FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Ilda Balliu

Implementation of a user-centered visualization platform for stream data

Department of Software and Data Engineering

Supervisor of the master thesis: Mgr. Martin Nečaský, Ph.D.

Study programme: Software and Data Engineering

Specialization: Software Engineering

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In..... date..... Ilda Balliu

signature

This thesis is without a doubt the result of the hard work and support of many people that have been by my side through this journey.

It has been a tough two years and I couldn't have made it without my parents and my sisters who believed in me and kept supporting me since my first day in Prague, always pushing me to go forward even when I only wanted to go back home.

I would also like to express my gratitude to my supervisor Mgr. Martin Nečaský, Ph.D. for his guidance and patience over the last few months.

Lastly, I would like to thank my team at SAP Concur, especially my manager Carolina Perea Orozco and my mentor Fabrizio Carannante, for giving me the opportunity to be part of this project and for helping me grow professionally ever since I joined.

This thesis is dedicated to my little nephew, Tian.

Title: Implementation of a user-centered visualization platform for stream data

Author: Ilda Balliu

Department: Department of Software Engineering

Supervisor of the master thesis: Mgr. Martin Nečaský, Ph.D, Department of Software Engineering

Abstract: With the complexity increase of enterprise solutions, the need to monitor and maintain them increases with it. SAP Concur offers various services and applications across different environments and data centers. For all these applications and the services underneath, there are different Application Performance Management (APM) tools in place for monitoring them. However, from an incident management point of view, in case of a problem it is time consuming and non-efficient to go through different tools in order to identify the issue. This thesis proposes a solution for a custom and centralized APM which gathers metrics and raw data from multiple sources and visualizes them in real-time in a unified health dashboard called Pulse. In order to fit this solution to the needs of service managers and product owners, Pulse will go through different phases of usability tests and after each phase, new requirements will be implemented and tested again until there is a final design that fits the needs of target users.

Keywords: stream data, visualization, usability

Contents

1. Introduction

Being a large-scale enterprise means that the need to monitor and control the applications is also at a large scale. In case there is an incident caused by buggy code, unexpected load or hardware failure in any of the parts of a certain application, it needs to be identified and fixed as soon as possible to minimize the effect on the end user experience. SAP Concur is a Software-as-a-Service (SaaS) company providing travel and expense management services to businesses (SAP Concur, 2020). Having over 50 locations around the world, Concur offers its products to hundreds of businesses, including the public sector such as government organizations and institutions to help them with travelling and expense management. Therefore it is crucial to constantly monitor the services and handle possible incidents.

This thesis presents the implementation of a centralized internal tool called Pulse which helps manage and monitor customer-facing services and applications in real time. Pulse allows users to see real-time health metrics for the services and applications that they need to monitor. The metrics are collected from different sources, including other external Application Monitoring Management (APM) applications that are already in use in some of the teams. The main purpose is to build a tool where users with different technical skills are able to see which part of an application is faulty and needs some attention.

At the moment of writing of the thesis, this tool is still in development and given that it aims to fulfill primarily the needs of incident managers, software engineers and also higher management within the company, it is necessary to follow a user-centered design (UCD) approach for building it so that the tool contains the necessary amount of data and visualizations for them to able to identify potential problems and issues.

1.1. Related works

This thesis combines several research topics in the current software engineering landscape including data visualization, stream data processing and user-centered design approach. In the following sections, we provide an overview of recent work on these concepts.

1.1.1. Data visualization

"Data is the new oil" is a phrase first coined by Clive Humby in 2006 while he helped Tesco build the Clubcard loyalty program for its customers (Bridle, 2018). However, both data and oil need proper governance and management in order to gain significant benefits from them (Firican, 2019). This means that in order to get value from the data you obtain, you need to manage them properly. Among other meanings, this can also be interpreted as being able to visualize your data in such way that you can obtain valuable information from it. The greatest value of a picture is when it forces us to notice what we never expected to see (Tukey, 1977). Corporations, government bodies, and scientists are realizing the challenges and, moreover, opportunities that exist with effective utilization of the extraordinary volumes, large varieties, and great velocity of data they govern. However, to unlock the potential contained within these deep wells of ones and zeros application of techniques to explore and convey the key insights is required. Through visualization, we are seeking to portray data in ways that allow us to see it in a new light, to visually observe patterns, exceptions, and the possible stories that sit behind its raw state. This is about considering

I		II		III		IV	
x	y	x	y	x	y	x	y
10.0	8.04	10.0	9.14	10.0	7.46	8.0	6.58
8.0	6.95	8.0	8.14	8.0	6.77	8.0	5.76
13.0	7.58	13.0	8.74	13.0	12.74	8.0	7.71
9.0	8.81	9.0	8.77	9.0	7.11	8.0	8.84
11.0	8.33	11.0	9.26	11.0	7.81	8.0	8.47
14.0	9.96	14.0	8.10	14.0	8.84	8.0	7.04
6.0	7.24	6.0	6.13	6.0	6.08	8.0	5.25
4.0	4.26	4.0	3.10	4.0	5.39	19.0	12.50
12.0	10.84	12.0	9.13	12.0	8.15	8.0	5.56
7.0	4.82	7.0	7.26	7.0	6.42	8.0	7.91
5.0	5.68	5.0	4.74	5.0	5.73	8.0	6.89

Table 1: Anscombe's quartet sets of data

visualization as a tool for discovery (Kirk, et al., 2016), which suggests that visualization is to be considered a powerful tool that can yield many benefits. The Anscombe's quartet (Ascombe, 1973) is an example that

demonstrates how much more value can be gained by looking at your data from a different perspective. The experiment involves four sets of data, each having almost identical statistical properties including mean, variance, and correlation.

Looking at Table 1 above, we would not be able to notice any patterns or anything else of interest, except for the fourth set having a sequence of eights. However, when visualized in graphs such as in Figure 1, we are able to notice that there are in fact interesting behaviors of these sets. Starting from left to right, the first graph shows a similar trend of the values going up and down in the same fashion, followed by the second graph where the values follow the shape of a curve and it does not have a normal distribution. In the third chart there does not seem to be any particularity in behavior, the curve follows a linear distribution except for one outlier of y, and the last graph shows that one outlier is enough to produce a high correlation coefficient. The aim of this experiment was to show that it is much easier to draw conclusions on the presence (or lack thereof) of patterns through a proper visualization rather than looking at raw data.

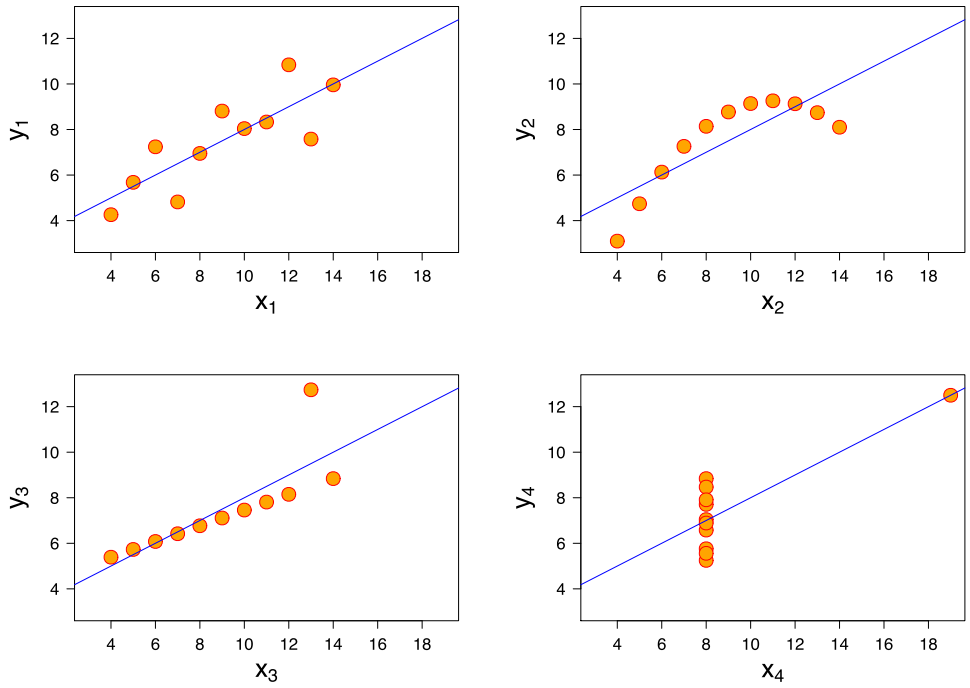

Figure 1: Anscombe's quartet visualization

Source: (Ascombe, 1973), https://commons.wikimedia.org/w/index.php?curid=9838454

1.1.2. Stream data processing

In a traditional application let's say web based, that performs any simple task such as retrieving and displaying information accordingly, the application probably would not have the need to display information unless instructed to. And that can be triggered by a button click, a route change, a selection etc., requiring user interaction. On the other hand, consider an application that monitors the stock markets. Having real-time data is crucial to perform any transaction or take any financial decisions. The users would not want to click repeatedly or take some other action to have real-time data displayed all the time. So, there is a need for some applications to provide the user with real-time data and update them whenever there is new information.

A data stream can be defined as a countable infinite sequence of elements used to represent data that is made available over time (Margara, et al., 2018). However, stream (i.e., event) processing is not a new topic, the roots of it can be traced back to the era of active databases (Adaikkalavan, et al., 2006). According to Dayarathna, it can be divided into two main areas called Stream processing and Complex Event Processing (CEP). The first area, Stream (i.e., event) processing supports many kinds of continuous analytics such as filter, aggregation, enrichment, classification, joining, etc. CEP on the other hand uses patterns over sequences of simple events to detect and report composite events. The boundaries between CEP and stream processing are not always clear (Dayarathna, et al., 2017).

A question that arises here is when do we start categorizing an application as a real-time system? In *Practical Real-Time Data Processing and Analytics* (Saxena, et al., 2017), such applications are defined as software applications that are able to consume, process, and generate results very close to real-time; that is, the lapse between the time the event occurred to the time results arrived is very small, an order of a few nanoseconds to at most a couple of seconds In *Software Engineering* (Sommerville, 2016), it is suggested that the behavior of a real-time system can be defined by listing the stimuli received by the system, the associated responses, and the time at which the response must be produced. Such stimuli can be periodic stimuli and aperiodic stimuli. Periodic means that they occur at predicted time intervals.

Aperiodic stimuli on the other hand, occur irregularly and unpredictably and are usually signaled, using the computer's interrupt mechanism. These stimuli come from the sources where you need to retrieve data, a server infrastructure, a network, a sensor and so on. A real-time system has to respond to stimuli that occur at different times. Therefore, the system architecture needs to be organized in such way so that, as soon as a stimulus is received, control is transferred to the correct handler. This idea is also supported in the paper *Architecture for Analysis of Streaming Data* (Hoque, et al., 2018), where the authors introduce a layered architecture consisting of a combination of microservice and instances of the publish-subscribe pattern, spread over multiple layers.

1.1.3. User-centered design

The term "user-centered design" was coined in Donald A. Norman's research laboratory at the University of California, San Diego (Norman, 1986). In his extensive work on design thinking, he outlines that in UCD the work starts with understanding people's needs and capabilities and the goal is to devise solutions for those needs, making sure that the end results are understandable, affordable, and, most of all, effective. Furthermore, in his paper *People-centered (not tech-driven) design* (Norman, 2018), he goes deeper into the topic by claiming that today technologies are no longer built with humans in mind and technology is deemed more important than people. He also raises a challenge for the future, a future where we need to come up with ways to ensure that our technologies are designed with people in mind, more humane, more collaborative, and more beneficial to the needs of people, societies, and humanity. According to Le (Le, 2017), UCD is an optimistic approach to invent new solutions. It starts with human beings and ends with the answers that are tailored to their individual needs. Whether you build an application for your customers or an internal tool, the goal is still the same: satisfying the requirements of your users regarding features of a product, task, goals, user flows, etc. In UCD this is done by firstly understanding them and their needs and then coming up with solutions that satisfy them (Le, 2017). Karat defines UCD as an iterative process whose goal is the development of usable systems, achieved through involvement of potential users of a system in system design (Karat, 1996).

The definitions of UCD and the theory behind reinforce the same idea: design with humans in mind. It provides a common language for scientists, stakeholders and end users. For example, Lunar Rover Mission of NASA integrated user-centered design techniques. NASA also benefits from user interviews, user observations in context, and wireframing (Treseler, 2016). But to achieve it there are a few guidelines to be followed in order to gain benefit from it. Karat in his paper *Evolving the scope of user-centered design,* also suggests considering UCD an adequate label under which to continue to gather our knowledge of how to develop usable systems. It captures a commitment the usability community supports: you must involve users in system design while leaving fairly open how this is accomplished (Karat, 1997).

On his website[1], Norman lists these four fundamental principles of human-centered design and applications (Norman, 2019):

1. Ensuring that we solve the core, root issues, not just the problem as presented to us (which is often the symptom, not the cause)

2. Focusing on people.

3. Taking a systems point of view, realizing that most complications result from the interdependencies of the multiple parts.

4. Continually testing and refining our proposals, ensuring they truly meet the needs of the people for whom they are intended.

On the paper *Key principles for user-centered systems design* (Gulliksen, et al., 2003) the authors argue that due to lack of an agreed upon definition, there is room for erroneous practices while trying to implement a UCD system. They present a list of 12 key principles for the adoption of a user-centered development process in an attempt to create a definition for UCD.

[1] https://jnd.org/

1. User focus – the goals of the activity, the work domain or context of use, the users' goals, tasks and needs should early guide the development

2. Active user involvement – representative users should actively participate, early and continuously throughout the entire development process and throughout the system lifecycle

3. Evolutionary systems development – the systems development should be both iterative and incremental

4. Simple design representations – the design must be represented in such ways that it can be easily understood by users and all other stakeholders

5. Prototyping – early and continuously, prototypes should be used to visualize and evaluate ideas and design solutions in cooperation with the end users

6. Evaluate use in context – baselined usability goals and design criteria should control the development

7. Explicit and conscious design activities – the development process should contain dedicated design activities

8. A professional attitude – the development process should be performed by effective multidisciplinary teams

9. Usability champion – usability experts should be involved early and continuously throughout the development lifecycle

10. Holistic design – all aspects that influence the future use situation should be developed in parallel

11. Processes customization – the UCSD process must be specified, adapted and/or implemented locally in each organization

12. A user-centered attitude should always be established.

However, authors do agree that implementing UCD is a trial and error process until you have fully defined requirements and specifications. You need to be comfortable with failure, restarting, adapting to change and learning from the previous mistakes.

1.2. Thesis structure

To describe how Pulse was designed, built and tested over time, this thesis is split into 6 following chapters.

Chapter 2: Defining Requirements
This chapter is an analysis on the initial requirements of Pulse set out by the stakeholders and the gathering process.

Chapter 3: Design Process
This chapter describes design process of the system, having the initial system requirements and the changes to those requirements after a few iterations involving the stakeholders and the engineering team.

Chapter 4: Functionality
Here we go through the main functionalities that Pulse offers and how they serve to the end users.

Chapter 5: Implementation Overview
The fourth chapter is an implementation overview from the system architecture to the back end and front end applications as well as a detailed look on the communication between different entities of the system.

Chapter 6: Visualization of Stream Data
This chapter goes deeper in to the implementation by focusing on the visualization, firstly by describing the types of data we are receiving and then a brief introduction to React JS as the chosen visualization framework and Crossbar as the networking platform. Also, here we describe how the implementation of the real-time communication was achieved and how we built components to support the received data.

Chapter 7: User testing

The sixth chapter focuses on the usability of the tool and the reflections on the design process after several iterations of usability testing with potential users.

Chapter 8: Conclusions and future plans

On the last chapter, we describe some of the main conclusions drawn from the work done and future plans of extension and improvement of the tool.

2. Defining Requirements

The first stimulus to build Pulse came from higher management in the Cloud Services department at Concur as a response to the difficulties that mainly Service Management and Critical Incident Management teams were facing with handling incidents and especially those of high priority. These incidents of high priority, called P1, are those that directly affect the users and prevent them from accessing a certain part of the application or performing an action in the intended way.

