2] https://www.postgresql.org/docs/current/pgstatstatements.html

execution. In that way, we get statistics for statements that are only executed during the individual test.

We are interested mainly in these attributes of metrics data: `query` - text of a representative statement, `total_time` - total time spent in the statement, in milliseconds

4.6 Metrics visualizations

Since we are using Elasticsearch for gathering all metrics, we can use Kibana - open source data visualization plugin. We created three basic visualizations:

- CPU: Line chart that shows the percentage of CPU time spent in states other than Idle and IOWait.
- Memory: Line chart that shows the percentage of actually used memory.
- SQL queries: Data table with query text and total time spent in the statement.

To obtain information about resources utilization of one needs to look in build logs at the exact start and end time of performance test run of corresponding TeamCity build. The acquired time range can be specified in Kibana.

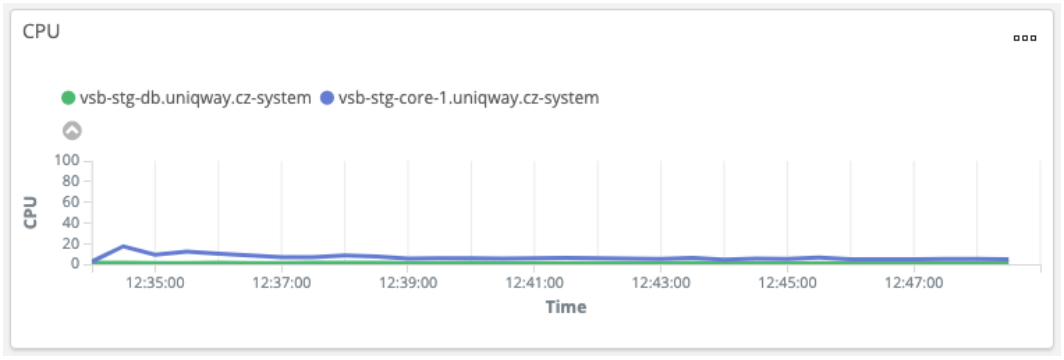


Figure 4.1. CPU usage visualization

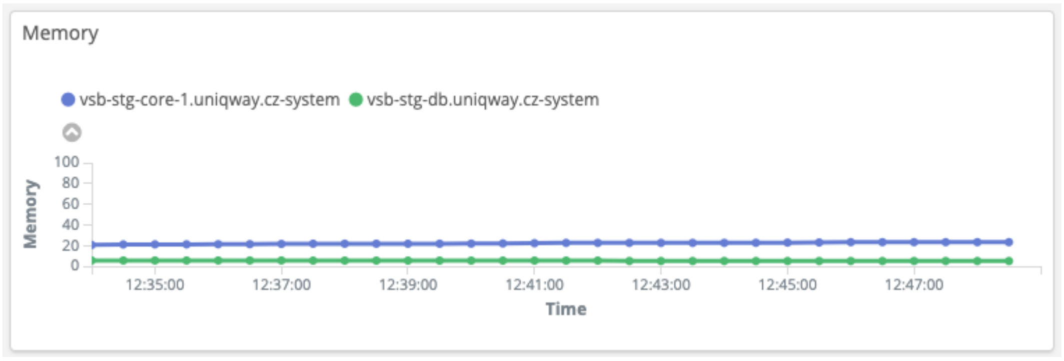
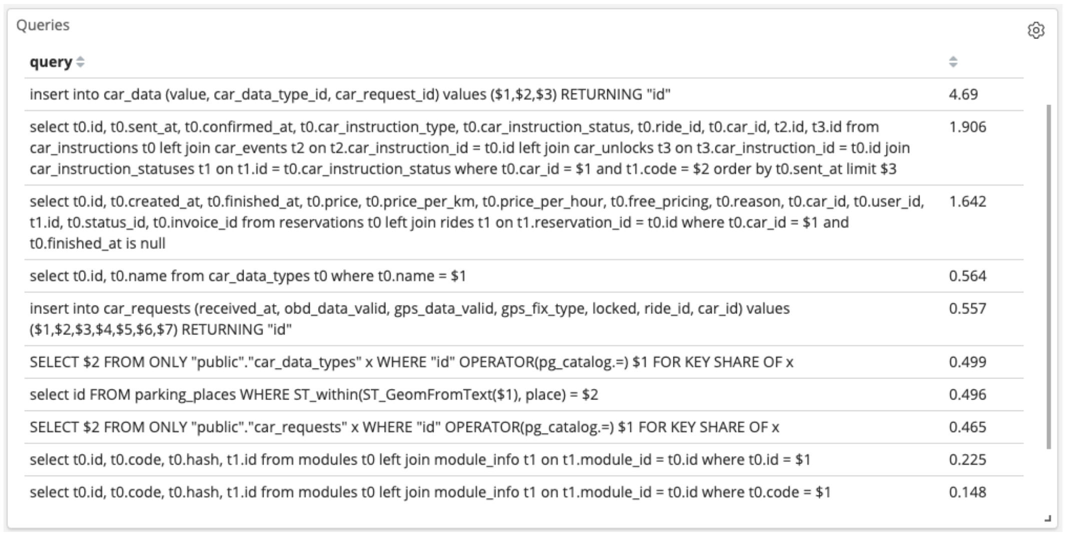


Figure 4.2. Memory usage visualization



Queries

query	
insert into car_data (value, car_data_type_id, car_request_id) values ($1,$2,$3) RETURNING "id"	4.69
select t0.id, t0.sent_at, t0.confirmed_at, t0.car_instruction_type, t0.car_instruction_status, t0.ride_id, t0.car_id, t2.id, t3.id from car_instructions t0 left join car_events t2 on t2.car_instruction_id = t0.id left join car_unlocks t3 on t3.car_instruction_id = t0.id join car_instruction_statuses t1 on t1.id = t0.car_instruction_status where t0.car_id = $1 and t1.code = $2 order by t0.sent_at limit $3	1.906
select t0.id, t0.created_at, t0.finished_at, t0.price, t0.price_per_km, t0.price_per_hour, t0.free_pricing, t0.reason, t0.car_id, t0.user_id, t1.id, t0.status_id, t0.invoice_id from reservations t0 left join rides t1 on t1.reservation_id = t0.id where t0.car_id = $1 and t0.finished_at is null	1.642
select t0.id, t0.name from car_data_types t0 where t0.name = $1	0.564
insert into car_requests (received_at, obd_data_valid, gps_data_valid, gps_fix_type, locked, ride_id, car_id) values ($1,$2,$3,$4,$5,$6,$7) RETURNING "id"	0.557
SELECT $2 FROM ONLY "public"."car_data_types" x WHERE "id" OPERATOR(pg_catalog.=) $1 FOR KEY SHARE OF x	0.499
select id FROM parking_places WHERE ST_within(ST_GeomFromText($1), place) = $2	0.496
SELECT $2 FROM ONLY "public"."car_requests" x WHERE "id" OPERATOR(pg_catalog.=) $1 FOR KEY SHARE OF x	0.465
select t0.id, t0.code, t0.hash, t1.id from modules t0 left join module_info t1 on t1.module_id = t0.id where t0.id = $1	0.225
select t0.id, t0.code, t0.hash, t1.id from modules t0 left join module_info t1 on t1.module_id = t0.id where t0.code = $1	0.148

Figure 4.3. Slow queries visualization

4.7 Environment for running tests

Tests are run from teamcity build agent against staging environment that is the same as production. We also have a build that can sync data from production database to staging.

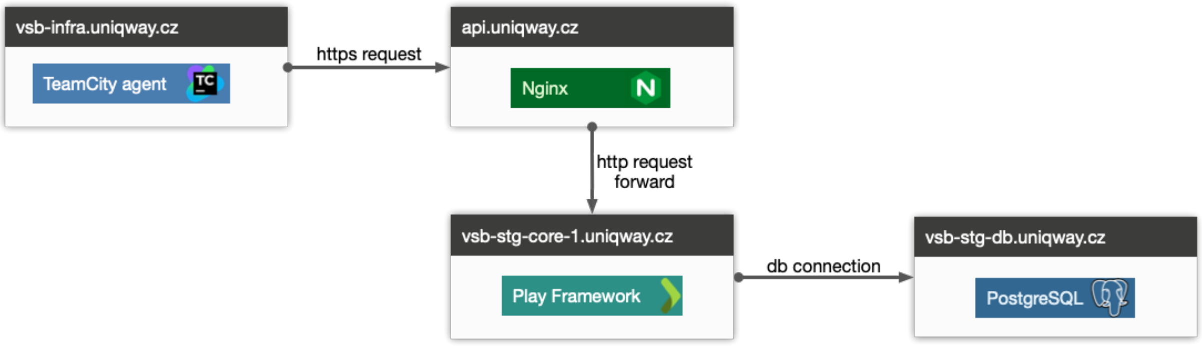


Figure 4.4. Diagram environment for tests.

