Scalability Performance Evaluation of the E-Ambulance Software Service

Dejan Stamenov*, Dimitar Venov†, Marjan Gusev‡
Faculty of Computer Science and Engineering
Ss. Cyril and Methodius University
Skopje, Republic of Macedonia
Email: *stamenov.dejan@outlook.com, †dimitar.venov@students.finki.ukim.mk,
‡marjan.gushev@finki.ukim.mk

Abstract—**E-Ambulance is a Software as a Service (SaaS) solution for computer assisted diagnoses of ECG recordings. It is an expert system, providing doctors and their patients a cloud solution for computer assisted diagnoses of ECG recordings and means of collaborative monitoring and treatment. In this paper, we aim at evaluating the scalability performance of the E-Ambulance SaaS solution by presenting the architecture of the SaaS application and annotation REST service for diagnoses/anamneses, followed by the testing methodology. The overall goal is to evaluate the scalability performance and find the optimal architecture that will offer highest performance for the user.**

We have developed a prototype of the main SaaS application and a prototype of one web service to be used within the SaaS application for management of annotations. Both were developed in different technologies to find out if there is any influence on the performance.

Index Terms—**Architectural performance, Cloud Computing, ECG Visualization, Load testing, REST, Service Oriented Architecture, Software As A Service**

I. Introduction

Cloud computing is the technology that offers a long-held dream of computing as a utility, which has recently emerged as a commercial reality [1]. Developers with innovative ideas for new services can use all of the advantages of cloud computing and significantly lower all of the risks accounted to developing new services. With cloud computing, developers do not need to be concerned with over-provisioning, thus, wasting costly resources, or under-provisioning and missing potential customers and revenue.

E-Ambulance is an expert system, providing doctors and their patients a SaaS solution for computer assisted diagnoses of ECG recordings and means of collaborative monitoring and treatment. Each patient is given a limited cloud storage that the patient will use to upload and organize the personal ECG recordings obtained by wearable ECG sensors. As soon as the ECG recording is uploaded onto the cloud, it is processed by the diagnoses framework and the findings from the parametrization phase are saved along with the recording. After the ECG recording has been processed, the patient is given an access to a visualized perspective of the imported data and is able to submit anamnesis for a particular and flexible time frame regarding the personal perceptions that concern the cardiac condition for the corresponding time frame. Each doctor can access the own patient's records, along with any anamnesis on the momentary condition, and is able to add information for a particular and flexible time frame regarding the diagnosis and the treatment of the interpreted condition.

E-Ambulance enables different system roles, such as various level of administrators (system, ambulance administrator, officers), doctors and patients. It is a unique system that interconnects a service for setting diagnoses and anamnesis for each ECG in the system and also, a service for ECG data visualization. Based on the amount of users that may use the system, we will evaluate scalability performance with tests to find out how the performance scales with scaling of the architecture and users. This evaluation will help us find the optimal performance metrics of the system. The provided data is captured with Apache JMeter [2], a load testing tool to analyze and measure the performance of web applications and services.

The paper structure is as follows. Related work is presented in Section II. Section III describes the E-Ambulance application and Section IV the methodology to test the scalability of the solution. The results are presented in Section V, while Section VI describes the evaluation and comparison of the results. Finally, the conclusion and future work is presented in Section VII.

II. Related Work

Almadani et al. [3] propose an E-Ambulance framework as a smart ambulance system model that provides health monitoring of patients for remote medical professionals. The main objective of the system is to provide health monitoring and auto response in terms of alerts and suggestions to paramedic stuff that work in the ambulances. Their E-Ambulance system architecture consists of multiple units where DDS middleware will take place in most of the units to hide heterogeneity of these units and control QoSs needed for the proposed system.

Ristovski et al. [4] present an ECG experts system developed as a SaaS application. They address a challenge to deal with streaming ECG data in a SaaS solution by an architecture composed of an ECG sensor used for recording, a phone used to connect to the sensor via Bluetooth to reroute the data from the ECG sensor towards a streaming unit, a data processing unit used for storing the ECG data in a database and, finally, the architecture contains a main unit used for storing the user personal data.

Pandey et al. [5] propose a Cloud based system for monitoring of patients who suffer from cardiac arrhythmias, requiring continuous episode detection where the ECG data is obtained in real-time from commodity wearable sensors. The authors describe the ECG data analyses process and how they model the problem as a workflow. The prototype system is composed of a web service which is responsible for handling the client requests, a Container Scaling Manager used for instantiating and turning off containers, Workflow Engine which is hosted in a container and it is used for managing the execution of tasks from the ECG workflow and Aneka [6] which is a workload distribution and management platform.

Two approaches are identified in [5] when testing the scalability of the application: horizontal and geographic scalability. The horizontal scalability provides the ability of a system to easily expand its resource pool to accommodate heavier load, while the geographic scalability provides better performance and usability over a given geographical area. The goal of the architecture is to seamlessly handle the changes in request patterns. The authors provide different setups of the architecture, based on which the scalability will be tested. These setups range from fixed number of Amazon EC2 resources for 25 virtual machines, to dynamic resource allocation policies and containers. In all these setups, the user requests range from 80 to 2000, gradually increasing over time. From the results of the conducted experiments they conclude that dynamical scaling results gradually increase in response time as the number of user requests increased instead of the sharp rise in response time at the beginning for all the tasks in the case when there are fixed number of worker nodes.