The main problem that these teams identified was the difficulty of quickly identifying the root cause of the problem. Considering that an application has many services underneath, it takes time to understand which service is causing the problem, whether it is caused by a bug in the code, an unexpected load of the system, database unavailability, a third-party service or even a hardware failure. However, since not only technical teams are involved in case of a P1 incident but also communication or customer service teams and higher management in escalated cases, it is often difficult to translate into non-technical terms what is the problem and keep customers updated on the issue. Therefore, there was a need of a unified user-friendly health dashboard to monitor the applications and services.

2.1. Gathering process

In order to collect and define the initial requirements of Pulse, several interviewing and brainstorming sessions were conducted with representatives from Cloud Services department managers and team members from Service Management, Critical Incident Management, Logging, Quality Assurance, Container Ecosystem, Database Administration, Customer Support and Customer Performance Engineering team which is the engineering team building the application.

However, this thesis is focusing on how the web application (i.e. user interface of Pulse) was designed and built, so we will present here requirements pertaining to it and only general system requirements that are relevant to the context.

During these sessions, two types of users were defined: users with normal access and users with privileged access. Normal users include all users who will use the tool strictly for looking at the health metrics of different applications and services, while users with privileged access include service owners who can control what should be displayed in Pulse by creating custom alerting rules for services and application.

Requirements for users with normal access:

1. Users should be able to login using the corporate login identity provider. (Okta[2]).

2. Users should be able to access the application in different datacenters, to check health metrics of services deployed in different environments or regions.

3. Users should be able to navigate to a dashboard for Infrastructure metrics.

    3.1. The Infrastructure dashboard should allow the user to navigate to applications > role types of application > pools of role type > hosts of pool > host metrics.

    3.2. Each level should visualize its elements with the proper visualization component.

    3.3. Users should be able to see health scores (aggregation of metrics values based on rules created by service owners, however the users don't necessarily know how this was calculated) as well as the proper color in each visualization component in any navigation level under Infrastructure: (application, role type, pool and host) indicating the component at a certain level is in healthy, warning or critical state.

    3.4. Users should be able to see basic information regarding the pools and the hosts inside it: number of available nodes, number of

[2] https://www.okta.com/

connections, if it is active and enabled.

4. Users should be able to see host health metrics in the Infrastructure dashboard, when navigating the host metrics level.

    4.1. Users should be able to see metrics defined by service owners for each host.

    4.2. Users should be able to select the timeframe of the metrics shown for each host.

    4.3. Users should be able to tell whether these metrics are healthy or not at a glance.

5. Users should be able to navigate to a dashboard for Database metrics (only Couchbase[3] for the time being).

    5.1. Users should be able to navigate to Couchbase database clusters (if any) and see the nodes and buckets for each of them.

    5.2. Users should be able to see metrics in the visualization component for Couchbase databases on each level of navigation (cluster, node, bucket).

    5.3. Each Couchbase cluster, node or bucket visualization component has the appropriate coloring to indicate if it is in healthy, warning or critical state.

6. Users should be able to understand at any depth level of navigation in the Infrastructure or Database dashboard, which one of the components is vulnerable at the moment, based on the color of the visualization component (red = critical, yellow = warning, green = healthy).

Requirements for users with privileged access:

[3] https://www.couchbase.com/

1. These users should have all the accessing rights as a normal user and furthermore should be able to create rules and alerting definitions per metric and per service.

    1.1. By creating rules, privileged users can set thresholds for which a metric is considered healthy, warning or critical for each application and each role type or service belonging to it.

    1.2. The user should be able to filter metrics by origins and set thresholds for the metrics coming from different origins (if there are any).

Requirements for the system

1. The system shall be able to allow users to log in using the Okta identity provider.

2. The system shall be able to retrieve users' rights and permissions for accessing applications and services.

3. The system shall be able to allow the user to retrieve all Infrastructure and Database components that they have rights to, based on their authentication details.

4. The system shall allow the user to create custom rules for metrics of a certain application only if they have rights to.

5. The system shall be able to retrieve metrics for each component under Infrastructure dashboard (applications, role types, pools, hosts) and Database dashboard from the Application Programming Interface (API) s of New Relic[4], Elasticsearch[5], F5[6], RMS[7], Couchbase.

    5.1. To define the metrics to be collected, the corresponding teams that monitor the services shall be consulted after a first research on what

[4] https://newrelic.com/

[5] https://www.elastic.co/

[6] https://www.f5.com/

[7] RMS is an internal tool of Concur for resource management.

is available from the APIs.

6. The system shall be able to show a visual representation of each application under Infrastructure (Expense, Travel, Invoice... etc.) and a visualization of all their corresponding components.

7. The system shall use RMS and Compute's[8] metadata to build components and their logical relationships.

8. The user interface shall display aggregated infrastructure and database metrics per each component.

9. The system shall calculate an overall health score for each component, allowing to quickly identify faulty components in the datacenter using custom rules.

10. The visualization of each component shall allow each service owner to see the metrics that they deem important for identifying problems and faulty components.

11. The system shall allow service owners to create rules and restrictions for the metrics and services by choosing an application, the roletype and as many metrics as are available and setting thresholds for each metric.

12. The system shall reflect restrictions and rules created by each service owner.

2.2. Analysis and decisions

These consultations resulted in having initial requirements clear enough to start working on a prototype for the application. However, given that the tool needs to serve to several teams at the same time, and there are teams or members with different levels of technical knowledge, participants agreed that the application needs to be as user-friendly as possible but at the same time should allow the user to have the necessary amount of information available to identify problems. To tackle this aspect, we decided to use the

[8] Compute is an internal tool to manage servers in different data centers of Concur.

UCD approach for building the application. After each major implementation or change in the design, there will need to be a new consultation with the stakeholders in order to decide whether it is appropriate, necessary or it needs to be changed further. These consultations will also involve usability tests with several participants in order to determine whether the design of the UI matches the needs of the users.

Furthermore, service owners and incident management teams are those who decide which metrics or aspects of a service are important to them, they know better what steps should be followed in case of an incident, what they need to check first and so on. Having this input from the teams is crucial to building the application in a way that it fulfills its purpose by providing an easy way to solve incidents faster.

Another decision taken was to set aside at the moment such requirements as user login and showing content only for components that the user is allowed to and has access rights. The requirements pertaining to this are to be implemented at a later stage when the application is more well defined and has taken its shape in terms of implementation and usability.

3. Design Process

After gathering the initial requirements from the stakeholders, the engineering team responsible for Pulse had to prepare firstly a proof of concept (POC) prototype of the application. The main purpose of this first prototype was to further define the initial requirements, avoid any confusion or ambiguities about the features and functionalities as well as to get feedback from other teams which are in fact the potential users of Pulse. Here we will firstly describe the system architecture and high-level decisions for more context and then focus on the design of the frontend application.

3.1. Pulse POC

The first POC prototype for Pulse consists of first determining the implementation units of the system in order to build an initial architecture, deciding what are the responsibilities of each implementation unit, choosing the technologies to be used for each smaller part and designing an initial UI for the first demo with the stakeholders.

The first decision to be make was splitting the system into different implementation units. According to the initial requirements, the application needs to gather data from several sources. These data needs to first stored in a database for the UI to retrieve them and for the aggregations to be computed. To facilitate the communication of the UI with the backend logic, there is another backend unit serving as a REST API, which can also be used outside Pulse from other teams which need to obtain metrics and other data. This way, there are four major implementation units:

Auto-discovery module is written in Python[9] language and collects data using the API of Elasticsearch, New Relic, Prometheus, F5, Compute and RMS for:

- Collecting information about the logical relationships between applications, role types, pools and hosts as well as between clusters, buckets and nodes.

[9] https://www.python.org/

- Collecting metrics for each component level.
- Creating and running scheduled jobs to collect information from the mentioned sources.
- Aggregating the received metrics and calculating health scores based on rules set by service owners or using default system rules.
- Storing the data in the database.

Backend module is also written in Python using the Pyramid[10] framework and is built as a REST API of https endpoints which are publicly available within the corporate environments and allow the UI to retrieve and send data from/to the database.

Database module: the chosen database of the application is MongoDB due to the fact that it is a document-oriented NoSQL database and allows for each metric entry to be entered as a JSON document, it offers powerful querying and it is very fast. The data retrieved from auto-discovery module are saved in the corresponding collections in Mongo and the backend module manipulates them in cooperation with the front-end module.

Frontend module (UI) is built using JavaScript and the React[11] library and creates a user-friendly interface for the Pulse application with all the necessary visualization components.

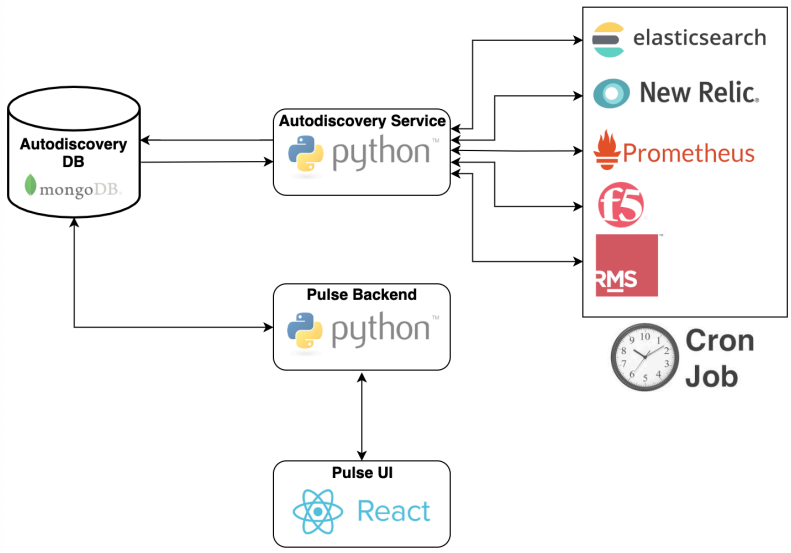

Figure 2: Pulse modules decomposition

[10] https://trypyramid.com/

[11] https://reactjs.org/

3.2. Visualization components design

As mentioned in section 3.1., the UI module is build using JavaScript and the React library. On this phase, it was decided to not jump to building the full UI yet, but first create a skeleton and the visualization components and put them into a small internal library and further on use them as necessary. This helps in developing reusable code not only within the project, but also within the enterprise. To showcase the built components and create an internal components library Storybook[12] was used, which is an interface development environment and playground for UI components that allows to develop UI components in isolation, defining their behavior and testing without being concerned about application dependencies or requirements.

Considering that Pulse is a health dashboard, it was decided to follow and Admin Single-Page-Application[13] (SPA) approach in the design: A foldable sidebar for main navigation, header to display user information and other global features and the rest is main content, shown in the skeleton in Figure 3.

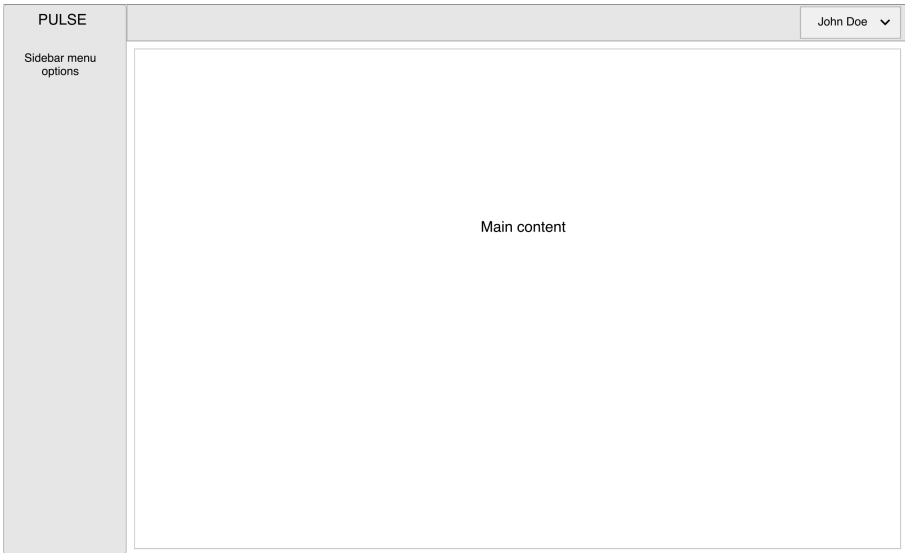

Figure 3: Pulse web application skeleton

Another design decision to be taken at the point was how to visualize each component under Infrastructure and Database dashboards. They both have several elements inside, each with their own particularities. Under

[12] https://storybook.js.org/

[13] https://en.wikipedia.org/wiki/Single-page_application

Infrastructure we have first the applications level, each application has its own role types, each role type has its pools and each pool has a list of hosts. Each pool should have basic information regarding the hosts inside it and the hosts are the elements for which we are collecting proper metrics. For each host we should show a detailed metrics view. However, there is an aggregation of the elements' health from the lower level to the top, in order to see from the applications level for example, which application is currently having problems or faulty components. As we navigate inside that application, it becomes clearer where is the issue. On the other hand, in the Database dashboard, we have fewer elements inside. Each Couchbase cluster has its own hosts and buckets. But we are collecting metrics from the REST Couchbase API for each of those levels and we need to show metrics for each level of navigation. Also, the aggregation to define a health score and visualize it, is not done from down up anymore, but one the same level.

3.2.1. Infrastructure dashboard

The first components to be designed were those under Infrastructure. Each element on any level from applications to the list of hosts falls under a component called a *score card* containing a health score, name of the element and the proper coloring to indicate its health. Figure 4 shows the first designs of the score cards using a 0.5 score threshold to determine the color and the three possible cases of the card look and feel. The score of each element is represented in a percentage circle.

Figure 4: Infrastructure score cards

The purpose of this design is to allow the user to see in one look which of the elements under any level of navigation (application/role type/pool/host) are having issues and how severe is that issue. The visualization is not concerned with how this score was calculated, as it

is done using the custom rules that the privileged users have previously defined, and it is simply retrieved from the API for each element.

As stated in requirements for the pools level, the UI should display some basic information regarding these pools. Figure 5 below shows how this *score card* looks. It includes the name of the pool and the score percentage circle as well as indicators if the pool is enabled or available, numbers of host nodes inside and number of connections.

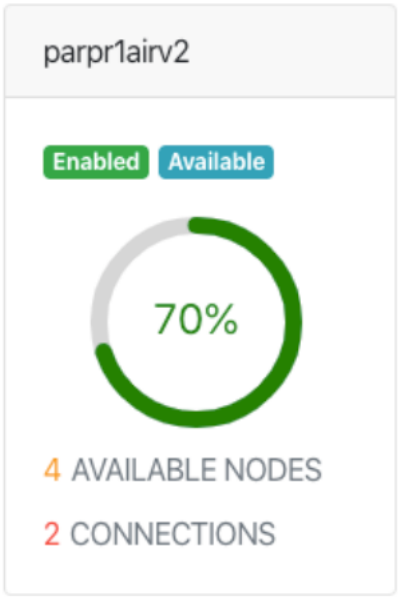

Figure 5: Pool score card

Another point in the requirements that was addressed, was creating a dedicated view for host metrics. Here the user is able to see each of the metrics in a chart in order to look at the trends or any other behavior

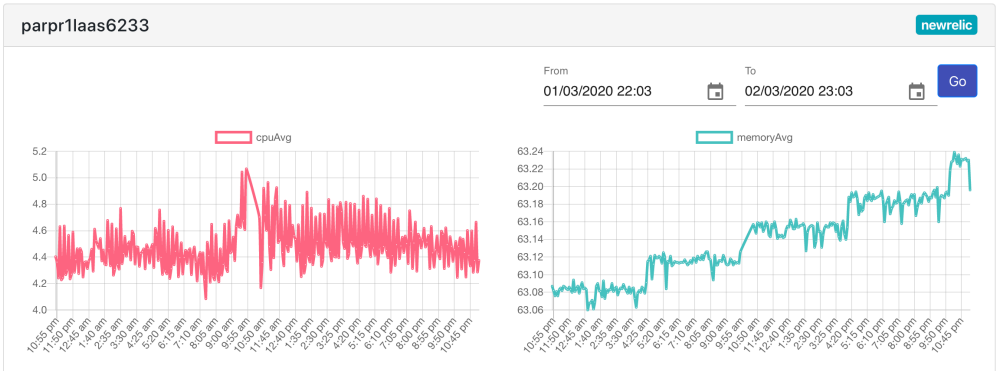

Figure 6: Host metrics

patterns for each host. There is a possibility to change the time range and feed the chart more data as desired. For a demonstration purpose, the CPU average usage and memory average usage metrics were chosen, but the view will render a chart for any other metric provided.