4.8 Performance testing

4.8.1 Apdex

Apdex is a feature that is missing in locust. Luckily for us, locust offers a defined way to extend its `functionality`[1] using event hooks.

We need to implement apdex class that will be able to compute the apdex index. We started with a test:

```
def test_apdex_correct_computation(self):
    apdex = Apdex(apdex_file_path='apdex',
                  satisfied=1000,
                  tolerated=3000)

    satisfied = [333, 111, 334, 33, 44, 11, 344, 481, 114, 560]
    satisfied_count = len(satisfied)
    tolerated = [2000, 2010, 2799, 2787, 1220]
    tolerated_count = len(tolerated)
    frustrated = [4444, 8984, 14849]
    frustrated_count = len(frustrated) + 2

    for response_time in satisfied + tolerated + frustrated:
        apdex.process_request(response_time)

    apdex.process_failed()
    apdex.process_failed()

    expected_apdex = (satisfied_count + (tolerated_count / 2)) / (
            frustrated_count + satisfied_count + tolerated_count)

    self.assertEqual(expected_apdex, apdex.apdex)
```

- `process_request(response_time)`:
  - method with a response time of request as a parameter
  - will compare time to satisfied and tolerated parameters (set in constructor) and will increase the corresponding count
- `process_failed()`: if the response code is different than 200, it means that request failed and we do consider it as frustrated and increment the corresponding count.
- `write_apdex_to_file()`: we need to propagate computed apdex so we can read the information, we will write the result to a text file as JSON, that we can parse easily

Then we have to define event handlers and bind them to corresponding events.

```
apdex = Apdex(satisfied=os.environ['APDEX_SATISFIED'],
              tolerated=os.environ['APDEX_TOLERATED'],
              apdex_file_path=os.environ['APDEX_FILENAME'])

def apdex_success_handler(request_type, name, response_time):
    apdex.process_request(response_time)

def apdex_fail_handler(request_type, name, response_time, exception):
```

[1] https://docs.locust.io/en/stable/extending-locust.html

```
    apdex.process_failed()

def finish_handler():
    apdex.print()
    apdex.write_apdex_to_file()

events.request_success += apdex_success_handler
events.request_failure += apdex_fail_handler
events.quitting += finish_handler
```

4.8.2 Requests statistics

All values in report form locust performance test run are in milliseconds. Measurement report for requests statistics from the load test is a table where fields for every URL are:

- number of requests
- number of failed requests
- average response time
- minimal response time
- maximal response time
- median response time
- number of requets sent per second

Name	# requests	fails	average (ms)	median (ms)
GET admin/cars	13	0	1244	929
GET admin/dashboard	8	0	2361	2071
GET admin/reservations	13	0	1268	757
GET admin/rides	14	0	992	581

Table 4.2. Example output from load testing of administrator behaviour.

Locust can export reports in CSV format, so output files can be easily parsed. We used this feature and at the end of every test run, CSV files are parsed and sent to mentioned Elasticsearch. Thanks to that, we have easy and practical access to the reports together with all other metrics.

Chapter 5
Identifing scalability bottleneck in existing system

Developed simulation from the previous chapter can be now used to generate a load of various numbers of specific users and to collect key performance metrics data specified in chapter 3.

5.1 Scalability of software systems

Scalability is the ability of a system to properly operate and adapt to rising load, by means of adding resources. The rising load is usually caused by an increased number of clients and it's connected to the growth of popularity of the system.

Applied to our system, it can be interpreted as an ability to process and persist telemetry from car modules with an increasing number of cars and clients while maintaining a tolerated response time of requests from client applications.

Scalability bottleneck is usually caused by part of the system utilizing the resources ineffectively - for example, slow database query or suboptimal algorithm.

There are two types of scaling - horizontal and vertical.

5.1.1 Horizontal scaling

Increasing resources by adding more nodes to the cluster. This type of scaling needs some abstraction for handling various challenges such as load distribution to available nodes, cluster configuration, health-checking of nodes - there is a need for knowing that node became corrupted so no more additional work is assigned to such node)

5.1.2 Vertical scaling

Scaling by adding more resources to existing nodes (virtual machines or hardware servers). Typically increasing number of CPUs, adding memory or disk space. Vertical scaling is usually a more expensive option and it is almost impossible to apply when it comes to adaptation to the load generated by millions of users.

5.2 Results evaluation from the first load test.

To evaluate apdex for client API endpoints, we need to specify time response thresholds for every client type. We are most strict while setting satisfied and tolerated parameters for the car sharing user. In the book Effective Performance Engineering [2], the author is mentioning performance culture in Google. They keep in mind that 40% of users abandon a site when response takes more than 2-3 seconds. That's why we set tolerated time to 2 seconds.

For admins, we will setup apdex parameters to be more tolerant because the user experience is less important for employees. Although we realize that slow response is affecting productivity, the main concern is maintaining admin app usable.

Client app	satisfied (ms)	tolerated (ms)
User	1500	3000
Admin	3000	6000
Module	4500	5000

Table 5.1. Specified thresholds for client applications.

As mentioned before, the module is sending periodically every 5 seconds, when reserved. We are assuming case when all cars are fully utilized - biggest load to the system.

After running first test with current number of cars (17), users (400) and admins (50) for 15 minutes we got following results:

Client app	apdex	# requests	fails	average (ms)	median (ms)
User	0.999	50842	0	39	21
Admin	0.008	1440	0	25644	25000
Module	1	2884	0	59	52

Table 5.2. Report for first performance test run.

We can observe that module and user client types have reasonable Apdex and with a current number of clients and modules system performance is acceptable. For admin it's very different, apdex is 0.008 and that is not acceptable.

5.2.1 Performance evaluation for admin endpoints

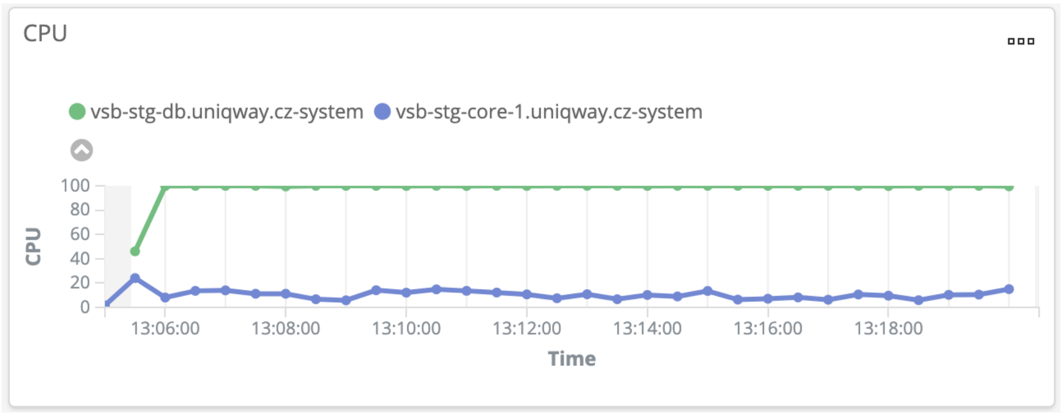
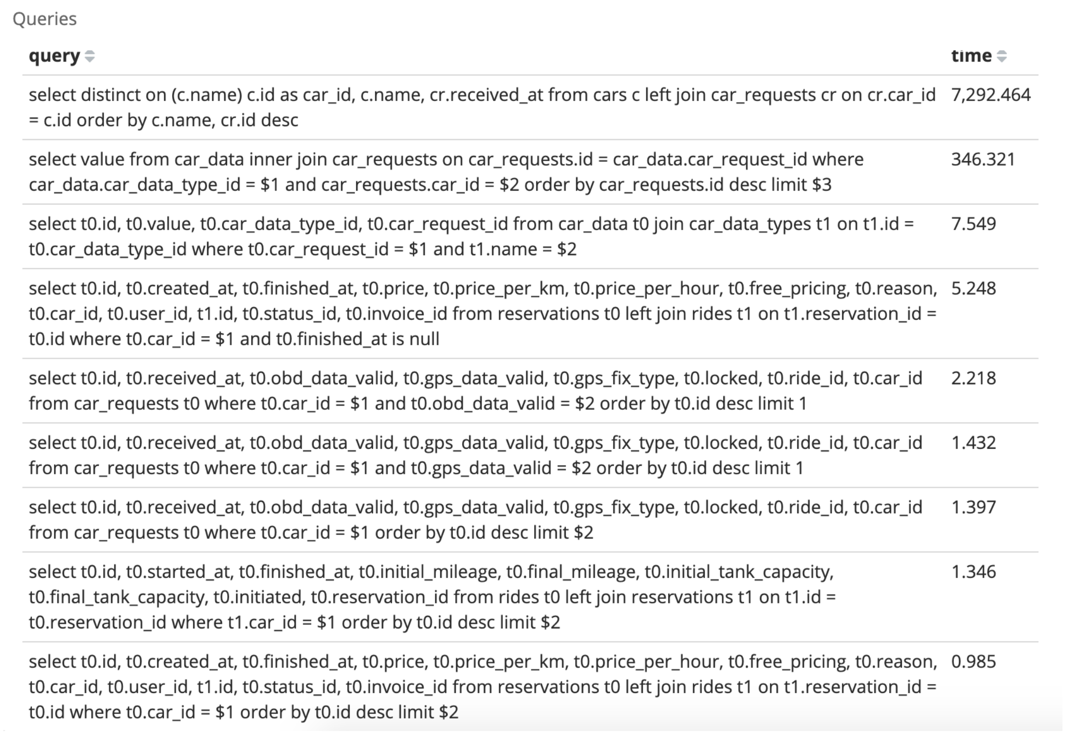


Figure 5.1. CPU usage

From visualization we can see that CPU utilization is at 100% most of the time.