3.2.2. Database dashboard

The Database dashboard on the other hand, has a slightly different flow. At this stage, only Couchbase was required in order to build a visualization concept and consult it with Database Administration teams to add further details. Couchbase servers are configured in clusters and each of them has its own nodes and buckets. We are collecting metrics at cluster, node and bucket level and aggregating the health of each of them. Here we also have a similar visualization to the score card which we will call *cluster card.* Figure 7 below shows three clusters in an 'unknown' health state (grey), healthy (green) and unhealthy (red). If a cluster is grey, there either are no metrics for it at the moment, or there are no rules specified how to interpret and aggregate this metrics.

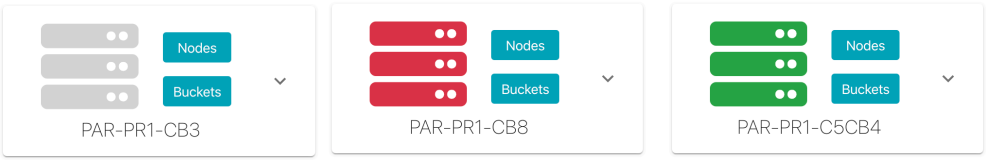

Figure 7: Cluster cards

Each card contains the name of the cluster, the icon colored based on the result of the metrics aggregation of that metric and two buttons that

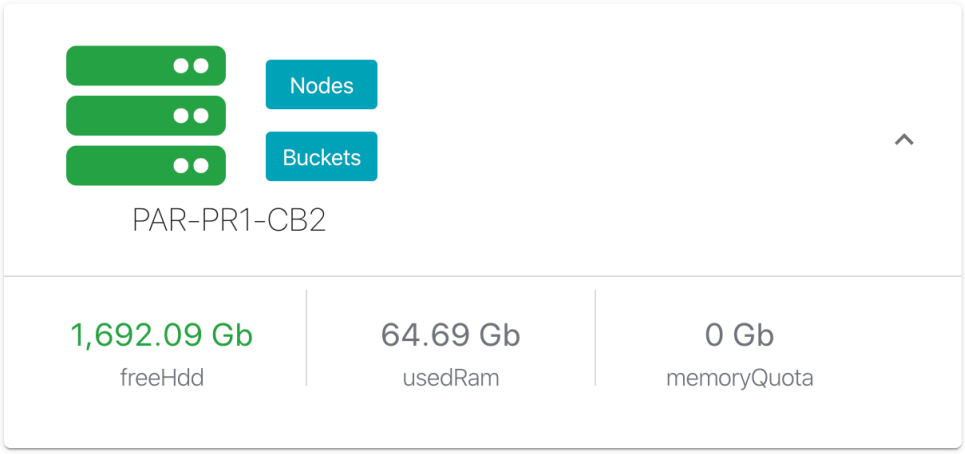

Figure 8: Expanded cluster card

will navigate the user to the buckets and nodes of each cluster. The cluster card is expandable in order to show the metrics. The purpose of this is to avoid any "noise" in the UI by having metrics displayed all the time given that there can be a lot of clusters in the same environment. An expanded cluster card is shown in Figure 8.

To check the nodes and buckets in any cluster, the user needs to click on the respective buttons and the design for node and bucket cards is similar to the cluster cards, except for the buttons. Figure 9 is an example of the bucket cards.

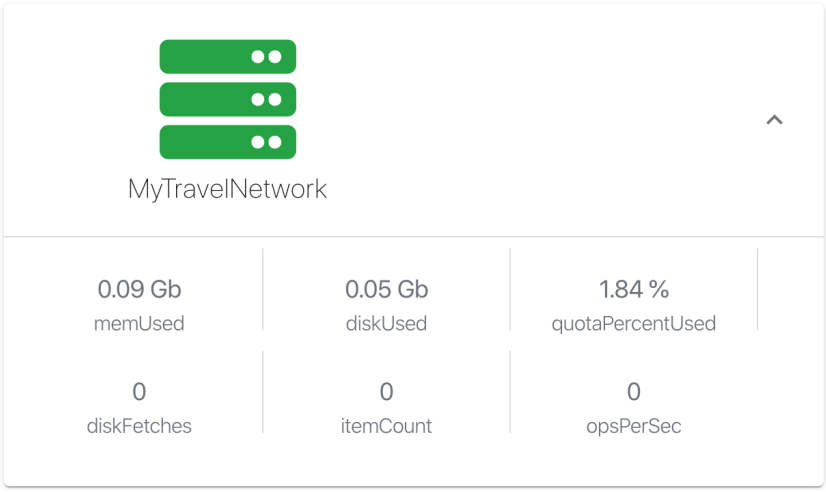

Figure 9: Bucket card

3.2.3. Create rules wizard

In order for the collected metrics and the visualizations to make sense and to bring real value to the users, there needs to be some input from the service owners and incident managers so that the tool is customized up to the last metric. This means that they have the possibility to create their own rules of when a metric is considered healthy, critical or it's close to being critical. The design at this stage supports creating rules for any metric of any application under the infrastructure Dashboard for which health scores are calculated. We will not go into detail about the formula of this calculation however it is a result of consultations with the service owners based on what is more important to them. For example, if they deem the memory average usage metric more important than the CPU average usage, then memory average has a bigger weight

on the calculation. However, normal users are not allowed to create rules, as they are only using the rules that the service owners and incident managers set for a certain application, knowing the product in detail.

The wizard to create rules is an important process that fundamentally helps Pulse in the background to achieve its purpose. Therefore, it is split into small steps in order to allow the privileged users create them easily.

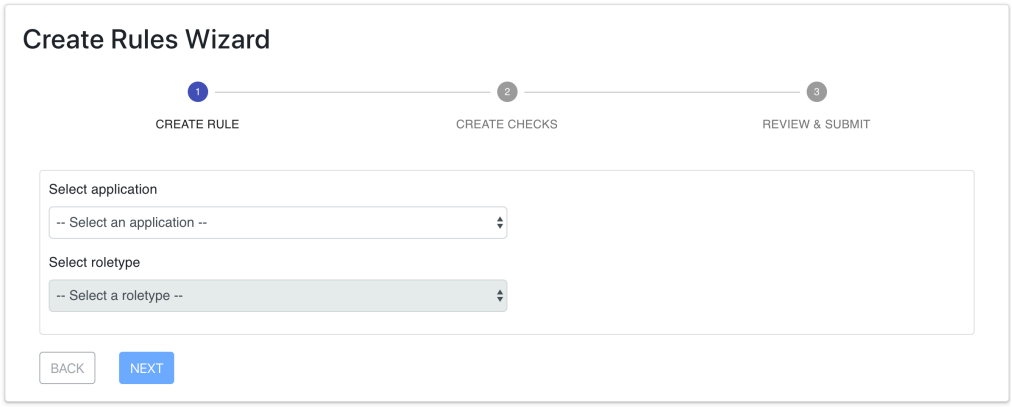

Figure 10: Create rules wizard: Step 1

In the first step, users can choose the application from a selection list of all available applications and then the role type from the list of all role types for that application. After both are chosen, the user can go to the next step by clicking on the *NEXT* button which will be enabled. In the second step, users can define the rules for any metric that they want that is available for the selected application and role type.

Figure 11: Create rules wizard: Step 2

The user first needs to choose the origin for which they want to apply rules to the metrics, currently this metrics can be New Relic, New Relic Insight which is a sub product of New Relic and F5. After origin selection, they can choose the metric from the list of metrics being collected from that origin and set the thresholds for that metric. They can add as many metrics as there are available for all origins at the same time. The thresholds selection is a slider element that can be dragged to the desired value by clicking on the balloon with the number. On the third step (Figure 12), users can do a sanity check of the rules they created and if they are satisfied, they can save the results by clicking on the *FINISH* button. After saving the rules, they will receive a pop-up message notifying the successful submission of the results.

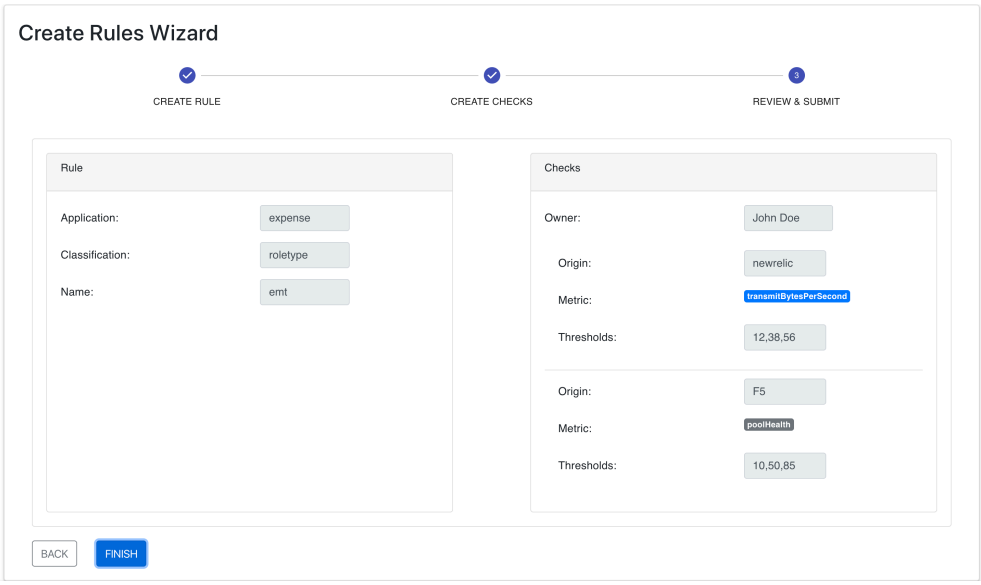

Figure 12: Create rules wizard: Step 3

3.3. Stakeholders review

On the initial prototype of the POC described in section 3.1. and the designs described in section 3.2. were ready, another brainstorming and interviewing session was conducted with the stakeholders including at least one representative from each team with an interest in the tool and higher management. This was to ensure that the requirements so far were fulfilled, and the application was in the right path and that all stakeholders are in the same page going forward.

Before jumping into the design decisions and implementation details, some of the attendees participated in a usability testing session. There was a brief explanation on the existing prototype and its current limitations regarding the use of mocked data imitating the API responses and some instructions for navigation inside the interface. First, participants were asked to freely share their thoughts at any moment about what they saw in the interface, from landing to the initial screen till the end of the tasks in the test.

The tasks that the participants were asked to perform:

1. Check the health status of the ***Expense*** application.
2. Check the alive connections of the ***emt*** role type pools.
3. Check the memory average usage of the ***parpr1emt603*** host for the past hour.
4. Check the health status of the ***PAR-PR1-CB2*** cluster.
5. Check if there are any problematic nodes in the ***PAR-PR1-CB2*** cluster.
6. Check if there are any problematic buckets in the ***PAR-PR1-CB2*** cluster.
7. Create a rule for the ***memory average*** metric in the ***expense*** application, set three thresholds.

These tasks aimed to make each user go through all the levels of navigation on both dashboards but also to match them to real scenarios that they might encounter in their everyday work so that any flaws or possible improvements could come to light. The seventh task is only given to participants that will have access to this view and will be able to create rules for any of the applications and metrics.

This usability test proved to be very productive in terms of identifying existing issues and adding new features to Pulse. The users pointed out that both the Infrastructure and Database dashboard indeed allow them to quickly identify which of the elements in them presents issues by the colors and the scores and this was true also for users with not such advanced technical knowledge or users who do not know the elements in their implementation details. The navigation between levels and dashboards is simple and doesn't leave much room for confusion as the views allows you to see your location within the application and has easy access to previous views in case you want to go back. None of the participants had any

problems while completing the tasks. However, there were remarks about allowing the user more freedom for example in the hosts metrics view, they are able to choose a timeframe for when the metrics should be displayed, but there is a common date picker for all metrics. Some of the participants, especially those that handle incidents argued that it is indeed more useful to have a common date picker, as they want to see how the metrics are behaving at the same timeframe in order to identify potential problems and would take more time to set a timeframe separately for each metric when there are more than two metrics (at this stage they were only shown two metrics for each host).

Another valid point made for the host metrics view is that the metric values in the chart are not updated unless the user changes the timeframe or reloads the page. These are metrics that change often and it is crucial for them to see the latest data all the time. And this is an important requirement added to Pulse, as from now the UI should be designed in such a way that displays real-time data and after a discussion it was decided to apply the same logic to both dashboards, the data is updated whenever there is new information from the sources where data is being collected from.

The Database dashboard design was also received well, but since there are also Memcached clusters, it is necessary to differ between them in the UI and have different views for each under Couchbase. However, Memcached clusters should also have access to their nodes and buckets and show the corresponding metrics. The MySQL database should also be added under the Database dashboard. There should be metrics shown for each host of this database.

Another major change to be made to the interface is shifting all dashboards under a common menu which will be Assets. Under this menu will be the existing dashboards and a new one for Applications, which differs from Infrastructure in the sense that it is split using a business logic point of view. It will only have the main applications that Concur offers directly to its customers: Travel, Expense and Invoice, both mobile and web. The components under these applications will be the underlying parts such as Login, Booking attempts or Bookings. For example, if a user reports that they have problems logging in the application, the UI should display metrics related to the number of requests made, errors, latency and availability.

The rules creation wizard satisfied the requirements that were previously set, but participants agreed that it should support the formula creation as well, so that the whole process is automated and there are custom formulas for each application.

3.4. Adapted requirements

After the review with the stakeholders, the requirements of the application changed and new ones were added. The updated list includes:

1. The user should see all dashboards under a common menu called Assets.

    1.1. The user should be able to choose between the dashboard they want to access under Assets: Infrastructure, Database, Applications.

    1.2. The user should see real-time data for all metrics.

2. In the database dashboard, the user should be able to choose between Couchbase and MySQL.

    2.1. The Couchbase dashboard should include Couchbase and Memcached clusters.

    2.2. Memcached dashboard should contain clusters, hosts and buckets and metrics for each of these levels.

    2.3. The MySQL dashboard should include all belonging hosts and metrics retrieved from New Relic.

    2.4. For each element under databases, the user should see an indicator that there are violations of the metrics as warnings or critical when the metrics are not meeting the pre-defined Service Level Objectives (SLO). This information is retrieved from the API.

4. Functionality

This chapter will describe the functionalities of Pulse after the definition of the second round of requirements described in chapter 3 and further usability tests which will be described in chapter 7. It should be noted that the application is still in development, and further changes are expected to be made in the future.