Queries

query	time
select distinct on (c.name) c.id as car_id, c.name, cr.received_at from cars c left join car_requests cr on cr.car_id = c.id order by c.name, cr.id desc	7,292.464
select value from car_data inner join car_requests on car_requests.id = car_data.car_request_id where car_data.car_data_type_id = $1 and car_requests.car_id = $2 order by car_requests.id desc limit $3	346.321
select t0.id, t0.value, t0.car_data_type_id, t0.car_request_id from car_data t0 join car_data_types t1 on t1.id = t0.car_data_type_id where t0.car_request_id = $1 and t1.name = $2	7.549
select t0.id, t0.created_at, t0.finished_at, t0.price, t0.price_per_km, t0.price_per_hour, t0.free_pricing, t0.reason, t0.car_id, t0.user_id, t1.id, t0.status_id, t0.invoice_id from reservations t0 left join rides t1 on t1.reservation_id = t0.id where t0.car_id = $1 and t0.finished_at is null	5.248
select t0.id, t0.received_at, t0.obd_data_valid, t0.gps_data_valid, t0.gps_fix_type, t0.locked, t0.ride_id, t0.car_id from car_requests t0 where t0.car_id = $1 and t0.obd_data_valid = $2 order by t0.id desc limit 1	2.218
select t0.id, t0.received_at, t0.obd_data_valid, t0.gps_data_valid, t0.gps_fix_type, t0.locked, t0.ride_id, t0.car_id from car_requests t0 where t0.car_id = $1 and t0.gps_data_valid = $2 order by t0.id desc limit 1	1.432
select t0.id, t0.received_at, t0.obd_data_valid, t0.gps_data_valid, t0.gps_fix_type, t0.locked, t0.ride_id, t0.car_id from car_requests t0 where t0.car_id = $1 order by t0.id desc limit $2	1.397
select t0.id, t0.started_at, t0.finished_at, t0.initial_mileage, t0.final_mileage, t0.initial_tank_capacity, t0.final_tank_capacity, t0.initiated, t0.reservation_id from rides t0 left join reservations t1 on t1.id = t0.reservation_id where t1.car_id = $1 order by t0.id desc limit $2	1.346
select t0.id, t0.created_at, t0.finished_at, t0.price, t0.price_per_km, t0.price_per_hour, t0.free_pricing, t0.reason, t0.car_id, t0.user_id, t1.id, t0.status_id, t0.invoice_id from reservations t0 left join rides t1 on t1.reservation_id = t0.id where t0.car_id = $1 order by t0.id desc limit $2	0.985

Figure 5.2. Slow queries visualization

From PostgreSQL statements metrics of queries with biggest execution time during a load test, and generated CPU load for database virtual machines, we identified that most of the time is spent on a query that is getting last request times for every car. Next slow query is getting value from `car_data` table for specific `car_data_type`.

5.3 Identification of bottleneck

The hardware module is under active development as any other component of the system and that means that backend has to process and store telemetry with properties that vary. That's why every property of telemetry record is stored in a table as one row and mapped to the corresponding record.

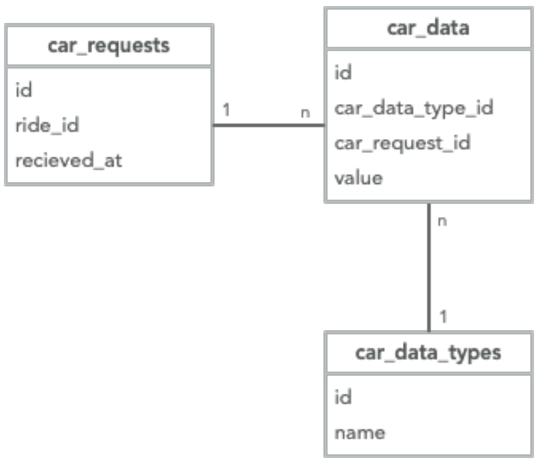


Figure 5.3. Diagram of SQL tables used for telemetry storage.

Currently, we have 21 data types that we need to store for every request for the car.

Basically, every time we need to get latest data for cars (for example location, a timestamp for last time car sent telemetry to serve, etc.), we need to search through `car_requests` and `car_data` tables.

Right now we are not querying any historical data from telemetry, but in future, we will need to. Method for obtaining the latest car data, therefore, needs to be changed, otherwise, we won't be able to achieve reasonable system performance.

Chapter 6
Design and implementation of system changes that lead to better scalability

In the previous chapter, we identified that we are storing all the data in `car_data` and `car_requests` table and access to them is very slow and causes intolerably long response times. For all our current use cases, we need to have access to the latest data from car modules.

6.1 Table for storing only the latest car data

We can significantly reduce the time of querying latest car data by storing them into separate table where instead of inserting, we will just update existing entry for a particular car data type.

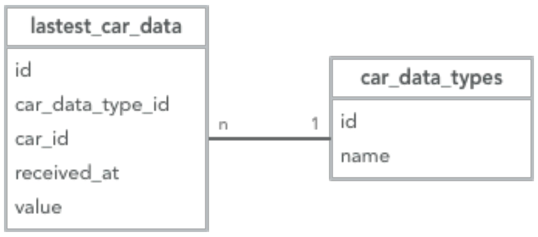


Figure 6.1. Proposal of SQL schema for new table.

6.2 System functionality testing

Before we start replacing getting data from `car_data` table, we will need to have confidence that changes are not going to affect the functionality of the system. Unfortunately, the system is poorly covered with tests, that's why we need to start with writing the tests first.

We decided that first, we will create a high-level test for basic system functionality so that we can be sure that the system is operational and works as expected. This test will have high sensitivity and low specificity - we will know that something is not working but it'll be harder to identify which part of the system is causing the test to fail.

In this test, we create a car with all required entities and create a reservation with a testing user. Then we simulate the opening of the car, the start of ride and movement of the car. During this, we are sending request from admin account and assert if car and ride are always in the state (have values of their attributes) that we expect.

6.2.1 Functionality test scenario

At first, we have to describe a scenario for the test:

1. Initialization: Before the actual testing starts, we need to do steps such as:
   - Login of admin client: We need to be able to interact with the system using admin endpoints. Since admin has access to all of the data that are sent from car modules, we will use this API to get the latest data and compare it to data that we sent to backend during the test.
   - Initialization of car module client: Using admin client we create car module with all required entities using admin API and we will use data like module id, car id, and module hash so we have everything we need to successful module authentication and therefore interaction with the system.
   - Login of user client: We will use it for creating reservation and interaction with the car - locking and unlocking.
2. Car sends initial data to backend
3. User reserves newly created car
4. Car unlock: This step includes user sending request that will store a new entry in `car_instruction` table. In the next step, we will send a request from the car where we will simulate user physically putting his RFID card to a card reader. The system will check if there is pending car instruction, if yes, it will compare RFID card to card of user that has reserved the car. If card numbers match, the system will pass instruction for unlocking to the car in request response. We will also send telemetry from car where `locked` value is set to true, so lock state of car will update in the system.
5. Drive with the car: We created a small array of different car locations that we will send to backend from car module. We will also increment mileage with every request.
6. Car lock: Simpler than unlocking because we don't need to approve instruction for locking the car with using RFID card.
7. Reservation finish: If the car is locked and in a parking zone, user can send a request for finishing the reservation.

This scenario is testing only positive flow, but in the future we can use the same approach to test also behavior of the system in unexpected and more complex situations.

6.3 Test implementation

Since functional test will operate on HTTP level and system simulation is already implemented in Python, it makes sense to use the same programming language. In that way, classes that already have some functionality (login, or car creation for admin user client) can be reused in functional tests.