Pulse is an Application Performance Management (APM) type of application which aims to give the users an overview of the health of the applications across datacenters. According to Wikipedia, such applications strive to detect and diagnose complex application performance problems to maintain an expected level of service and in their core they are "the translation of IT metrics into business meaning (i.e. value)" (Wikipedia, 2020). Pulse does this in three main

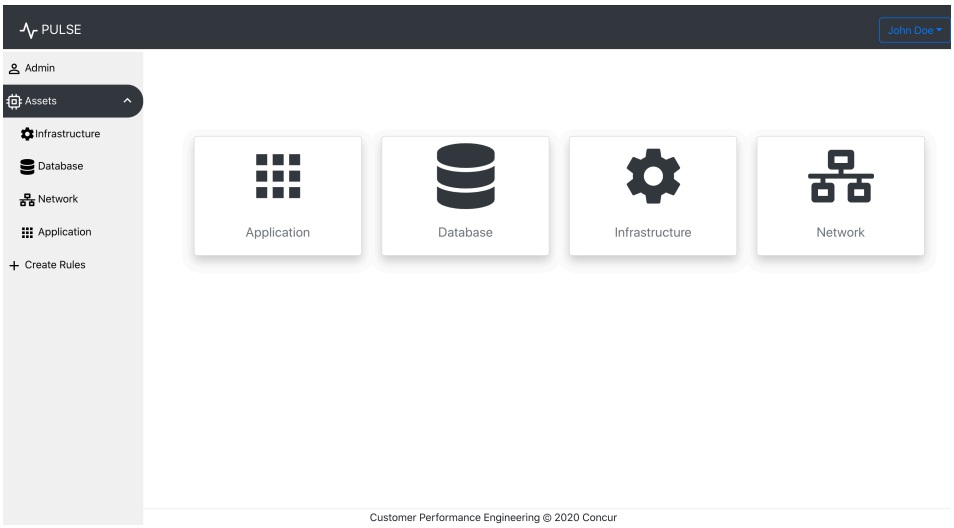

Figure 13: Pulse: assets page

ways: from an infrastructure, application, and database point of view. As defined in the requirements (Section 3.4), these are called *Assets* and at the moment are the only ones included. Later on, outside this thesis, there will be implemented other aspects such as Network. The main idea behind Pulse is of a tool that allows people with different technical knowledge levels included in an incident, to be able to identify faulty components in order to take the necessary steps for resolving it or report it to customers. It should allow for faster problem identifying by centralizing information gathered from multiple resources, depending on each application or service. The tool allows service owners and incident engineers to “teach” the tool what is important to them, which metrics they need to see, what are the allowed and critical values of

these metrics by giving them the opportunity to create rules and checks specific to each application.

The landing page of Pulse after the users logs in, will be the Assets page shown in Figure 13. Here, the users will be able to choose the type of Asset they want to observe. At the moment, the stakeholders do not find it necessary to have a visual representation of the health of each of these assets on this level, so each component is by default a grey color and no aggregation of the metrics in the underneath components is performed. Each of these assets represents a different dashboard: Infrastructure, Applications and Database. There is a side navigation bar available throughout the application for easy access from one dashboard to another or other parts of the application. At the moment, the user can also navigate to the Create Rules wizard, described in section 3.2.3.

4.1. Infrastructure dashboard

The infrastructure dashboard is organized into several layers. First it shows a view of the applications (Figure 14). Each application is represented by a *score card* as described in Section 3.2, containing the name of the application and an overall health score in a percentage score colored according to a threshold set to 0.5 (50%) for simplification purposes until specific rules are set for the applications once Pulse is in production.

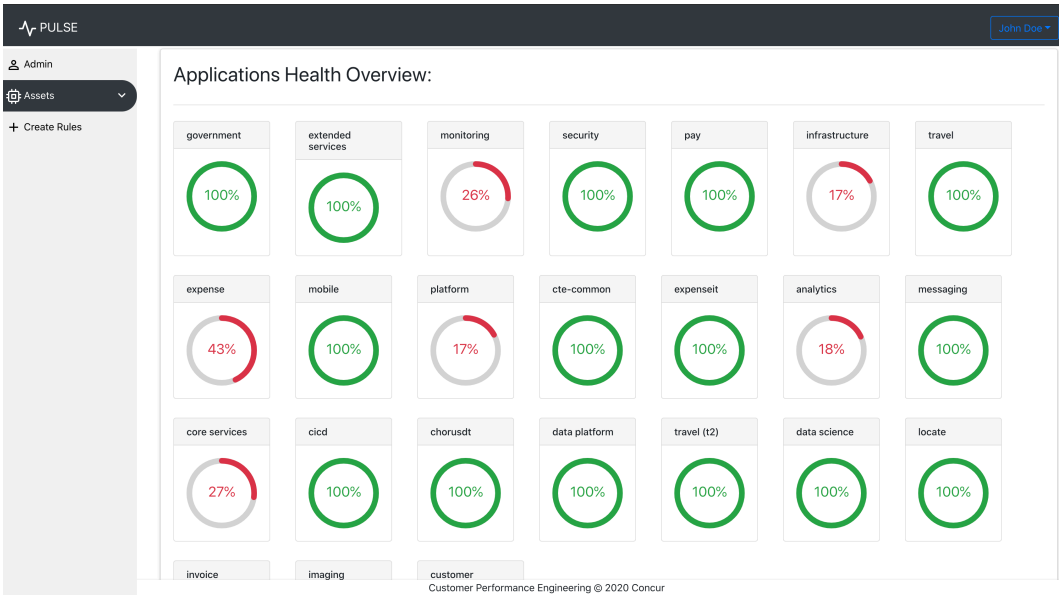

Figure 14: Infrastructure: applications

Looking at this page, it can be quickly seen that the application presenting problems are *'monitoring', 'expense', 'platform'* and so on those represented in red color to warn the user. Each application contains its own set of role types, which is practically a grouping of hosts that run the same service. For example, the *expense* application contains a number of role types, among them the *eui* role type belonging to the *expense user interface* and it identifies the hosts that run the user interface service for the *Expense* application. Clicking on any of the applications will show all the role types belonging to it, as in Figure 15 below.

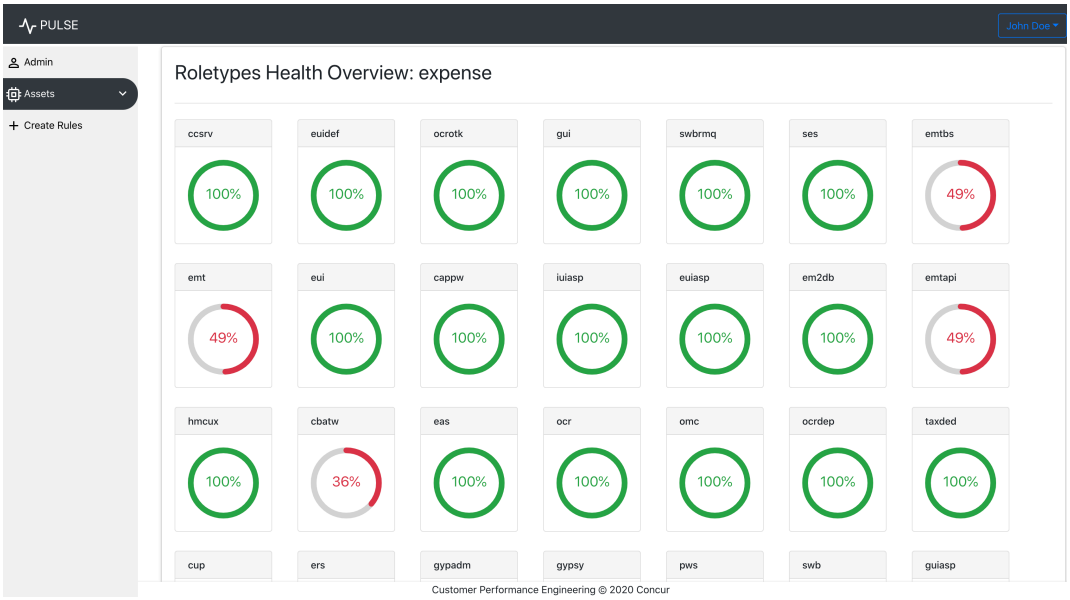

Figure 15: Infrastructure: role types

However, each role type has a number of hosts grouped into it which run the same service, and some of these hosts might belong to the same pool. Each role type may have more than one pool. For this, the user can navigate from the role type to the pool, if it has any and then to the hosts, or directly to the hosts if it does not, as shown in Figure 16.

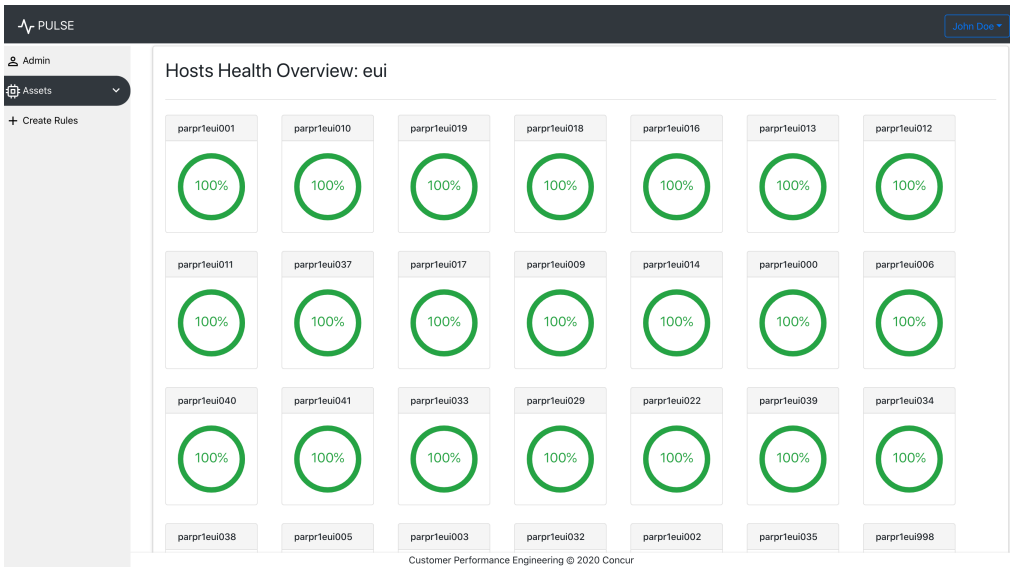

Figure 16: Infrastructure: hosts

Again, each of these levels contains the same information, name of component, an overall health score and the appropriate coloring. The level that contains the necessary metrics, is the host metrics page. Clicking on any of the hosts, will take the user to a Host Metrics page shown in Figure 18 for each host, which contains the metrics collected for each host visualized in charts, the source where they are being collected with a direct link to that source and a date picker for them to choose the time range. However, the time range is there as an added tool, because the view is continuously

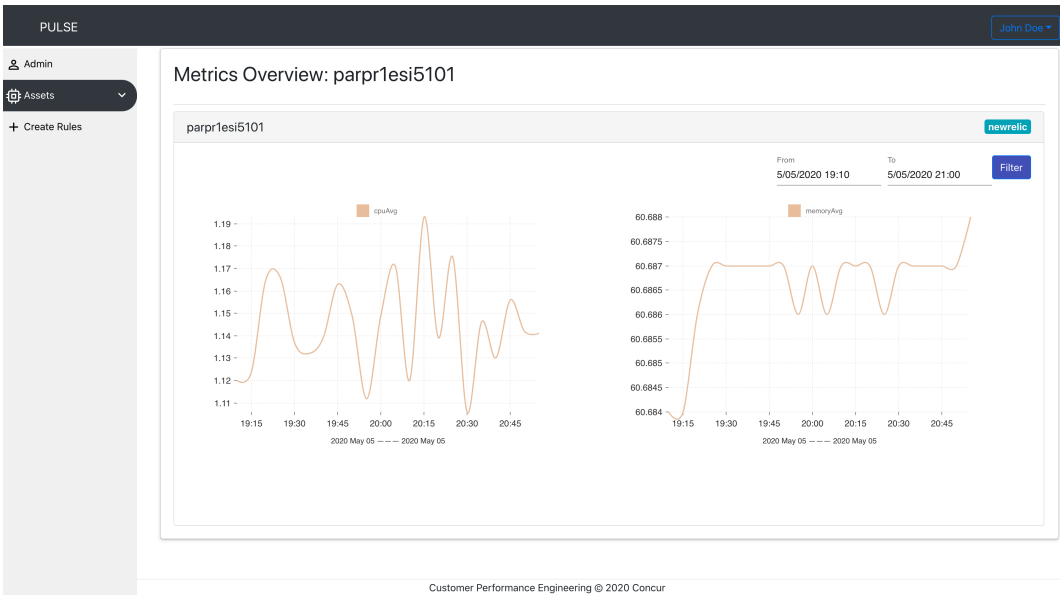

Figure 17: Infrastructure: Host metrics

communicating with the API to retrieve the latest information. When there are new metric values, they will be fed to the charts without refreshing the whole page but only adding more values in the chart, maintaining a real-time communication between the front end and the back end. This is done using a Publish-Subscribe architecture and will be explained in detail in Chapter 6.

4.2. Database dashboard

The database dashboard is dedicated to the databases currently in use by the services at Concur and is also organized in layers. First the user is able to choose which database they want to check. At the moment, there are two databases available: Couchbase and MySQL. To choose between them, the user needs to navigate from Assets to Database and will be shown the view in Figure 18. Each of the databases is visualized in a 'card' that holds the database icon and the name of the database. The icon will be colored according to the overall health of the components underneath, for example clusters in case of Couchbase.

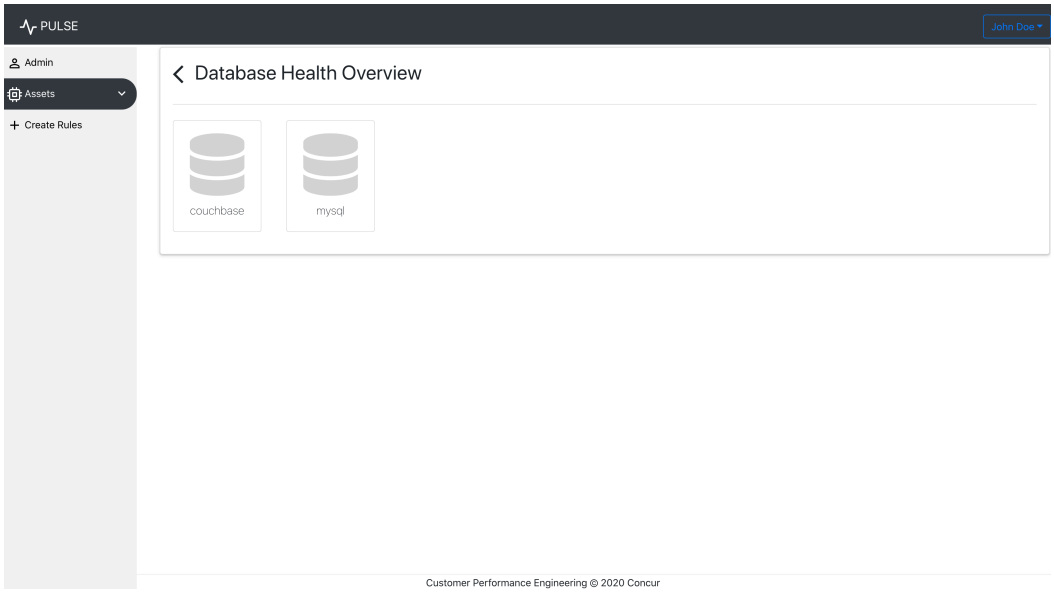

Figure 18: Database: available databases

The dashboards for Couchbase and MySQL follow a similar structure given that they are both organized in clusters and then hosts. Couchbase has one extra level which is buckets. In Couchbase, the clusters can be Couchbase or Memcached clusters, that's why they are split into two tabs correspondingly as shown in Figure 19.