Example method, where the car is being unlocked and checked if everything proceeded as expected.

```
def __unlock_car(self):
    self.client.car_unlock()
    car_event_response = \
        self.module.send_car_event()
    instruction_id = \
        car_event_response.get('instructionId')
    instruction_code = \
        car_event_response.get('instructionCode')
    self.assertNotEqual(-1, instruction_id)
    self.assertEqual(2, instruction_code)

    self.module.update_instruction(
        instruction_id,
        instruction_status=3)
    unlock_status = \
        self.client.get_unlock_status().get('status')
    self.assertEqual('ok', unlock_status)
```

6.4 Implementation of table for latest car data

The goal is to introduce a database table that will serve for purposes of holding only the most recent data sent from car modules. In that way, data will be accessible without the need for searching through a table filled with historical data.

Implementation is divided into steps. Firstly `evolutions`[1] script needs to be created for the new table.

```
# --- !Ups
create table latest_car_data
(
    id                bigserial    not null,
    car_data_type_id bigint       not null,
    value             varchar(255) not null,
    car_id            bigint,
    received_at       timestamptz  not null,

    constraint pk_latest_car_data primary key (id),
    constraint fk_latest_car_data_car_data_type_id
        foreign key (car_data_type_id)
        references car_data_types (id)
        on delete restrict on update restrict,
    constraint fk_latest_car_data_car_id
        foreign key (car_id)
        references cars (id)
        on delete restrict on update restrict
);

# --- !Downs
drop table if exists latest_car_data cascade;
```

Now model class LatestCarData can be created. It represents the entry of `latest_car_data` SQL table.

```
@Entity
@Table(name = "latest_car_data")
public class LatestCarData extends Model {

    @Id
    @GeneratedValue
    private Long id;
```

[1] https://www.playframework.com/documentation/2.7.x/Evolutions

```
    @ManyToOne
    @JoinColumn(name = "car_data_type_id", nullable = false)
    private CarDataType type;

    @Column(nullable = false)
    private String value;

    @Temporal(TemporalType.TIMESTAMP)
    @DateTimeFormat(pattern = "yyyyMMdd hh:mm:ss")
    @Column(nullable = false, name = "received_at")
    private Date receivedAt;

    @ManyToOne
    @JoinColumn(nullable = false, name = "car_id")
    private Car car;
}
```

We can continue with the implementation of DAO (Data Access Object) for inserting data to SQL table and also for accessing them. Since we are extending the functionality of BaseDao class, we can use methods like find, persist and update out of the box.

```
public class LatestCarDataDao
    extends BaseDao⟨LatestCarData, Long⟩ {
    public LatestCarDataDao() { super(LatestCarData.class); }

    public LatestCarData getLatestDataOfType(Long carId, String type) {
        return finder.query().where()
                .eq("car.id", carId).eq("type.name", type)
                .findOne();
    }
}
```

New service that works with the car data model needs to be implemented. It'll consist of two public methods. A public method for updating the latest data with a responsibility to update an existing entry in the database, or inserting new table row - in case that entry doesn't exist yet. In that case, private method for storing of the entry is called.

```
public LatestCarData update(String dataType,
                            String dataValue,
                            Date receivedAt,
                            Car car) {
    LatestCarData latestCarData =
        findLast(car.getId(), dataType);
    if (latestCarData == null) {
        return store(dataType, dataValue,
                     receivedAt, car);
    }

    latestCarData.setValue(dataValue);
    latestCarData.setReceivedAt(receivedAt);
    latestCarDataDao.update(latestCarData);
    return latestCarData;
}
```

```
private LatestCarData store(String dataType,
                            String dataValue,
                            Date receivedAt,
                            Car car) {
    CarDataType carDataType =
        carDataTypeService.findByName(dataType);
    LatestCarData data = new LatestCarData();
    data.setType(carDataType);
    data.setValue(dataValue);
    data.setReceivedAt(receivedAt);
    data.setCar(car);
    latestCarDataDao.persist(data);
    return data;
}
```

Mechanism for obtaining specific latest car data for specific car and car data type also has to be provided.

```
public LatestCarData find(Long carId,
                         @NotNull String type) {
    return latestCarDataDao.getLatestDataOfType(carId, type);
}
```

After all required framework layers for proper integration of newly created table are created, we need to identify all usages of original service classes that access the table with historical car data, so they can be replaced with the usage of service created in previous steps.

6.5 Usages of public methods in CarDataService and CarRequestService

Both `CarDataService` and `CarRequestService` provide multiple public methods that are querying the latest data from `car_data` and `car_requests` tables.

6.5.1 Usages of CarRequestService public methods

- `findLast(Long carId)`: Method used in CarService for endpoint /admin/cars used for geting data like last request timestamp, gps validity and gps fix type.
- `findLastWithValidGps(Long carId)`: Method that is used just in `CarDataService`
- `findLastWithValidObd(Long carId)`: Method that is used just in `CarDataService`
- `getLast()`: Method that is used just in `DashboardService`

6.5.2 Usages of CarDataService public methods

- `findLastWithFwVersion(Long carId)`: Method that is used just in `CarService`
- `findLastWithHwVersion(Long carId)`: Method that is used just in `CarService`
- `findLast(Long carId, String type)`: Method that is used just in `CarService`
- `findLastWithValidGps(Long carId, String type)`: Method that is used in `CarLocationService` and `ParkingService`
- `findLastWithValidObd(Long carId, String type)`: Method that is used in `CarService`, `CarTankService` and `RideService`

Every one of these methods can be replaced with `find(Long carId, String type)` method from `LatestCarDataService`.

6.6 Experimental performance tests after the changes

We did small tests for 15 minutes. For a current number of cars - 17 and for 100 cars, apdex numbers were okay (for the client it went to 0.892 for 1800 clients). But when we run the test with 300 cars, apdex for module fell to 0.017 with 43 failed requests and median response time 12000 and avg response 13618.

From visualization of slow SQL statements, we can see that

```
select id, value, received_at, car_data_type_id, car_id
    from latest_car_data where car_id = \$1
    and car_data_type_id = \$2;
```

takes most of the time - 218.512 seconds. That query is invoked when we want to find and then update LatestCarData model.

Since it's the most expensive operation now and we have to do it multiple times for just one request from the car it has become our current bottleneck. One way of limiting accesses to the database that is occurring just for obtaining the ID of entry for a specific car and car data type is to cache IDs in memory.

6.7 Caching IDs of model entities.

Caching is a widely used way of optimization in modern applications. Instead of always accessing used data store, data that are frequently queried are loaded into memory. In our case we will use caching for quick access to the IDs of latest car data entries.

6.7.1 Implementation of cache in Play Framework

Play Framework offers `cache implementation`[1] based on open source caching library `Caffeine`[2].

In our case, we need to have quick access to the ID of `LatestCarData` model for a specific car and car data type. That's why we will use string ⟨*car ID*⟩`:`⟨*car data type ID*⟩ as a key for the cache.

We can configure basic cache properties such as initial capacity. This parameter is set accordingly to a number of used car data and number of the car so every ID will be able to fit into cache.

6.7.2 Code changes for cache integration

After adding cache to project dependencies, we are able to inject it into code using standard dependency injection. In class LatestCarDataService we add:

```
private SyncCacheApi cache;

@Inject
public void setCache(SyncCacheApi cache) {
    this.cache = cache;
}
```

[1] https://www.playframework.com/documentation/2.7.x/JavaCache
[2] https://github.com/ben-manes/caffeine/

Now we are able to use its method for updating the latest data. We will need to do a few adjustments. At first, we will check if ID we need to use is aleady in the cache, if yes we are now able to create an object that we will use for database update. If we don't find the desired ID in the cache, we are going to need to look into the database like in the first version of the method. If we get an object from the database, we will add a new entry to cache and proceed. If null object returns from the database, we'll know that this parameter is not persisted yet and we need to store it.

Now we have an object with ID and we can set new values for properties receivedAt and value, then update it in the database.

```
update(String dataType, String dataValue,
        Date receivedAt, Car car) {
    Optional⟨Long⟩ id =
        cache.getOptional(car.getId() + ":" + typeId);
    LatestCarData latestCarData;

    if (!id.isPresent()) {
        latestCarData = find(car.getId(), typeId);
        if (latestCarData == null) {
            return store(typeId, dataValue, receivedAt, car);
        }
        cache.set(car.getId() + ":" + typeId, latestCarData.getId());
    } else {
        latestCarData = new LatestCarData();
        latestCarData.setId(id.get());
    }

    latestCarData.setValue(dataValue);
    latestCarData.setReceivedAt(receivedAt);
    latestCarDataDao.update(latestCarData);
    return latestCarData;
}
```

6.8 Measurements after implementation

We are going to run a set of tests to see if we achieved the desired speedup of saving the latest data to a separate table. Results from last test for 500 cars, 10000 clients and 50 admins:

Client app	apdex	# requests	fails	average (ms)	median (ms)
User	0.891	116777	1242	1963	720
Admin	0.966	5758	0	1807	1700
Module	1	73081	0	581	480

Table 6.1. Performance test run report after cache integration.

Those are fairly satisfactory results and achieved performance is sufficient, but the problem with storing historical data still persists and it has to be solved.

Chapter 7
Method for persisting telemetry data

Currently, we have 6 months worth of data stored in `car_data` table, and we are almost at 25 million entries. A number of rows stored greatly depends on the utilization of cars - as we mentioned before, reserved cars are sending telemetry every 5 seconds, cars available for reservations only once every 30 minutes. In a state where all 500 cars are fully utilized and we are storing only data that the modules are able to gather now (21 entries in `car_data_types` table), we would reach almost 5 billion inserted rows every month. If we consider the performance of querying data from the table in the current state of the project, it's obvious that we have to come up with other solution for storing telemetry from hardware modules.

7.1 NoSQL databases

Today's software systems are expected to handle a huge number of users, process an extensive amount of data and they need to be ready to react and adapt to load that can increase in a short time. Traditional SQL databases are suitable for standard CRUD web applications that require high consistency and have well-structured data. As we mentioned before, they are typically hard to scale horizontally - scaling can get very expensive. To address issues with traditional databases, multiple new systems were developed. As Rick Cattell mentioned in his paper [3], NoSQL databases have properties such as the ability to horizontal scaling, distribution of data to nodes, dynamic attributes set of records, low latency for R/W operations and weaker consistency of data.

7.1.1 ACID and BASE concepts

Transactions of traditional relational databases usually have ACID [4] set of properties:

- Atomicity: If a transaction is atomic, it means that operations in the transaction will either be executed all or none of them. Operations are considered as one unit and changes will be visible only after all updates are done.
- Consistency: All defined constraints must be preserved all the time. If some operations inside transaction violate defined rules, all operations have to be rolled back.
- Isolation: In case that there are concurrent transactions accessing the same data, consistency must be preserved and resulting changes must be exactly the same as if the transactions were executed serially.
- Durability: In case of power failure or another unexpected failure, all operations in a transaction that was successfully executed before has to be persisted to disk.

Contrary to ACID transaction properties, NoSQL databases have BASE[5] philosophy:

- Basically Available: Response to request, but not guaranteed to get a correct result - there can be some changes happening and response can be ̈waiting for changes to complete¨

- Soft state: System state can change over time - eventual consistency can cause that data is moved across the system event without the impact of external factors.
- Eventual consistency: Data/System will eventually become consistent after all changes are done, data propagated to all nodes, etc.

7.1.2 CAP theorem

Eric Brewer formulated CAP theorem[6] in which he declared that distributed data storage can only have 2 of these properties:

- Consistency: Operations for retrieving data always returns the most recent data.
- Availability: Every request returns response - either successful one or an error response.
- Partition tolerance: System remains operational despite a number of failed/delayed messages between nodes.

Mapped to Brewer's theorem and BASE principles, NoSQL databases typically trade consistency for availability or partition tolerance.

7.1.3 Use case of storing telemetry from cars

Hardware module in car has mutliple source of data that can be gathered. It has GPS unit and is connected to on-board diagnostics of a car. It's able to collect data from these sources, create JSON and send HTTP request to backend with the JSON message as request data.

Example of JSON sent to backend:

```
{
    "backupBattery": true,
    "batteryVoltage": 20,
    "beBackBtn": true,
    "fwVersion": "Mar 5 2019 20:22:00",
    "gps": {
        "data": {
            "latitude": 50.069558,
            "longitude": 14.475211,
            "speed": 5
        },
        "gpsFixType": "D",
        "valid": true
    },
    "hwVersion": "ver.2",
    "locked": true,
    "obd": {
        "data": {
            "consumption": 0,
            "fuelLevel": 40,
            "odometer": 150,
            "rpm": 0,
            "speed": 0
        },
        "valid": true
    },
    "tamper": false
```

As we metioned before, hardware unit in car is under active development. Hardware team announced that module will be able to read data from gyroscope, and more informations from on-board diagnostics. For backend it means that we need to count that JSON sent to backend can be changed, fields can and will be added. These additional fields will not be critical for the basic functionality of the system, but we will definitely need to have acceess to them and be able to query them. This requirements fit to category of NoSQL databases with document store data model. Stored documents are indexed and they are semi-structured. Semi-structured data do not have to obey pre-defined structure and their attributes can differ. They are usually stored in XML or JSON format. This type of database fits our use case. We will select two document data stores for comparision.

7.1.4 MongoDB

MongoDB[7] is open source data store developed by MongoDB Inc. The used programming language is C++ and stores JSON-like documents. It supports horizontal scaling, automatic sharding and document distribution over nodes. For work with data it provides query mechanism with dynamic queries and atomic operations on fields.

7.1.5 Elasticsearch

Elasticsearch[8] is not only NoSQL datastore but also a search engine. It's open source project developed by Elastic NV written in Java programming language. Like Mongo DB it supports horizontal scaling, sharding and document distrbution, replication over cluster nodes and uses schema-free JSON documents. Elasticsearch also has additional features like support for full-text searches.

7.2 Integration of selected data stores to the existing system

Both data stores provide Java clients, we will have to test correct behaviour of the clients. Most realistic tests always include actual interaction with system, instead of mocking behaviour of the system.

We choose to use Java library `testcontainers`[1] which provides easy way to run and destroy mechanism for testing common database systems or other dockerized applications. We will describe integration testing for MongoDB and Elasticsearch separately. Tests will consists of data insertion to data store and then using privided mechanism for querying inserted data. In that way we will also have some knowledge that will possibly be useful for next steps of integrating selected data store to the system.

We will implement methods for passing telemetry data to both Elasticsearch and MongoDB. Then we will need to run performance test for both data stores, that's why we decided to control (enable/disable) telemetry passing using configuration file. We created class TelemetryService with method `passTemeletry` where application configuration is parsed and it's decided which data store is used for passing of a telemetry. We used dependency injection to inject service to ModuleService that is handling requests from car module.

[1] https://www.testcontainers.org/

In constructor of TelemetryService we inject Configuration object that is play.api package. Using this object we can read application configuration file `application.conf` where we added new entries for controlling data stores:

```
telemetry {
  mongo {
    enabled = no
    host = localhost
    port = 27017
    databaseName = "teledb"
    collectionName = "teledb"
  }
  elastic {
    enabled = no
    host = localhost
    port = 9200
  }
  sql {
    enabled = yes
  }
}
```

```
@Inject
public TelemetryService(Configuration configuration) {
    isElasticEnabled = (Boolean) configuration
                .getBoolean("telemetry.elastic.enabled").get();
    isMongoEnabled = (Boolean) configuration
                .getBoolean("telemetry.mongo.enabled").get();
    isSqlEnabled = (Boolean) configuration
                .getBoolean("telemetry.sql.enabled").get();
}
```

In method passTemeletry, we are not going to save JSON from a car module exactly like it was received because it would be impossible to map specific requests from a specific ride. We will do data enrichment and we will also add a field for car name and if there is existing registration for cars, we also add user ID, and reservation ID to fields. After that, we will make use of our configuration entries for Elasticsearch and MongoDB and if we enable the data store, the data will call method for data pass of the enabled service.

7.3 MongoDB integration

As mentioned before, we will use testcontainers for integration testing. We created class MongoIntegrationTest, where we implemented a basic test that will use MongoService for passing telemetry entry getting it from MongoDB data store, then compare a few basic data like timestamp and location.

We have to initialize and run container so we can connect our MongoService with it. We will do that before the start of every integration test. On background it'll start a docker container with an image name that is passed to the constructor - in our case official `MongoDB`[1] Docker image. We can get an IP address and port where MongoDB is running and pass it to the constructor of our MongoService.