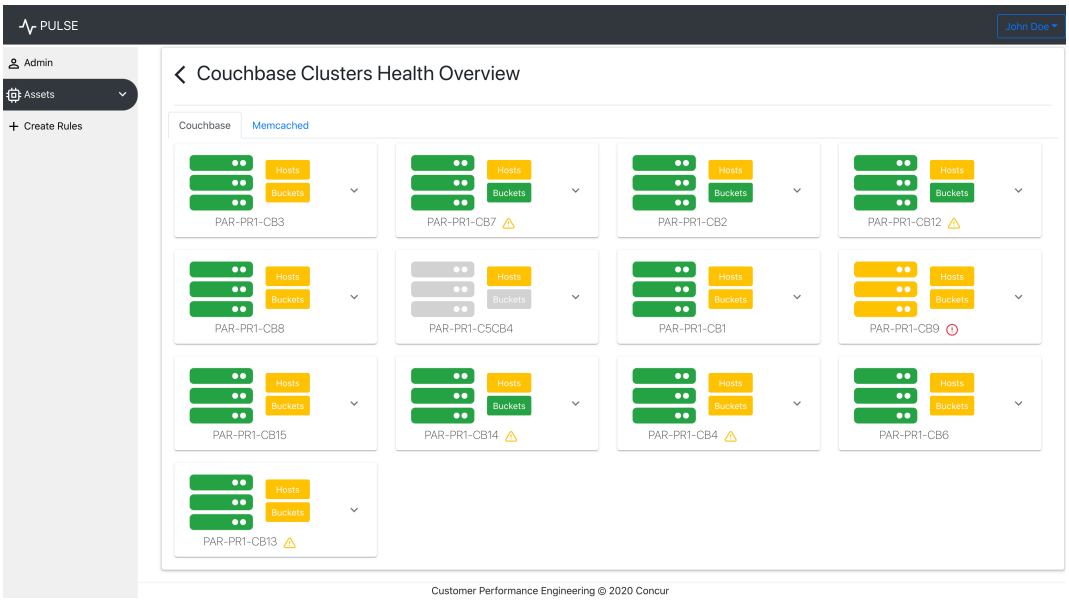

Figure 19: Database: Couchbase

Both tabs follow the same logic, and the same approach is applied also to MySQL clusters, however in that case they are split into tabs by role type. Recalling from section 4.2., a role type groups hosts running the same service. From a cluster, the user can navigate to its hosts and buckets using the corresponding buttons in cluster card. The cluster card is expandable and shows the metrics collected for each level. The colors of the cluster icon is determined based on an overall health calculation of the metrics,

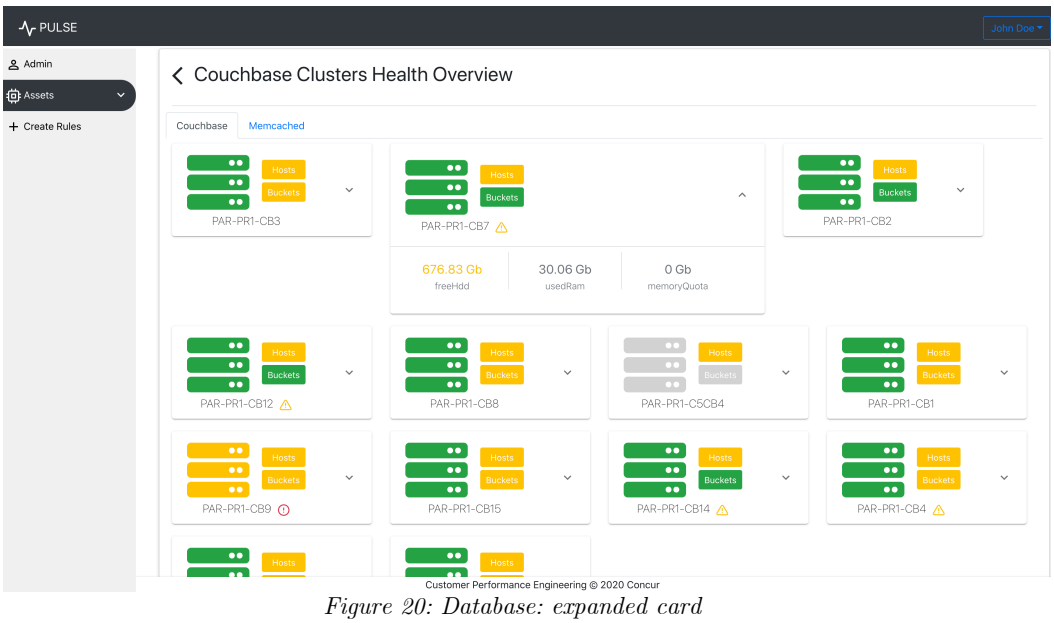

Figure 20: Database: expanded card

comparing them with the set thresholds in place. The colors of the buttons refer to the health of the corresponding components where each of them

navigates the user. If the host button is red, it means at least one host is in critical state, if there are no critical hosts but there is at least one in warning state, the button is yellow, otherwise if all hosts are healthy, the button will be green. If there are no health scores for that level, the button will be in gray color.

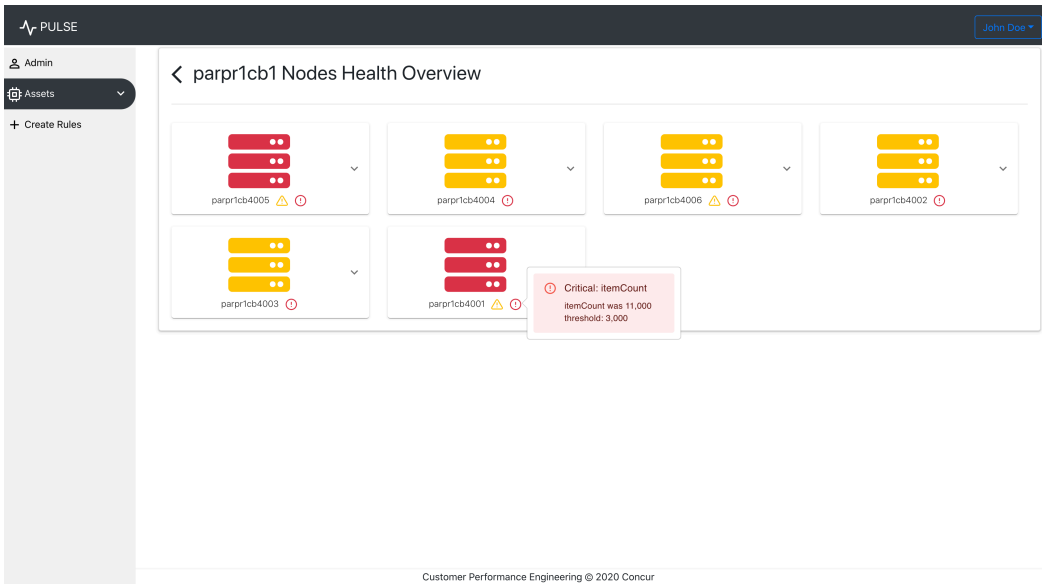

Figure 21: Dashboard: Hosts

The hosts view is available for both databases and the visualization components use the same approach with the icon coloring and the way the metrics are shown. There is also another element in the card itself for cluster, host and bucket level. Next to the name of each cluster/host/bucket there will be up to two icons indicating there are metrics in 'warning' or 'critical' state in order to make the user click on the cluster and see a detailed view even though the overall status might indicate better health. These alerts are shown in the form of small pop-ups next to the card, giving information about the metrics that has a violation of the set thresholds, the current value and the threshold. An example is in Figure 21.

4.3. Applications dashboard

The applications dashboard is a dedicated dashboard for applications, but from a more business-like perspective. Its aim is to create a service map of all the services under a certain component and have an overview of the health of the application by collecting and visualizing business Key Performance Indicators (KPI), followed by service-level measurements, and then finally, measurements of infrastructure performance. Under normal

state, the dashboard empowers Service Management teams to monitor the site and observe when services might start to miss SLOs but customer-impact is not yet at P1 or P2 level.

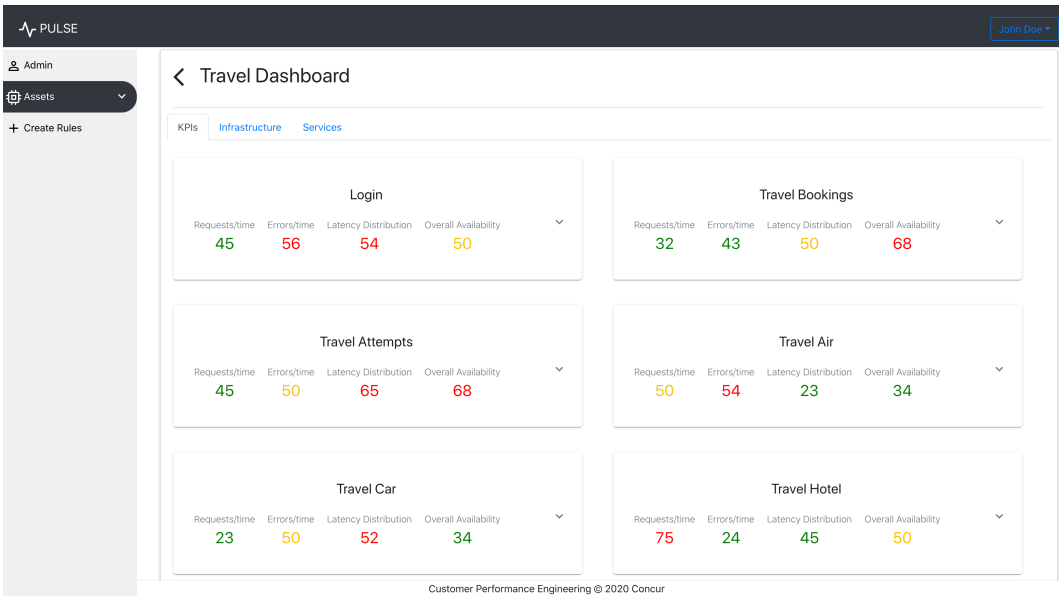

Figure 22: Application: Business KPIs

During a P1 incident, the dashboard allows SM to quickly pinpoint the lowest level service or piece of infrastructure that is failing, so that the correct runbook can be fetched, and attempts can be made to restore the service by the SM team. If the runbook does not restore service and true triage is needed, the dashboard should allow the triage team to focus their efforts instead of trying to find a needle in a haystack. The application dashboard metrics focus on the *Northstar* metrics (Barlow, 2019) of Concur which comprise of number of attempts, number of successes, average elapsed time, and 95th percentile elapsed time.

After choosing the Application dashboard from the Assets menu, the users can choose the application they want to check and the mobile or web version. This dashboard is still not fully developed and in the scope of this thesis, only the mobile version of the Travel application has been partially implemented. As explained before, it focuses of KPIs, Service Level measurements and Infrastructure performance, so the UI offers three different tabs for them as shown in Figure 22. Each of the applications is split into logical parts corresponding to main business areas of the application such as Login, Travel Bookings, Travel Attempts (unfinished bookings), Travel Air, Car, Hotel and so on depending on the services offered

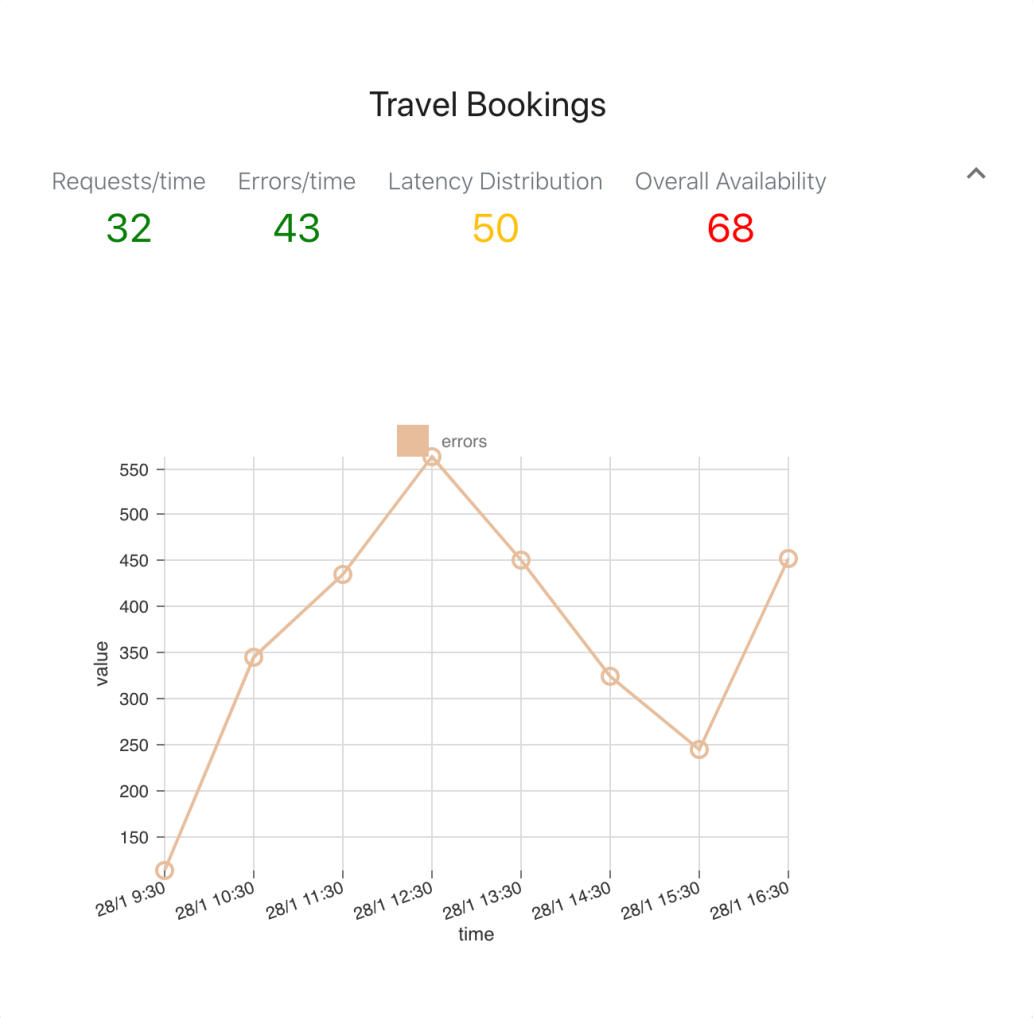

Figure 23: KPI card

by each app. Each card shows the 4 Northstar metrics and when clicked on any of them, it expands to show a real-time chart of its latest data.

The chart is updated whenever the backend publishes an event to which this component is subscribed to and the data are inserted in the chart without any reload of the page.

The next tab in the Applications dashboard is that of Infrastructure performance measurements and visualizes applicable health metrics for each infrastructure component used and owned by Travel. Each Agile team defines what the SLOs are for the components to be yellow and then to be red, or green. In this example, we have 4 main sections, Databases, F5, RabbitMQ and Kubernetes.