[1] https://hub.docker.com/_/mongo

We also have to implement `TearDown()` method that is called after every test run, where we need to stop and remove the container that we started before the integration test.

```
private GenericContainer container;

@Before
public void setUp() {
    container = new GenericContainer⟨⟩("mongo");
    String address = container.getContainerIpAddress();
    Integer port = container.getFirstMappedPort();
    mongoService = new MongoService(address, port);
}

@After
public void TearDown() {
    container.stop();
}
```

We will continue with the writing of a test, where we are assuming that MongoService already has implemented methods passTelemetryEntry(String json) and getFirstDocumentForModule(Long moduleId). We are using hardcoded JSON string of telemetry to pass to MongoDB, in second step we are getting few attributes from first record (in our case also the only record - we are always running new clean instance of data store before every test) and compare them to expected values.

```
@Test
public void testSimplePutAndGet() {
    boolean passingSuccessful =
        elasticService.passTelemetryEntry(telemetryJson);
    assertTrue(passingSuccessful);

    Map⟨String, Object⟩ firstDocument =
        elasticService.getFirstTelemetryEntryForModule(1L);

    assertEquals("2019-05-19T00:26:34.735",
        firstDocument.get("timestamp"));
    assertEquals("50.069558,14.475211",
        firstDocument.get("location"));
}
```

This test will fail because we didn't actually implement MongoService class yet. We will start with constructor where we initialize official MongoDB Java client and get collection we are going to use.

```
public MongoService(String address, Integer port) {
    ServerAddress serverAddress =
            new ServerAddress(address, port);

    List⟨ServerAddress⟩ addressList =
            Collections.singletonList(serverAddress);
    MongoClientSettings settings = MongoClientSettings
            .builder()
            .applyToClusterSettings(
                    builder -> builder.hosts(addressList))
```

```
                .build();

        collection = MongoClients
                .create(settings)
                .getDatabase("default")
                .getCollection("default");
    }
```

We have MongoCollection object initialized, so we can continue implementing a method that will insert document represented with JSON string to the collection.

```
public boolean passTelemetryEntry(String json) {
    try {
        collection.insertOne(Document.parse(json));
    } catch (MongoException e) {
        return false;
    }
    return true;
}
```

If we wanted to use console for such insert, the command would look very similiar: `db.default.insertOne({`name:example json`})`

In the next step, we implement a method for querying first record with module ID matching the id passed to the method.

```
public Document getFirstTelemetryEntryForModule(Long moduleId) {
    return collection.find(eq("moduleId", moduleId)).first();
}
```

Now we have passing integration test for MongoDB integration.

7.4 Elasticsearch integration

For Elasticsearch we will proceed the same way as we did with MongoDB. Integration test will look almost the same as it did for MongoDB, with few small changes. We don't have to use `GenericContainer` class, but we can use class ElasticsearchContainer created specifically for Elasticsearch. Another change is that we will operate with `Map`⟨*String, Object*⟩ object returned from a query and not with `Document` object as we did when working with MongoDB.

```
@Test
public void testPassAndGetTelemetry() {

    boolean passingSuccessful =
            elasticService.passTelemetryEntry(telemetryJson);
    assertTrue(passingSuccessful);

    Map⟨String, Object⟩ firstDocument =
            elasticService.getFirstDocumentForModule(1L);

    assertEquals("2019-05-19T00:26:34.735",
            firstDocument.get("timestamp"));
    assertEquals("50.069558,14.475211",
            firstDocument.get("location"));
}
```

We now have failing test, and we can proceed with implementation of class `ElasticSearchService`. In constructor, we initialize Java class `RestHighLevelClient` which is officially provided Java client for Elasticsearch.

```
client = new RestHighLevelClient(
    RestClient.builder(new HttpHost(address, port)));
```

Next, we can proceed and implement method for inserting JSON to data store:

```
public boolean passTelemetryEntry(String json) {
    IndexRequest indexRequest =
            new IndexRequest(indexName)
                    .source(json, XContentType.JSON)
                    .type("_doc");
    try {
        IndexResponse indexResponse =
                client.index(indexRequest, RequestOptions.DEFAULT);
        return indexResponse.status().getStatus() == 201;
    } catch (IOException e) {
        return false;
    }
}
```

Because Elasticsearch also supports full-text searching, mechanism for searching and querying data is more complex compared to MongoDB.

```
public Map⟨String, Object⟩ getFirstDocumentForModule(Long moduleId) {
    SearchRequest searchRequest = new SearchRequest(indexName);
    BoolQueryBuilder query = new BoolQueryBuilder()
            .must(new TermQueryBuilder("moduleId", moduleId));

    SearchSourceBuilder searchSourceBuilder =
        new SearchSourceBuilder();
    searchSourceBuilder.query(query);
    searchSourceBuilder.size(1);
    searchRequest.source(searchSourceBuilder);

    SearchResponse searchResponse;
    SearchHits searchHits;
    SearchHit[] hitArray = new SearchHit[0];

    try {
        searchResponse =
            client.search(searchRequest, RequestOptions.DEFAULT);
        searchHits = searchResponse.getHits();
        hitArray = searchHits.getHits();
    } catch (IOException e) {
        e.printStackTrace();
    }
    return hitArray[0].getSourceAsMap();
}
```

We can now run integration test also for Elasticsearch and it'll pass.

Chapter 8
Empirical evaluation of the existing and proposed changes

So far, there isn't an existing use case for querying historical data from data store even though there are plans to introduce features that will display the data on the presentation layer of the system. For example, showing driving route for previous user's car shares, or allowing the administrators to show the route for a specific ride.

However, it's crucial that the data is saved to the data store and there is a defined way how to query and integrate them to the system while implementing mentioned features. It's also important to have the option to do analytics and report on the data, for marketing purposes and statistics associated with the service, such as the most frequently used parking zones, the most common types of trips or the most frequent routes. The data are also vital for dealing with driving tickets and other potential legal issues associated with the service as a piece of evidence.

At the moment we are working with a scope that is reduced to the ability of the system to process and persist telemetry with an increasing number of cars. There is a question of how many cars really makes sense to simulate during performance tests. There is no point in testing on the scale of thousands of vehicles. Firstly we don't have the infrastructure that could process such load and secondly, in next few years we are aiming to expand to other universities, maybe in other cities like Brno, but so far there are no ambitions to expand internationally. If our project reached the same number of users and car utilization as CAR4WAY, a car sharing for the general public that is already in Brno, we would reach a number of around 500 cars. That is why our performance tests will assume with the gradual growth of car park by 50 vehicles.

Our goal is to empirically evaluate which one of our choosen NoSQL data stores will perfrom better under such load. We will use simulation that we have developed.

8.1 Infrastructure for performance tests

As we did for the performance test before, we will run a performance test from the TeamCity build server. During the run of previous tests, we developed a build chain, that is making a backup from production PostgreSQL database and restores it on the environment we are using for performance testing. After that continues build for running performance test for the existing number of cars - 17, then there is build configuration for creating an additional number of cars needed for the next test and so on. Code snipped from TeamCity configuration:

```
for (config in Configs.all()) {
    val createCars = assureCars(config.numberOfCars,
                        previousConfig.numberOfCars,
                        previousTest)
    val moduleLoadTest = loadTestBuild("module",
                        config.numberOfCars,
                            config.numberOfCars,
                            minutes,
                            createCars)
      buildTypes.add(createCars)
      buildTypes.add(moduleLoadTest)

      previousTest = moduleLoadTest
      previousConfig = config
  }
```

We need to deploy our selected data stores to some virtual machine. In the data center where we are running the whole infrastructure for the project, we deployed an additional virtual machine for performance tests. It has 2 CPUs, 16GB RAM and 50GB of storage. As on every other VM the operating system we are using is CentOS in version 7. We will deploy both MongoDB and Elasticsearch using official Docker images with recommended configuration of containers and tool Docker compose.

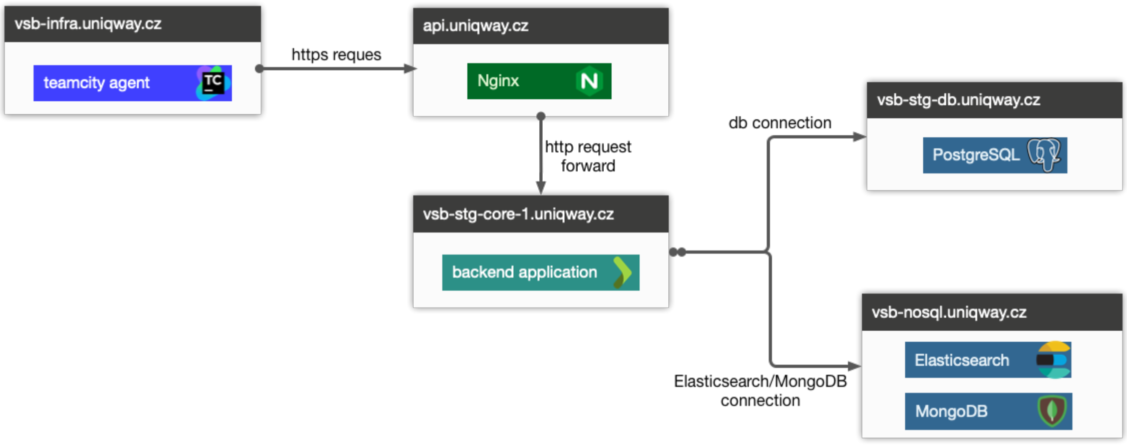


Figure 8.1. Infrastructure for evaluation

Firstly we will deploy MongoDB, after that we will take advantage of configuration file from where we can control all of the data stores (including storing data to original `car_data` table SQL) and will enable passing telemetry to MongoDB with corresponding hostname, port, and authentication credentials, then we can go ahead and run performance tests. After all, tests are done, we will do the same thing for Elasticsearch.