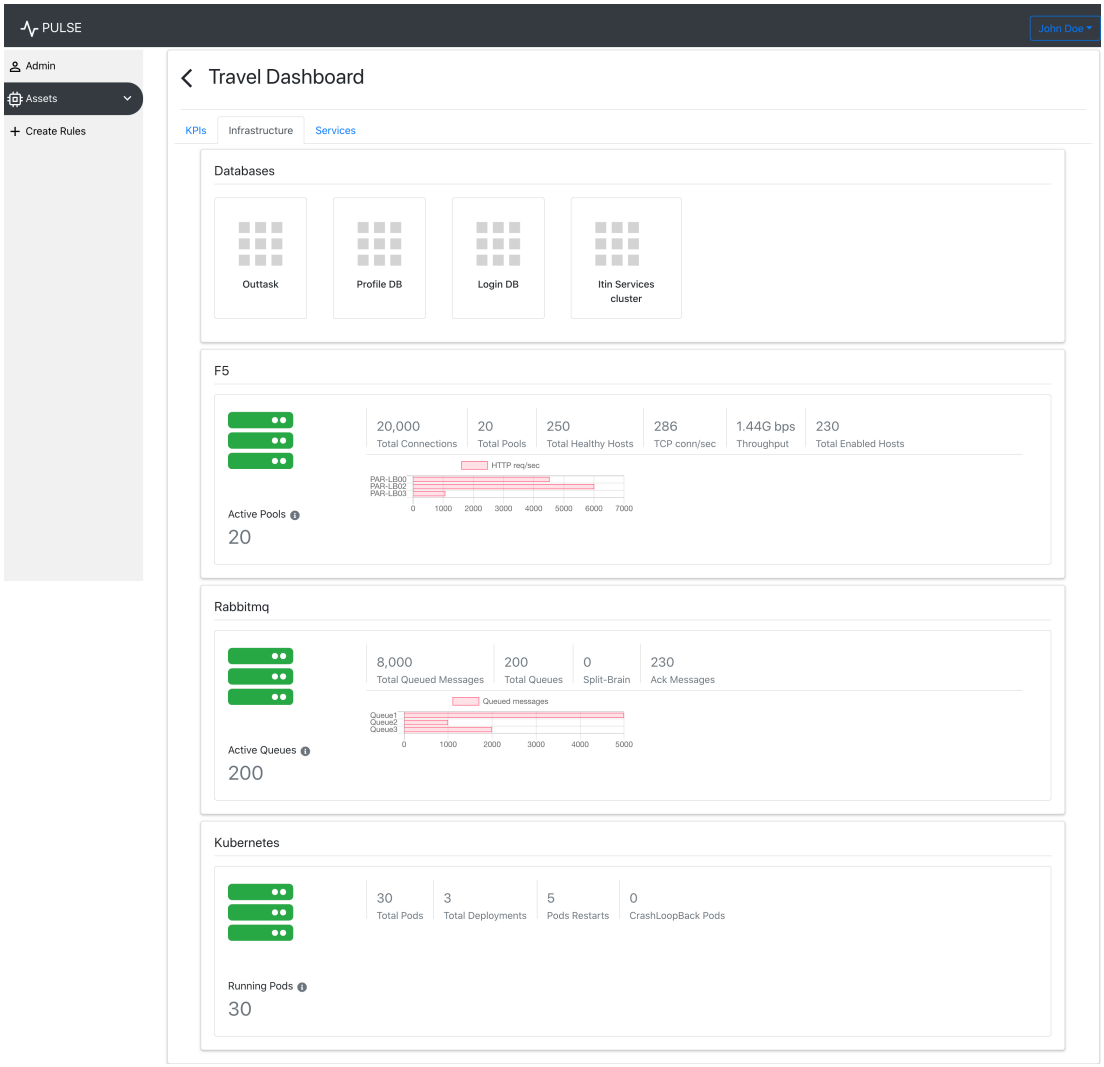

Figure 24: Application: Travel Service Levels

Travel owns an Outtask database, a Profile database, a Login database and a cluster for Itinerary Services. Travel also has its own F5 pools, which is a grouping of web services gathered to receive and process incoming traffic to

any of the nodes in the pool. There is also a chart that displays in real time the number of requests per second on each of the pools. The RabbitMQ (message broker) section, shows a breakdown of the queued messages, the total number of queues, active queues, messages sent and so on. The chart displays the number of messages in each queue in real-time. The last section belongs to Kubernetes, showing basic information regarding the number of pods running Travel services, the deployments, restarted pods and potential errors. The coloring of the icons follows the same logic as in the other dashboards. If considered healthy, it is green, otherwise it is yellow for warning level and red for critical level.

The third tab shows the Service Level measurements of each service owned by the Travel applications as a breakdown of the *Northstar* metrics. Before getting into the metrics, this tab builds a dependency graph of all the services belonging to the application. It is concepted as a *graph card* with all the necessary metrics shown in Figure 25. Ideally, the metrics shown below the dependency graph will change based on which service the user clicks on. Each of the nodes in the graph is a component itself, the color of which is based on the health of its metrics and the icon should be indicative of what type of service it is.

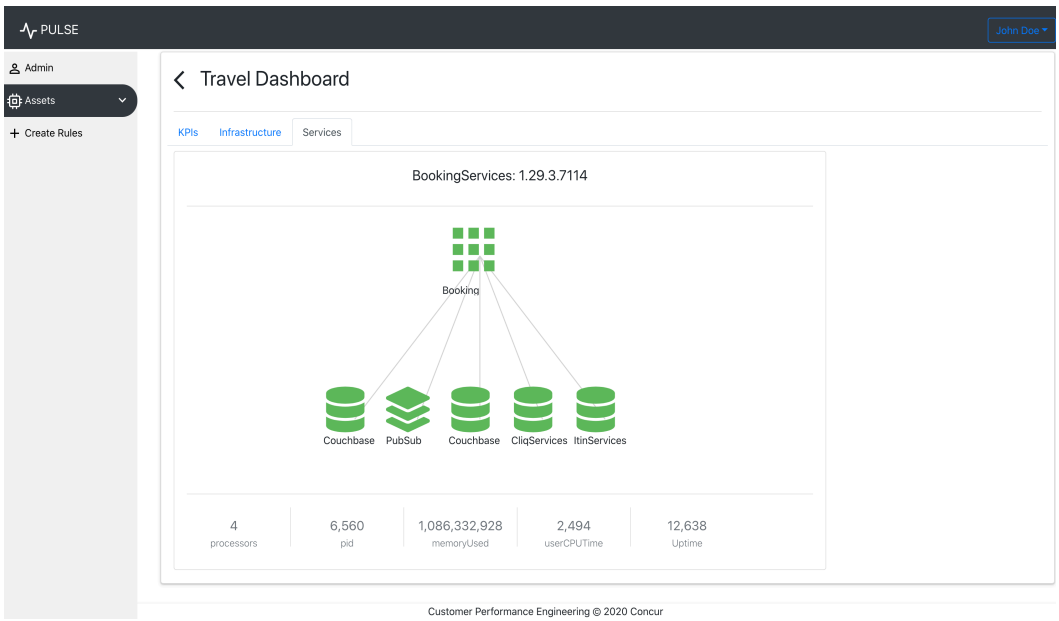

Figure 25: Applications: Services

However, the Applications dashboard is still being built and the current data shown are only using local files as the views are tested with potential users

so that they have only the necessary information for each tab and component. Travel was chosen arbitrarily to be the first application to build the dashboard for, but the same approach will be followed for other applications when it's finalized and in production.

5. Implementation Overview

This chapter focuses on describing how Pulse as a software product is built, including the frontend application, back end modules, database, how they relate and communicate to provide functionalities to the users. It also describes the rationale behind the decisions taken for the used technologies and approach in order to support user-centered design and be able to make further modifications.

5.1. System architecture

A brief introduction on the preliminary design of Pulse has been described in section 3.1. However, after some tests and discussions with the stakeholders, the requirements changed and therefore the architecture of the system changed. The system has 5 major modules: auto-discovery module, the backend API, the database module, the UI module and the Crossbar service module.

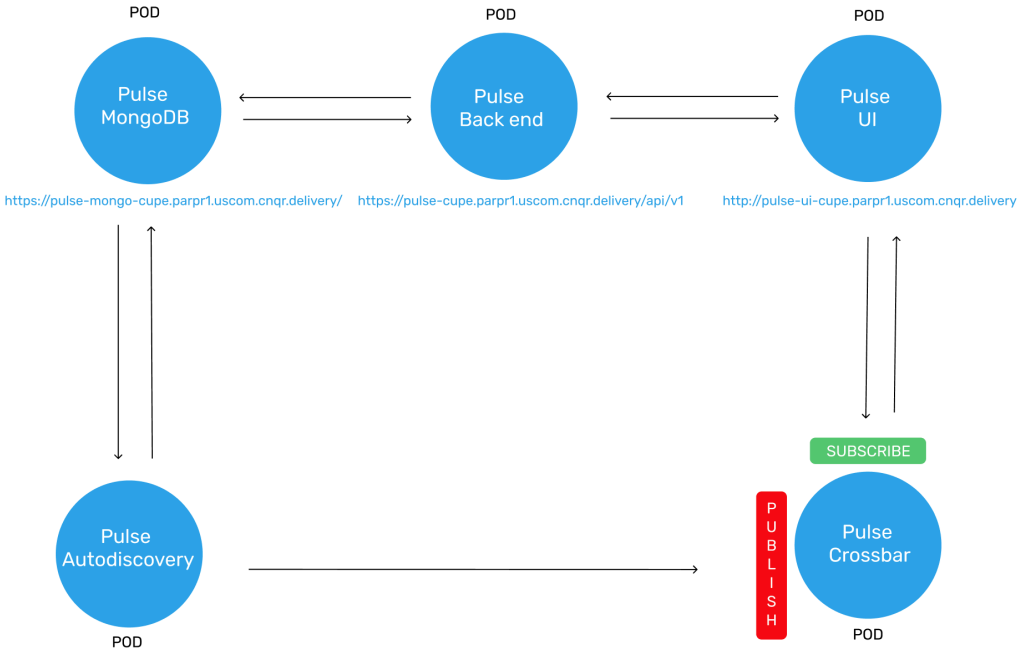

Figure 26: System architecture

The main idea behind this split of modules is to have clearly defined units that address a specific set of problems in order to keep a separation of concerns between them and have a loosely coupled architecture. This separation also encourages reuse of components. For example, the Pulse

back-end API can be used from other teams that need to use the same information. The majority of these modules belong to what would usually be called under a common name 'back-end', but they all serve a different purpose. The auto-discovery service is where all the data collection from multiple sources happens, the database module (MongoDB) is where all the collected data lives, the back-end module is where all the manipulation of the collected data happens, the Crossbar service contains the router for handling publishing and subscribing to events from other components, and the UI module is the visualization platform. Each of these components is a containerized application with Docker[14] and deployed inside a Kubernetes[15] cluster as a pod. Given that each of them is a different cluster of services, the communication for example between the UI and the back-end module is done using the ingress to expose an external route from outside the cluster to the services and pods inside the cluster and between the other components it uses the internal Kubernetes DNS to resolve the service.

5.2. Back-end infrastructure

The back-end infrastructure involves the implementation units mentioned in the previous section, excluding the Pulse UI module: auto-discovery, back-end, crossbar and database modules. In this section we will go through their main responsibilities and how they are built in order to better understand how the whole system works and how it is connected to the visualization platform described in this thesis.

Auto-discovery service module is built with Python 3.7 and it is responsible for all the data collection logic from any of the required sources including querying New Relic, Couchbase API, Compute, F5, RMS, Elastic Search. There are different types of jobs, such as those that only collect components and the relationships between them, for example all applications, role types belonging to each, pools and hosts. This information is currently not in just one source (hence one of the reasons to build Pulse), so it needs to get this information from Compute, RMS and New Relic as each of them have different parts. Then after collecting all the components and their relationships, there are jobs in place to collect the metrics for each and store

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

[15] https://kubernetes.io/

them in the MongoDB database using the *pymongo*[16] distribution to connect to the right database client, query and store data. Whenever there are new metrics, and event is published to the Crossbar service for that 'topic' from auto-discovery.

Database module is essentially a user-friendly UI to manipulate the MongoDB databases and collections in Pulse. Given that the metrics being collected are not always adhering to the same structure and format but are dynamic, it was more suitable to use a NoSQL database. These types of databases are precisely built to fit needs of modern applications which often differ from needs of a relational model. MongoDB is a document-oriented NoSQL database, with documents having a format similar to JSON and dynamic schemas. Each MongoDB has a set of databases, and each database has a set of collections which contain similar documents. As the application requirements changed, so did the way the data was stored and structured in the database for it to be better handled and scaled in the future. There are several databases containing all the data for Pulse such as a database for data coming from auto-discovery and the collections belonging to assets, components of each asset, RMS and F5 data.

Crossbar service module is responsible for the routing of published and subscribed events and makes it possible for the Pulse UI to display data in

Source: https://crossbar.io/

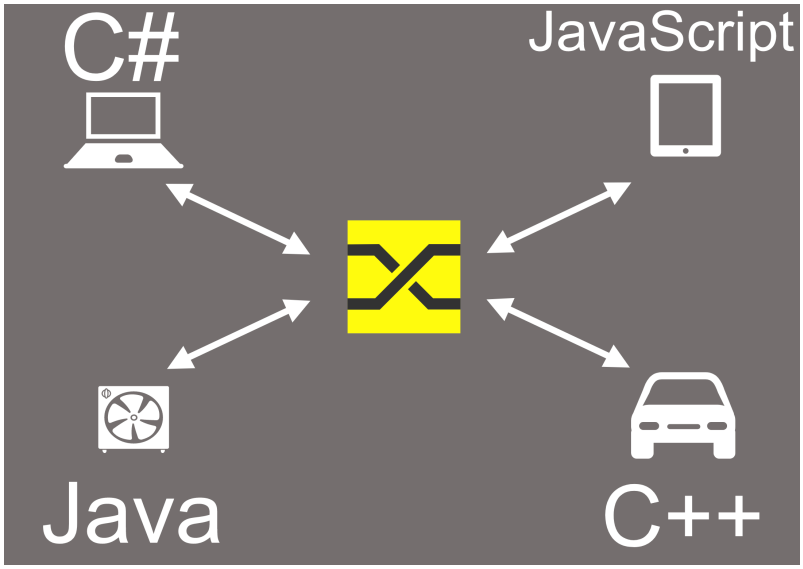

Figure 27: Crossbar router

Source: https://crossbar.io/

[16] https://pymongo.readthedocs.io/

real time. Crossbar.io[17] is an open source networking platform for distributed and microservice applications which implements the open Web Application Messaging Protocol[18] (WAMP). The WAMP protocol was proposed by Crossbar developers in 2012 and supported by them ever since, however the protocol is freely available under an open license. In this protocol, the components connect to a WAMP router which performs routing of messages between components and it supports the Publish-Subscribe and routed Remote Procedure Calls patterns. Although Crossbar.io is implemented in Python, it does not matter in which language the client is written as any client can connect to the router.
When a Crossbar.io node starts, a Node Controller Process is called which reads a configuration file and starts workers according to the instructions in it. The Crossbar module contains this configuration file for Pulse components to publish and subscribe to events using this router. To start a router and work with Crossbar.io, first there needs to be a realm and a transport mode defined. A realm is basically a namespace which Crossbar.io uses as a domain to differ between routing and administration. There are two types of transports: WebSocket and RawSocket transport and both support TCP, TLS, Tor and Unix Domain Socket endpoints.

A basic configuration for Crossbar.io contains a version, a controller and most importantly workers. In Pulse, the workers look as in the snippet below, including a router type of worker, the *pulse* realm and the permissions of different roles inside the realm.

```
{
  "version":2,
  "controller":{},
  "workers":[
     {
        "type":"router",
        "realms":[
           {
              "name":"pulse",
              "roles":[
                 {
                    "name":"anonymous",
                    "permissions":[
                       {
                          "uri":"",
                          "match":"prefix",
                          "allow":{
                             "call":true,
```

[17] https://crossbar.io/

[18] https://wamp-proto.org/

```
                           "register":true,
                           "publish":true,
                           "subscribe":true
                        },
                        "disclose":{
                           "caller":false,
                           "publisher":false
                        },
                        "cache":true
                     }
                  ]
               }
            ]
         }
      ],
      ...
   ]
}
```

However, as mentioned before, there also needs to be a transport type specified. The configuration for that is shown below. The transport is done using Web Sockets via TCP endpoints on port 8880 and on web via TCP endpoints on port 8888. The web transport type allows web socket, web services to use the same port.

```
{ …
    "transports":[
                {
                   "type":"websocket",
                   "endpoint":{
                      "type":"tcp",
                      "port":8880
                   },
                   "options":{
                      "enable_webstatus":true,
                      "allowed_origins":[
                         "ws://*",
                         "wss://*",
                         "http://*",
                         "https://*",
                         "file://*",
                         "file://"
                      ],
                      "auto_ping_interval":10000,
                      "auto_ping_timeout":5000,
                      "auto_ping_size":4
                   }
                },
                {
                   "type":"web",
                   "endpoint":{
                      "type":"tcp",
                      "port":8888
                   },
                   "paths":{
                      "/":{
                         "type":"publisher",
                         "realm":"pulse",
                         "role":"anonymous"
```

```
                    }
                }
            }
        ]
    }
]
}
```

The auto-discovery service publishes events to this service in different topics according to the metrics and assets and the UI subscribes components to these topics and updates them whenever there is new information. This is described in detail in Chapter 6.