8.2 Result data evaluation

Every run of performance test was one hour long. We run tests with a number of cars in this sequence: 17 (current number of cars in the system), 30 (expected number of cars after summer 2019), 60, 100, 150, 200, 250, 300, 350, 400, 450 and 500.

8.2.1 MongoDB

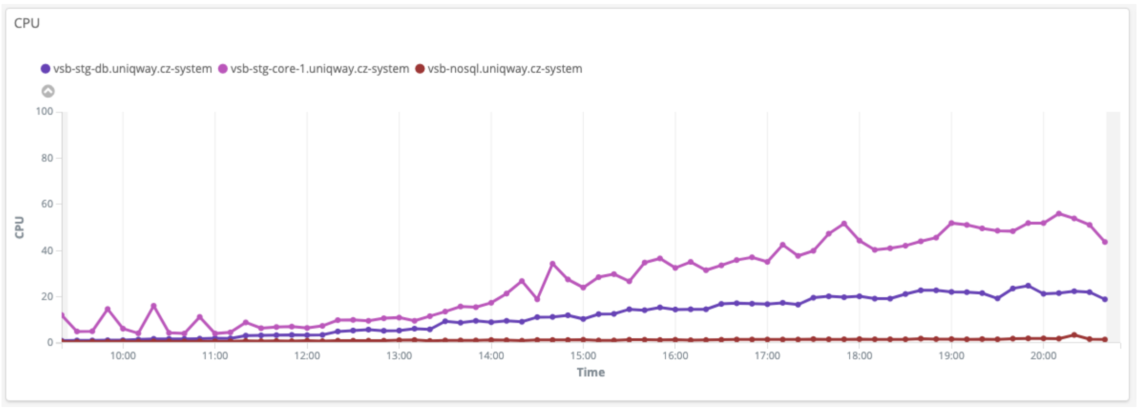


Figure 8.2. CPU utilization - Mongo

From visualizations it can be observed that the CPU of the virtual machine where was MongoDB deployed basically stayed on almost completely unused. On the other hand CPU usage of the machine with backend application gradually grew.

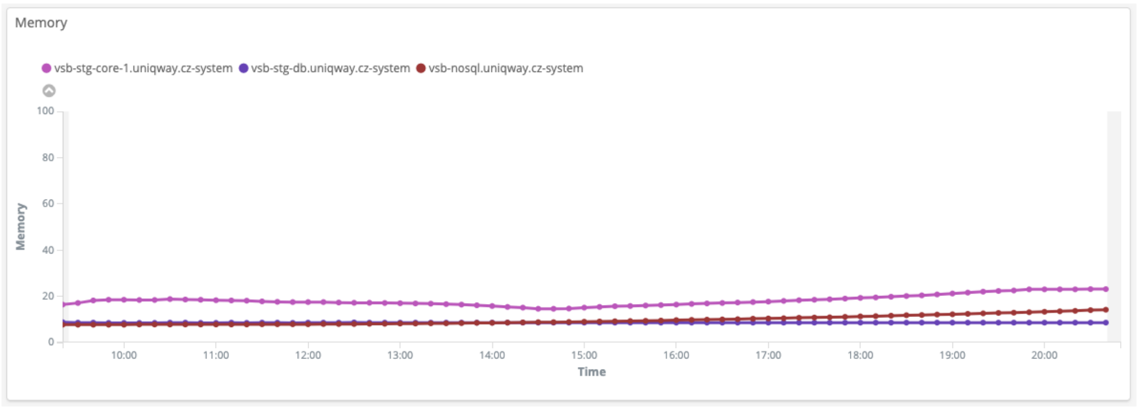


Figure 8.3. Memory utilization - Mongo

Memory utilization on all virtual machines involved in performance test stayed under 40 percent without significant growth.

#	Number of cars	apdex	# requests	fails	average (ms)	median (ms)
1	17	1	11,493	0	74	47
2	30	1	20,362	0	48	44
3	60	1	40,591	0	52	44
4	100	1	67,416	0	56	45
5	150	1	99,867	0	112	51
6	200	1	130,515	0	209	60
7	250	1	160,548	0	290	76
8	300	1	189,265	0	375	120
9	350	1	217,444	0	456	200
10	400	1	244,884	0	532	330
11	450	1	271,399	0	613	510
12	500	0.999	298,625	1	661	600

Table 8.1. Results for all performance test runs of MongoDB

Except for last performance test that generated largest system load, apdex stayed on 1 - which means that all requests response were satisfied and both average and median time stayed under 0.7 second, which is a satisfactory result.

8.2.2 Elasticsearch

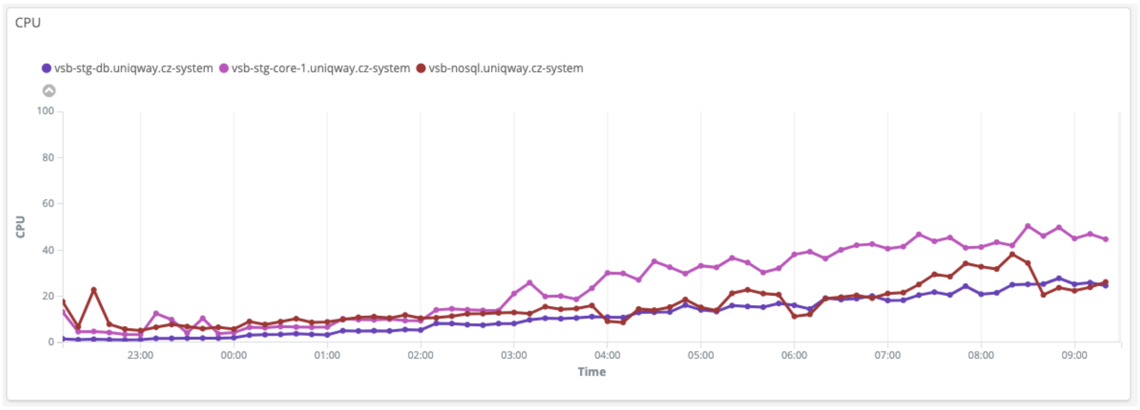


Figure 8.4. CPU utilization - Elasticsearch

Graphical representation of CPU utilization for Elasticsearch indicates a similar pattern of growth for a virtual machine for backend application. Contrary to the test of MongoDB, for VM were Elasticsearch is deployed increased CPU usage of can be observed. This can be caused by the fact that Elasticsears is also a full-text search engine and it's using CPU power for indexing for all documents that were sent there.

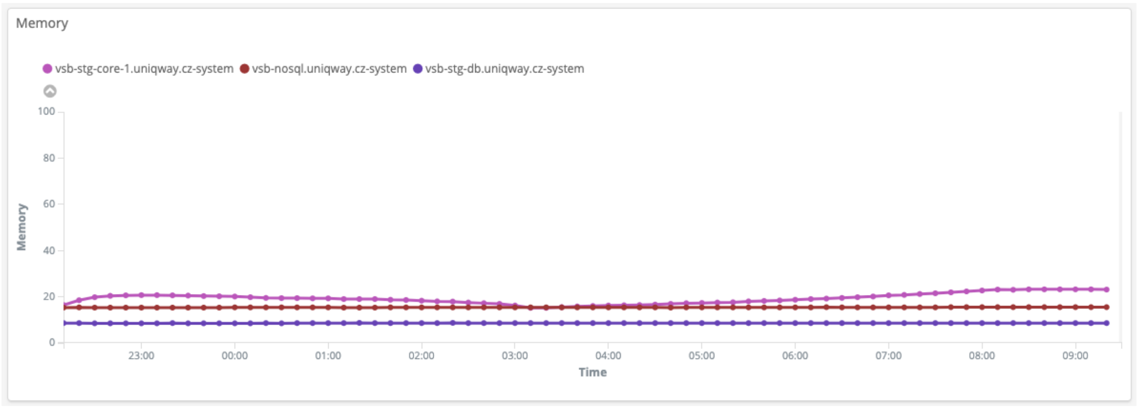


Figure 8.5. Memory utilization - Elasticsearch

In terms of memory usage, the situation is very similar performance tests of MongoDB - no significant growth occurred.

#	Number of cars	apdex	# requests	fails	average (ms)	median (ms)
1	17	1	11,537	0	54	51
2	30	1	20,357	0	51	47
3	60	1	40,580	0	54	47
4	100	1	67,341	0	60	47
5	150	1	100,281	0	93	51
6	200	0.999	131,125	0	181	58
7	250	0.999	161,294	0	261	76
8	300	1	191,936	0	295	110
9	350	0.998	218,035	0	431	180
10	400	1	249,103	0	426	190
11	450	1	278,210	0	465	250
12	500	1	303,048	0	575	440

Table 8.2. Results for all performance test runs of Elasticsearch

Apdex is not always 1, but no HTTP request failure occurred. Based on that we can say that some of the request response times exceeded trash hold of 5000 milliseconds.