Pulse backend module is implemented in Python and is a RESTful API getting data from MongoDB and making them available through several endpoints. These endpoints can be used by any of the teams inside the corporate network for their data visualization purposes outside Pulse. These endpoints include retrieving asset types, components and metrics, as well as helping to build a more dynamic UI by generalizing endpoints to serve multiple assets and types of components.

5.3. Client application

The client is a web application built with JavaScript ES6 and the React v.16.13.x library. Other smaller libraries are used to render nicely styled and functional components such as React Bootstrap[19], Material UI[20] or graphing libraries such as *nivo*[21]. The client communicates with the backend through a library called *axios*[22] which is fairly easy to set up and use.

React gives you the opportunity to create different types of components and the way the Pulse UI was built, allows to easily reuse and enhance components. There are smaller components such as the *score cards* which look like the example in the next code snippet:

[19] https://react-bootstrap.github.io/

[20] https://material-ui.com/

[21] https://nivo.rocks/

[22] https://github.com/axios/axios

```
const ScoreCard = props => {
  const { element, getSelected } = props;

  return (
    <>
      <Card
        onClick={() => getSelected(element)}
        className="score-card"
      >
        <Card.Header as="h5" className="card-title">
          {element.name}
        </Card.Header>
        <Card.Body>
          <CircularProgressbar
            className="circular-progressbar"
            value={getPercentage(element.score)}
            text={`${getPercentage(element.score)}%`}
            styles={buildStyles({
              pathColor: getStyle(element.score),
              textColor: getStyle(element.score)
            })}
          />
        </Card.Body>
      </Card>
    </>
  );
};
```

Similar components are defined for cluster score cards, which are expandable and contain more information, node/buckets score cards and smaller helper components such as a custom loading spinner for transitional loading states etc. These smaller components are wrapped in bigger components, which we will call here views. Such is for example the *HealthOverview* component shown below, which is made of smaller components.

```
const HealthOverview = props => {
  const { elements, getSelected, level, isLoading, selected } = props;

  return (
    <Card className="container-card">
      <Card.Body>
        <section>
          <h2 className="page-title">
            {level} Health Overview: {selected}
          </h2>
        </section>
        <Divider style={{ marginTop: "30px", marginBottom: "10px" }} />
        {isLoading ? (
          <LoadingSpinner />
        ) : (
          <>
            {elements.length ? (
              <Row>
                {elements.map(function(element, index) {
                  return (
                    <ScoreCard
                      key={index}
                      element={element}
                      getSelected={getSelected}
                      level={level}
                    ></ScoreCard>
```

```
                  );
                })}
              </Row>
            ) : (
              <div style={{ marginLeft: "45%" }}>No data found...</div>
            )}
          </>
        )}
      </Card.Body>
    </Card>
  );
};
```

However, this component does not have a state itself to manage, it's a child component which gets props from a class component, that acts as a controller for the component being in charge of updating the state and the view. These are React Class components, where you can define the initial state, how the view behaves when the component mounts, behavior of any other components in the view such as buttons, cards, date pickers etc. In React it is recommended to 'lift the state up' which means moving the state to the closest ancestor of the components that need to share the same state. In this case we have one parent (class component) and a child that wraps all other children (*HealthOverview*), so we move the state management logic to the class components. In the case of Couchbase or MySQL clusters, the tabs are sharing the same parent, and the state is managed inside that parent instead of on each of the tabs' components. An example of a class component is shown below.

```
class Applications extends Component {
  state = {
    // define any state variables and initial values
  };

  //handler for data coming from API
  fetchApplications = () => {
    // data fetching and setting state logic
  };

  componentDidMount = () => {
    // logic on component mounting
  };

  render() {
    return (
        <HealthOverview
          //pass any state variables as params
        />
    );
  }
}

export default withRouter(Applications);
```

The components and other views follow a similar logic. Each of these class components that manage the state of their children components are subscribed to Crossbar service and whenever there are new metrics on a topic for example *Roletypes*, the *RoletypeContainer* is updated and the state change is reflected in the view. In Chapter 6 we go more in detail about how this is achieved and handled from auto-discovery, crossbar and UI modules.

The UI is built in such way so that it is as dynamic as possible by connecting each asset to a major visualization component. For example, the Couchbase dashboard which is split into tabs and is using the expandable cluster score card, along with other data about the asset, there is also stored another field which defines what type of component the UI should render and how the structure of the page looks like. This information has a structure as shown below. The UI then makes use of React API methods such as `React.createElement()`[23] to create components dynamically.

```
ui_components: {
    tabs: [
        'couchbase',
        'memcached'
    ],
    card: 'ClusterCardBtns',
    buttons: [
        {
            name: 'hosts',
            path: 'hosts'
        },
        {
            name: 'buckets',
             path: 'buckets'
        }
    ]
}
```

[23] https://reactjs.org/docs/react-api.html#createelement

6. Visualization of Stream Data

This chapter focuses on describing how the whole system works together to build an application that visualizes stream data in real-time and how it makes use of the React lifecycle and Crossbar event messages to update components accordingly and have a smooth user experience while they troubleshoot problems in faulty services.

6.1. Analysis of received data

To understand more of how the data is processed on each stage from collection and storage to retrieval with the API and visualization in the web application, we will first go through the data we are working with. We described in Chapter 5 that the *auto-discovery* service module is responsible for collecting the data from New Relic, F5, Couchbase API, Compute and Elastic Search. As the data is collected from different sources and they often have different structure, it's necessary to structure them uniformly, in order to be able to create generic data manipulation methods and scale the application easily. And this is done at the time of collection, before storing them to MongoDB. The Pulse backend module that serves as a REST API is built to be generic and dynamic enough to allow for calls to be reused for different purposes of data manipulation and different components. This logic supports the React web application of Pulse to be loosely coupled and allow for visualization components and code to be reused by having centralized components that handle all cases for a certain asset.

Taking as an example the Database asset dashboard, there is currently a dashboard for Couchbase and MySQL. Later on, other dashboards for other databases will be added as the application grows. From this point of view, having a similar structure of data and a common component to handle the data retrieval and state updates, makes the application handle the new dashboards just by retrieving the metrics from the API and will automatically render the new view, without the developer needing to tweak the application again. As discussed in the previous chapter, the UI is rendered dynamically for each asset by creating a React element based on the name of the UI component that each asset is associated with. Continuing with our example, the structure of received metrics for Couchbase clusters, is shown in the snippet below which is a result to a call of the API endpoint:

`/api/v1/assets/database/couchbase/clusters/couchbase/metrics` which can be used also as `/api/v1/assets/database/couchbase/clusters/couchbase/metrics` for Memcached clusters or like `/api/v1/assets/database/mysql/clusters/pcl57c/metrics` for MySQL hosts which belong in the *pcl57c* role type.

Result example:

```
{
   "data":{
      "parpr1cb3":{
         "rest-couchbase":[
            {
               "cluster":"parpr1cb3",
               "metrics":{
                  //metric values and units
                  }
               },
               "beginTime":"2020-05-16T16:45:02Z",
               "endTime":"2020-05-16T16:45:02Z",
               "origin":"rest-couchbase"
            }
         ],
         "autodiscovery":[
            {
               "cluster":"parpr1cb3",
               "metrics":{
                  // overall status of the cluster
                  }
               },
               "beginTime":"2020-05-16T16:45:01Z",
               "endTime":"2020-05-16T16:45:01Z",
               "origin":"autodiscovery",
               "level":"cluster"
            }
         ]
      },

    // other clusters
   }
}
```

This metrics object is then parsed in the UI using a helper function which transforms it into the format below so that it easily mapped in the component built to visualize the metrics and health indicators for each cluster:

```
[
  {
    "metrics": [
      {
        "name": "freeHdd",
        "value": 1004.89,
        "unit": "Gb",
        "status": "good",
        "failedThreshold": null,
        "impact": "warning"
```

```
    },
    …
  ],
  "name": "parpr1cb3",
  "hosts": "#28a745",
  "buckets": "#ffc107",
  "overallStatus": "good"
},]
```

6.2. Overview of React concepts

When first starting to build this application and choosing the "best tools for the job", there were many web technologies and frameworks considered before deciding to use JavaScript ES6 and React, such as plain JavaScript, Vue or Angular. Even though React is in fact a library, while Angular and Vue are web frameworks, there are similarities in their core concepts about how to build UI components and applications. However, the choice was based on a few realistic factors as the UI was to be built from scratch and the engineering team's primary focus in the past had been mostly on back end technologies.

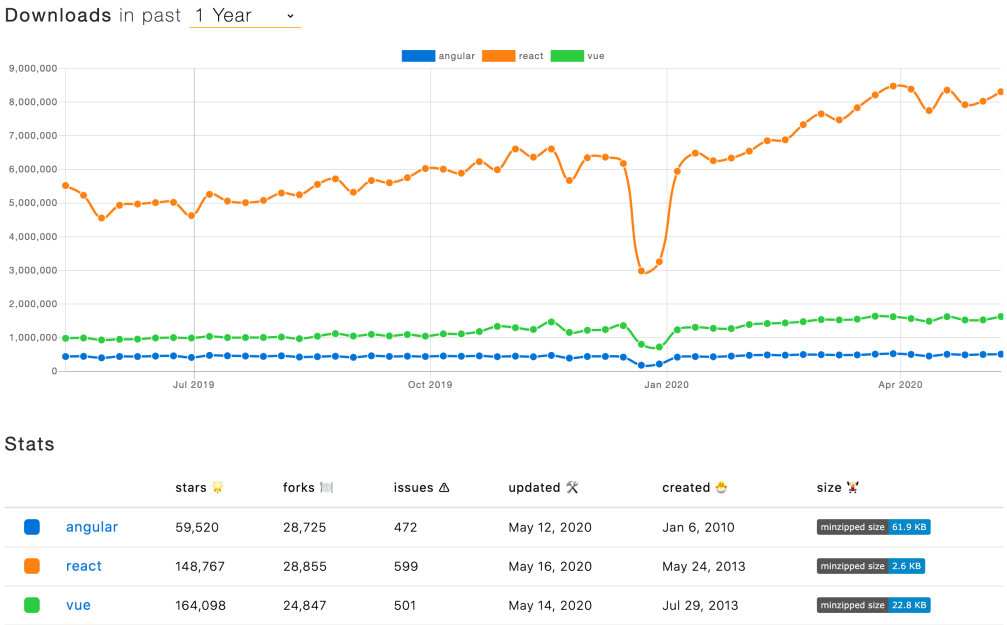

Figure 28: React vs Angular vs Vue
Source: https://www.npmtrends.com/angular-vs-react-vs-vue
Data comes from npm's download counts api and package details come from npms.io

After consulting with other more experienced teams inside the organization, it was decided to use React (currently on version 16.x), because it offered more support inside Concur so we could get help from other teams as well as outside it, having a large user community and being maintained and

developed by Facebook in 2013 which means there is plenty of online help to get started with it and also indicates that the library will be around for a long time. From Figure 28, we notice there is a huge gap in the number of downloads of React compared to Angular and Vue in the past year.

Another reason to choose React, is the easy logic of creating components. React uses JSX, which is an XML-like language used for templating, and instead of adding specific directives to your HTML like Angular does through *ng-if* or *ng-for,* you use the same logic as you would with JavaScript. Using the simplest example from the React documentation page:

```
const name = 'John Doe';
const element = <h1>Hello, {name}</h1>;
```

Inside the curly braces you can use any JavaScript expression you want, and you can also use JSX expressions in loops or if conditions, assign them to a variable, pass them as arguments or return them from a function. After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.[24]

One of the main reasons why React was chosen is that it's lightweight (only 2.6Kb) and blazing fast, which is attributed to its Virtual DOM. In case, for example, we have constructed a component with many variables in our JSX template and the state changes, React will not re-render the whole component but will find the parts that changed and update them, a bit similar to how Git defines differences in files. Angular on the other hand uses regular DOM and as the application grows and more requests are made to this component, the performance of the application is affected drastically.

React is indeed not a framework, but rather a library, and as such does not enforce any rules on how to structure your application or what other libraries to pair it with. Although React itself offers a lot of functionalities for which you do not need to install any third-party source, there are a lot of other libraries out there and you can use any one of them as long as they are compatible with JS and React and fulfill your purpose. This may bring problems because you need to be in charge of third-party libraries deprecations and upgrades yourself, however there are many libraries which are commonly used together with React and there is a whole community to

[24] https://reactjs.org/docs/introducing-jsx.html

seek help from. React Redux[25] is an example of a well-known and widely used library for global state management falling into libraries commonly used with React. However, to keep the application lightweight and use more of what React itself offers, we decided to handle this using the React lifecycle

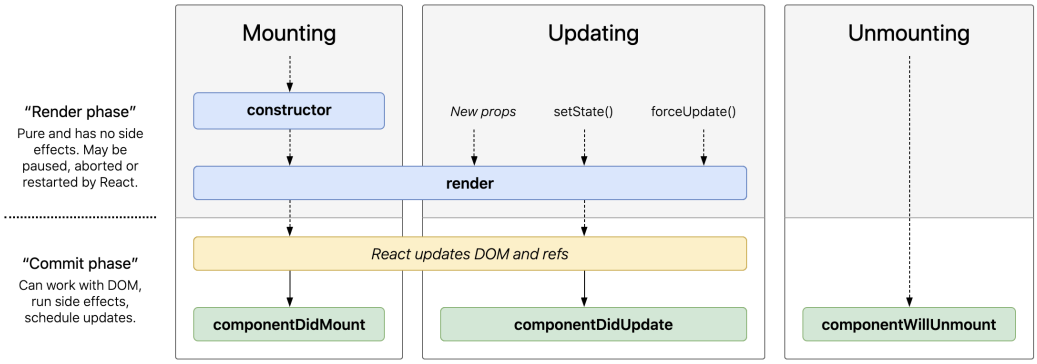

Figure 29: React lifecycle

Source: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/

methods and component state management callback functions to manage state inside each component and global one. React lifecycle is an important aspect of this library, as it offers full control of each stage a simple component goes through and allows developers to manage each of these stages. Indeed, the idea is quite simple. A component is born (mounted), the component lives (gets updated, changes), and then the component dies (unmounts). The logic implemented on any of these methods, depends entirely on the developer, however in the documentation there are useful hints and scenarios to help with understanding the purpose of each stage.

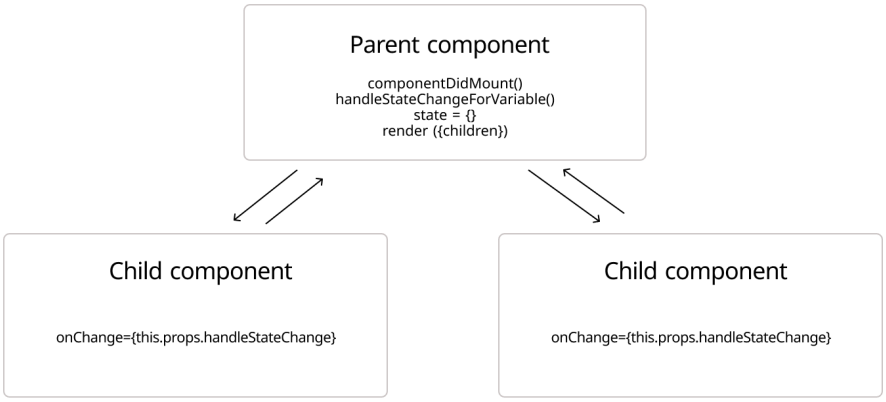

Figure 30: Lifting state up

[25] https://react-redux.js.org/

Another useful concept in React ifs 'lifting the state up'[26] which is a recommendation to handle the state of two children components sharing the same ancestor in the ancestor rather than in the children component. React has only one-way binding, compared to Angular which has two-way data binding, meaning that a change in the view will not update the state in the model, but a change in the model will update the view. However, if two components need to share the same state but they are each handling it on their own handlers, React does not ensure the values will be in sync and that there will be no discrepancies in the values. On the other hand, sharing the same state, means the children always have the same state data whenever there is a change, and it passes to children through props. Speaking of props, it is often confusing how they work compared to state, but as a rule of thumb: state changes, props do not. However, handling state changes in the parent, means changing the props passed to the children, although the children cannot explicitly change the props themselves.