8.2.3 Comparison of response times

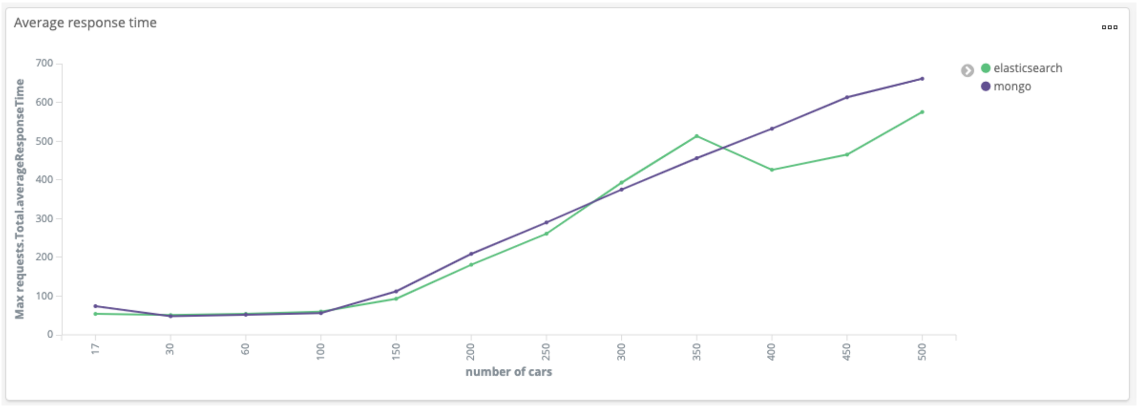


Figure 8.6. Average response time

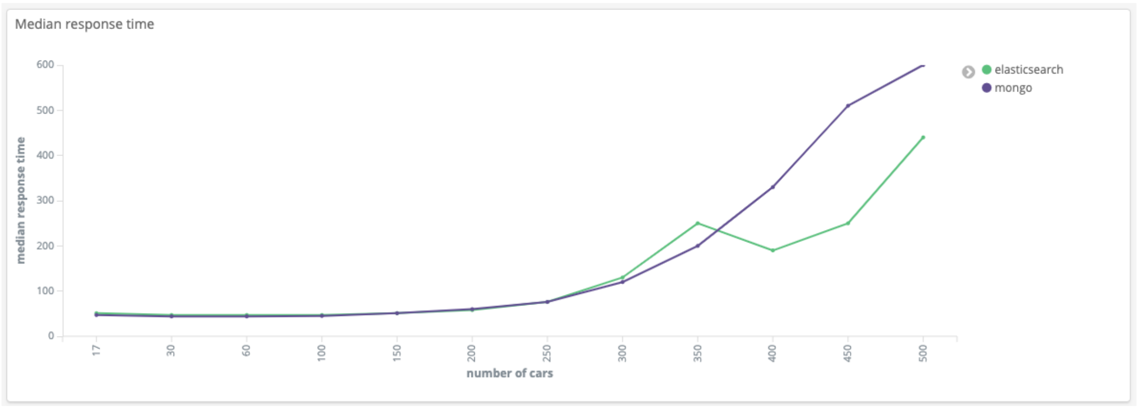


Figure 8.7. Median response time

The same pattern can be seen for both average and median response time. Elasticsearch performed slightly better when a number of cars was over 350.

8.3 Conclusion

Both selected data stores were able to process the telemetry during simulation without any major problems and performance issues. They could both be successfully integrated into the existing system. MongoDB data store was more efficient in terms of resources usage and overall system integration, especially querying mechanism seemed simpler and more straightforward compared to Elasticsearch.

Chapter 9
Discuss results and propose further improvements of the system

During the early stages of work on the thesis, the system was practically not operational under the load of 300 cars. Database queries were so inefficient that client application for car park administrators was unusable even with s small number of simultaneous users. We identified the biggest scalability bottleneck and implemented changes that resulted in significant performance improvements.

The testing platform that we built also brings significant value to the project and its potential expansion. We developed well defined and automated process for performance testing that can be reused and new use cases can be added. The project developers can also continue with its development and increase its functionality. The platform also helped to build and maintain performance culture within the whole team. One of the already mentioned notes from Google's talk about this issue says: "There is no "I" in performance. A performance culture is a team sport.".

One of the steps that would ensure sufficient performance and therefore better scalability, is to set the thresholds and build performance testing into the release process and pipeline. This would prevent the introduction of possible scalability bottlenecks, that would be otherwise merged and eventually deployed to the production system.

Besides Java unit tests, the system was lacking more complex testing that would cover the overall functionality of basic features. It was necessary to implement such tests, in order to unlock the ability to safely make changes to the existing system. We implemented REST API changes that will contribute to the overall reliability of the system and will add a certain level of confidence to developers before deploying the new release to production. We understand that the system is still inadequately tested and although we contributed to the quality assurance of the system, it was done only in scale necessary for the completion of this thesis.

In the age of big data with a huge emphasis on targeted commercials and tailored experience for users, a database like ours could be useful for conducting research by other students or projects. Information about the usage of vehicles could prove useful in many fields and it could be used to further develop the service.

9.1 Further improvements

Performance tests we run were on the scale of hundreds of vehicles. In case that we would need to scale up to thousands or even tens of thousands, the situation would be dramatically different and it would have an impact on several things. We will discuss the impact since it'll be a good way to describe further possible improvements that would be resulting in even better overall scalability of the system.

9.1.1 Caching

As we found out during performance evaluations, accesses to the database are really time-consuming. One of the reasons why is that JVM driver for a database is synchronous, so we can't take advantage of backend framework that is asynchronous. We need to access the database for practically every HTTP request that is sent to the system. That is why it would really ease up the load if we would cache more data from the database. For example, we could cache current locations of the vehicles and parking zones - these are data that are retrieved most often. And since cars are sending the telemetry only once every 30 minutes, they are quite static and don't change very often.

In case we had to deal with load from hundreds of thousands and even millions of users, we would definitely need to scale up count of nodes used for running backend application - in that case, we would also need to change technology used for caching - because used cache, based on Caffeine is not distributed and every instance of the backend application would need to load data to its own cache memory. For distributed caching we could use for example open source in-memory data store `redis`[1]. There is already existing plugin[2] for the integration of redis with Play Framework. This plugin also has an asynchronous variant that could use the benefits of the mentioned architecture of the framework.

9.1.2 Infrastructure provider

Our whole infrastructure is running in a data center in Ostrava and even for a change of CPUs number for some virtual machine there has to be a contract amendment and even such a simple change can take several days or even weeks. This system would be very inefficient and would need to be changed in case we would have to dynamically react to a significant user base growth.

We could follow the trends and migrate to a cloud solution such as Amazon Web Services or Google Cloud platform. In the cloud, we could scale nodes in much more dynamic fashion, even implement features like auto-scaling where we could dynamically add or terminate nodes after detection of significant load increase or decrease. However, with such features come challenges such as load balancing. A lot of problems like that are already solved in platforms for container orchestration such as `Kubernetes`[3]. We are already deploying backend application using Docker, that's one of the reasons why we should reach for such solution.

9.1.3 Separation of backend application

In the current situation, our backend application is handling requests for car sharing users, car park administrators, and car modules. There are several disadvantages connected to traditional monolith architecture. One of them is lack of ability to deploy changes connected only to clients or car modules. At the moment we have to deploy all parts of the system at once. One of the consequences of the separation of the system logic that is handling requests from modules to separately deployable unit would be the ability to not only deploy but also scale that part of the system separately. It would be especially useful in situations where there is big portion of concurrent car shares and a lot of cars are sending telemetry every 5 seconds instead of every 30 minutes.

[1] https://redis.io/
[2] https://github.com/KarelCemus/play-redis
[3] https://kubernetes.io/

References

[1] Fleura Bardhi, and Giana M Eckhardt. Access-based consumption: The case of car sharing. *Journal of consumer research*. 2012, 39 (4), 881–898.

[2] Shane Evans Todd DeCapua. *Effective Performance Engineering*. O'Reilly Media, Inc., 2016. ISBN 9781491950869.

[3] Rick Cattell. Scalable SQL and NoSQL data stores. *Acm Sigmod Record*. 2011, 39 (4), 12–27.

[4] Theo Haerder, and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*. 1983, 15 (4), 287–317.

[5] Werner Vogels. Eventually consistent. *Communications of the ACM*. 2009, 52 (1), 40–44.

[6] Eric Brewer. CAP Twelve years Later: how the. *Computer*. 2012, (2), 23–29.

[7] Kristina Chodorow. *MongoDB: The Definitive Guide, 2nd Edition*. O'Reilly Media, Inc., 2013. ISBN 9781449344689.

[8] Clinton Gormley Zachary Tong. *Elasticsearch: The Definitive Guide*. O'Reilly Media, Inc., 2015. ISBN 9781449358549.