6.3. Implementing real time communication

To have real-time data at all times in the front-end application, we need to first establish a connection between the auto-discovery service, Crossbar service and the front-end application. The connection is as follows: auto-discovery collects data, structures them and stores them in the database, before this operation finishes it also publishes an event to the Crossbar service which looks similar to the snippet below:

```
self._crossbar.publish(
    topic=CONFIG["CROSSBAR_KEYS"]["NEWRELIC"]["TOPIC"],
    msg={
        "roletype": roletype,
        "level": CONFIG["CROSSBAR_KEYS"]["NEWRELIC"]["LEVEL"],
        "type": CONFIG["CROSSBAR_KEYS"]["NEWRELIC"]["TYPE"]})
```

Publishing is done through a HTTP Bridge which is a service that allows clients to submit PubSub events via HTTP/POST requests. Crossbar will receive the event data via the request and forward the event via standard WAMP to any connected subscribers in real-time. A class for Crossbar is created in auto-discovery and handles publishing of events to `pulse-autodiscovery-cupe.parpr1.uscom.cnqr.delivery/wsp`. The method in

[26] https://reactjs.org/docs/lifting-state-up.html

the snippet above, calls this publish method and passes as arguments the topic for which it wants to publish an event and the message which can define the role type, level of the component being updated and the type of message which can be an update or a metric. In Chapter 5.2. it is described in more detail how the Crossbar service is built to have a router for handling clients publishing and subscribing to events. However, once these events are published, they need to be "caught" by the front-end application to update the UI. To do this, a package called *autobahn-react* is used which is essentially a client that implements the WAMP router in order to connect to the router, to publish events or to subscribe to it from the React application. This package offers a quite intuitive API to establish a connection, to subscribe and to publish events as well as handlers for when the connection is ready, lost or unreachable to allow the developer the implement their own logic on each case according to the needs of the respective application.

The root of the React application for Pulse is in the file *App.js*, where there are defined all the routes for navigation and the corresponding components for each route. As we need the connection to the Crossbar service to be available to most components in the application, the connection is initialized before the application itself builds and renders any of the other components. To do this, we first import the autobahn-react package and include the snippet below.

```
Autobahn.initialize(
  "wss://pulse-autodiscovery-cupe.parpr1.uscom.cnqr.delivery/ws",
  "pulse");
```

This is enough to start establishing the connection between the front-end application and the Crossbar service. Then, on each component that needs to use this connection, using one of the React lifecycle methods such as `onComponentDidMount()`. When this method triggers, the component makes sure the connection is ready, subscribes to the topic it needs and then executes the logic necessary. For example:

```
  Autobahn.Connection.onReady(details => {
    console.debug("Hey, connection established!", details);
    Autobahn.subscribe("ROLETYPE", details => {
```

```
        if (details[0].roletype === this.state.roletype) {
          console.debug(
            "updating role type",
            details[0].roletype,
            this.state.roletype
          );
          this.feedMetricsToChart();
        }
      });
    });
    Autobahn.Connection.onLost(details => {
      console.debug("Hey, connection lost!", details);
    });
```

The `Autobahn.Connection.onReady()` method provides a callback function with an object that describes the details of the established session with this connection and other data about the realm, authentication and broker/dealer roles. In the session data we can see the current subscribers to the router and the topics they are subscribed to:

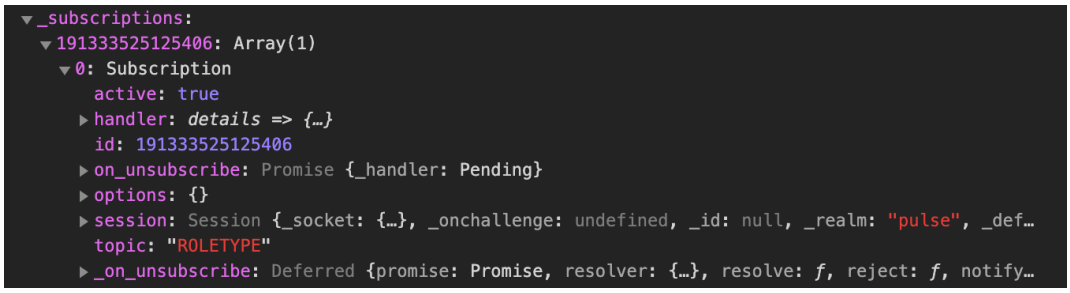

Figure 31: Subscribers information

`Autobahn.subscribe()` is also a callback function that provides us some information regarding the level of the update on the topic, the role type and

Figure 32: Subscribed event data

the type of the update that happened. In this example, we have an update on the host level metrics of the pcl57c roletype.

With this data, we have enough information to know which call to make to the API to give us the updated data. The response contains only the latest data just collected by the auto-discovery service, and we push this information to the object we have already mapped in the UI, and if the data

is visualized into a chart, it is continuously fed in the chart dataset to create a "flowing" real-time metrics chart. Otherwise, if it is a single metric, we replace the object accordingly and display the new metrics on each visualization card.

In case the connection is lost, we get the following information, possibly with a reason and a message, and the client will retry to establish the connection after a short delay.

Figure 33: Connection lost information

7. User testing

This thesis is part of larger efforts at SAP Concur to build a centralized APM tool that enables the engineering teams and higher management to have an overview of the impact of faulty services to the customers. The purpose is to have a centralized source of truth that gives a high-level view of the applications and their components across data centers. As users are spread out among many environments both virtually and geographically, it is important to continuously monitor the products offered to them in order to deliver software solutions that fulfill the customer needs at all times.

In order to test whether this tool fulfilled the needs of the target users, a series of usability tests were conducted. The main purpose is not to have a beautifully looking UI, but rather a functional one. Participants were from teams involved in the concepting and building process from the beginning as well as other team members from service and incident management teams that will eventually use it daily. As Pulse is an internal tool, it was easier to form testing groups within the company. Although in the last months of writing this thesis and building Pulse, there was a global situation going on that forced all teams to work virtually, the usability tests were still conducted via online communication platforms.

Chapter 3 describes the first usability test we conducted with a group of target users when Pulse was still in the concept phase to see if the first designs were aligned with the initial requirements. After that phase, the requirements changed and adapted to the remarks made from participants. In the next phases, a similar approach was followed: forming a target group with participants from all teams that will use the tool, building a set of tasks for the participants to complete and follow along with them as they go through the tasks. All participants were asked to share their thoughts freely and possibly "think loudly" in order to be able to catch possible errors or areas to be improved.

A major change in requirements was that of building a real-time dashboard, so whenever there are new metrics, they are displayed and updated in the interface. As the impact of this is better noticed in the Host Metrics view, we asked users to pay careful attention to any details in that view and also in every other metrics page where this is implemented. There were two major

study phases involving all three dashboards. Development of the Create rules functionality has been put on hold until there is enough input from all teams on a workflow of building rules and setting constraints and thresholds for each service. Until then, it will only be available for applications and hosts under the Infrastructure dashboard.

7.1. Usability tests

The testing sessions were conducted right after the UI was organized into Assets which involved the Infrastructure and Database dashboard. The group included members of all teams that will be using the tool later on, such as Service Management, Critical Incident Management, Logging, Quality Assurance, Container Ecosystem, Database Administration, Customer Support as well as representatives from higher management in the Cloud Services department. The tasks in this phase were quite similar to the ones in the preparatory session described in Chapter 3. The user needed to:

Set 1.

1. Navigate to Infrastructure dashboard applications view
2. Identify low performance applications
3. Navigate to role types of the *Travel* application
4. Identify a low performance role type of *Travel*
5. Navigate to hosts of that low performing role type
6. Identify a low performance host (if any)
7. Navigate to metrics of the host, check different time ranges.

Set 2.

1. Navigate to Database dashboard
2. Navigate to Couchbase dashboard
3. Check the metrics of the PAR-PR1-CB1 cluster
4. Identify the health status of buckets and hosts of PAR-PR1-CB1 cluster
5. Identify cluster warning/critical alerts, if any at the moment
6. Navigate to hosts of PAR-PR1-CB1
7. Check host metrics
8. Navigate to buckets of PAR-PR1-CB1
9. Check bucket metrics

Set 3.

1. Navigate to Database dashboard
2. Navigate to MySQL dashboard
3. Identify low performance clusters
4. Identify the health status of hosts
5. Navigate to hosts of any cluster
6. Check host metrics

Set 4.

1. Navigate to Applications dashboard
2. Navigate to Travel dashboard (mobile or web)
3. Check KPI metrics for Travel business components
4. Check the Overall availability metric details
5. Check Infrastructure details, identify the overall health status of each infrastructure component
6. Check Travel services dependencies and metrics

After completing these tasks, the participants were asked to fill a short questionnaire with 10 statements of the System Usability Scale (usability.gov, 2020) standard.

1. I think that I would like to use this system frequently.
2. I found the system unnecessarily complex.
3. I thought the system was easy to use.
4. I think that I would need the support of a technical person to be able to use this system.
5. I found the various functions in this system were well integrated.
6. I thought there was too much inconsistency in this system.
7. I would imagine that most people would learn to use this system very quickly.
8. I found the system very cumbersome to use.
9. I felt very confident using the system.
10. I needed to learn a lot of things before I could get going with this system.

For each of these statements, they needed to indicate to what level they agree with it from *Strongly disagree* to *Strongly agree* (1 to 5). The process to get an overall score from the responses is as follows:

- For odd items: subtract one from the user response.
- For even-numbered items: subtract the user responses from 5. This scales all values from 0 to 4 (with four being the most positive response).
- Add up the converted responses for each user and multiply that total by 2.5. This converts the range of possible values from 0 to 100 instead of from 0 to 40.

This survey was completed by 10 participants and conducted on each testing session and the SUS score was on average 84. Given that the average SUS score is 68, it was clear that the UI was well received by the users and they didn't have any difficulties understanding or using the application. However, there was a discussion with the participants in the test and stakeholders on areas to improve in the near future and what other functionalities to add. While they agreed that the interface is limited in terms of details regarding each component as it only offers an overview of the health status of each component, there should be a direct way to navigate to the source of the metrics, for example to New Relic or Couchbase clusters for the teams to investigate further once they understand which one is the faulty component.

The Application dashboard, although in early stages, was received quite well, especially the dependency graph and the users would like to see a dependency graph or service map for each service outlining all components included in an application. This would enable them to identify at which part in the process the user is having issues. Considering for example, the login process, they would like to see implemented a "customer journey" from opening the login UI to sending the request to login and point out where it failed.

Another important feature to add is connecting the application with the alerting tool PagerDuty, so that when there is a critical violation or close to be one, the responsible engineer on duty, will receive a notification that will help prevent the issue having impact on the customers.

8. Conclusions and future plans

This thesis describes a one-year work on building an APM application from scratch to monitor application and services offered by SAP Concur. The purpose is to build a centralized monitoring tool to manage high priority incidents that directly affect customers in order to reduce the time to identify faulty services and as a result the time to resolve the issues.

The process starts with analyzing the initial requirements and building a POC based on them to see if the idea is viable and how it can be put into an actual implementation. Furthermore, these requirements were changed and adapted after discussions with the stakeholders and went through a few stages of testing in order to find the best fitting functionalities.

After the POC got the green light from the stakeholders, the appropriate system architecture was built and afterwards, each component of this architecture. The main focus of this thesis is the implementation of the web application and it goes well into details also on other modules such as auto-discovery and Crossbar service, as it is crucial to understand how then the web application achieves its goal of being a real-time visualization tool. It also describes the main concepts behind technologies such as Crossbar.io router and React, in order to put the implementation in context.

Even though it is still in its early stages, Pulse goes beyond this thesis and will continue to be implemented and improved in order to achieve its goal of centralizing the monitoring process and offering both technical and non-technical teams a corporate wide view of the health of SAP Concur's services across different data centers and environments.

Bibliography

SAP Concur. 2020. SAP Concur. *SAP Concur.* [Online] 2020. https://www.concur.com/.

Firican, George. 2019. How Data Is (And Isn't) Like Oil. *Tdwi.* [Online] April 22, 2019. https://tdwi.org/articles/2019/04/22/data-all-how-data-is-like-oil.aspx.

Tukey, John. 1977. *Exploratory Data Analysis.* 1977.

Kirk, Andy, et al. 2016. *Data Visualization: Representing Information on Modern Web.* Birmingham : Packt Publishing Ltd., 2016. 978-1-78712-976-4 .

Ascombe, Francis. 1973. Graphs in Statistical Analysis. 1973, Vol. 27, 1.

Margara, Alessandro and Rabl, Tilmann. 2018. Definition of Data Streams. *Encyclopedia of Big Data Technologies.* 2018.

Adaikkalavan, Raman and Chakravarthyv, Sharma. 2006. SnoopIB: Interval-based event specification and detection foractive databases. *Data & Knowledge Engineering.* 2006, Vol. I, 59.

Dayarathna, Miyuru and Perera, Srinath. 2017. Recent Advancements in Event Processing. *ACM Computing Surveys.* 2017, Vol. 1, 1.

Saxena, Shilpi and Gupta, Saurabh. 2017. *Practical Real-Time Data Processing and Analytics.* Birminghan : Packt Publishing Ltd., 2017. 978-1-78728-120-2 .

Sommerville, Ian. 2016 . *Software Engineering.* s.l. : Pearson Education Ltd, 2016 . 1-292-09613-6.

Hoque, Sheik and Miranskyy, Andriy. 2018. Architecture for Analysis of Streaming Data. *IEEE International Conference on Software Architecture Companion (ICSA-C).* 2018.

Le, Katy. 2017. User-centered Design Method. *Medium.* [Online] November 7, 2017. https://medium.com/redcatstudio/user-centered-design-method-28e3aafc8c8a.

Norman, Donald. 1986. *User Centered System Design.* s.l. : CRC Press, 1986. 0898598729.

—. 2018. People-centered (not tech-driven) design. *Encyclopaedia Britannica.* Anniversary Edition, 2018.

—. 2019. The Four Fundamental Principles of Human-Centered Design and Application. *jnd.* [Online] July 23, 2019. https://jnd.org/the-four-fundamental-principles-ofhuman-centered-design/.

Karat, John. 1996. User Centered Design: Quality or Quackery? 1996.

—. 1997. Evolving the scope of user-centered design. *Communications of the ACM.* 1997.

Gulliksen, Jan, et al. 2003. Key principles for user-centred systems design . *Behaviour & Information Technology.* 2003, Vol. 22, 6.

Treseler, Mary. 2016. O'Reilly. *Jay Trimble on user-centered design, Agile, and design thinking at NASA.* [Online] O'Reilly, December 22, 2016. https://www.oreilly.com/content/jay-trimble-on-user-centered-design-agile-and-design-thinking-at-nasa/.

Bridle, James. 2018. Opinion: Data isn't the new oil — it's the new nuclear power. *Ted Ideas.* [Online] July 17, 2018. https://ideas.ted.com/opinion-data-isnt-the-new-oil-its-the-new-nuclear-power/.

Wikipedia. 2020. Application Performance Management. *wikipedia.* [Online] 2020. https://en.wikipedia.org/wiki/Application_performance_management.

Barlow, Buckley. 2019. What, Why, How: The Northstar Metric. *In The Know.* [Online] September 22, 2019. https://beintheknow.co/north-star-metric/.

usability.gov. 2020. System Usability Scale (SUS). *usability.gov.* [Online] 2020. https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html.

Table of Figures

List of Abbreviations

API	Application Programming Interface
APM	Application Performance Management
CEP	Complex Event Processing
DOM	Document Object Model
JS	JavaScript
KPI	Key Performance Indicator
POC	Proof of Concept
REST	Representational State Transfer
RMS	Resource Management System
SaaS	Software as a Service
SLO	Service Level Objective
SM	Service Management
SPA	Single Page Application
SUS	System Usability Scale
TCP	Transmission Control Protocol
TLS	Transport Layer Security
UCD	User Centered Design
UCSD	User Centered System Design
UI	User Interface