Vecchiola et al. [6] consider multiple cloud models when determining the scalability of their system, including Platform as a Service (PaaS) and Infrastructure/Hardware as a Service (IaaS/HaaS). When testing the scalability of the system, they consider the existing cloud platforms: Google AppEngine, Microsoft Azure and Amazon EC2. Their system provides support for distributed execution of evolutionary optimizers and learning classifiers, while yielding significant speed up on distributed environment, compared to a single machine. The results also show that for a small number of requests there is no advantage in leveraging the execution of a distributed environment.

III. System architecture

This section describes the overall architecture of the two system modules. The first one is the *front-end Software as a Service (SaaS) application* used for initial communication with users, and the second one with a *distributed processing unit* to allow monitoring access of the ECGs. Due to large storage and processing demands, a distribution unit can provide services only for a limited number of users. The main system is scalable so that if there are more users, then another distributed processing unit can be added.

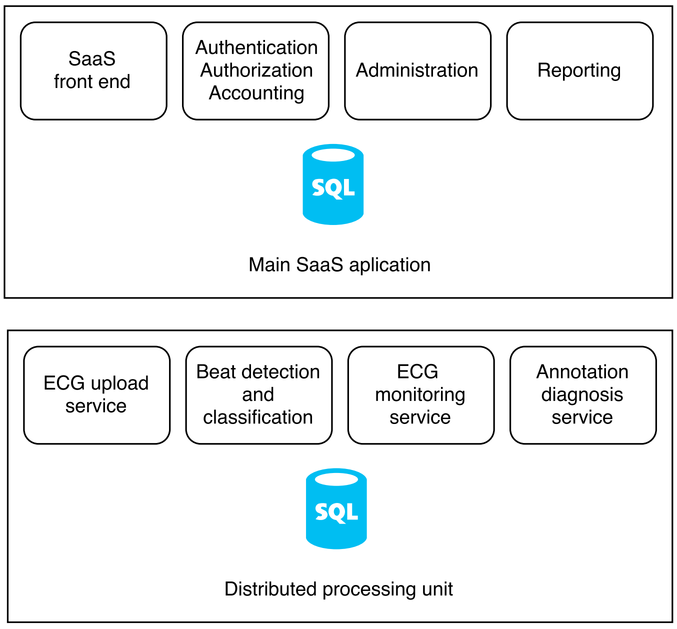
SaaS front end
Authentication Authorization Accounting
Administration
Reporting
SQL
Main SaaS aplication
ECG upload service
Beat detection and classification
ECG monitoring service
Annotation diagnosis service
SQL
Distributed processing unit

Fig. 1. Overview of the E-Ambulance system architecture

This separation of the main application and distributed processing unit was done in order to enable a common entry point for the main SaaS application, and to distribute the processing of a very large quantities of ECG data. To illustrate the operation of the distributed processing unit we have implemented a web service for submitting and updating diagnoses, therapies and anamneses.

Figure 1 presents a logical framework of the main SaaS eAmbulance application and one distributed processing unit with various web services for ECG processing and monitoring.

In case of a small number of users one virtual machine instance can host the main SaaS application unit, and another one can host the distributed processing unit.

A. *Software as a Service system architecture*

The Software as a Service (SaaS) software distribution model is used in the development of the E-Ambulance application, in which all of the components are designed to be services [7]. The core of the architecture is an ASP.NET web application, implemented with Model-View-Controller (MVC) pattern. The nature of the project defines strict rules realized by corresponding MVC patterns.

The system is personalized for any ambulance that is registered at the system, thus requiring full control over the rendered HTML, ease of integration of different frameworks and following the nature of the stateless web principles.

The SaaS application uses a separate Microsoft SQL Server database for controlling user access, ambulance personalization and overall work flow of the system. Relational database is chosen as a primary database of the system, because of the strict rules and personalization for each ambulance and user.

External services interconnect with the SaaS application, which provide utilities for visualizing ECG data, beat detection and classification and also, service for saving diagnoses and anamnesis. These services are part of the distributed processing unit.

B. *Annotation service for diagnoses, therapy and anamnesis*

A REST service is used for creating anamnesis information by the patients and also for creating diagnoses and therapies by the doctors for a given time span. Patients can access ECG recordings and browse diagnoses and recommended therapies based on query parameters.

The service is a web application written in the programming language Python [8] using the Django Web Framework [9]. Additionally, we were using the Django Rest Framework [10] for easier definition of the REST APIs for the service.

The service is composed of two models, *Diagnoses* used for storing the diagnoses and *Comments* used for storing anamnesis. As a database for the models we were using the MySQL database [11] and the models were stored in two separate tables.

The *Diagnoses* model is composed of *id*, *created_timestamp*, *from_timestamp*, *to_timestamp*, *diagnoses*, *patient* and *doctor* fields. The *id* field is the unique identifier for the resource. The *created_timestamp* field is a time stamp that marks when the resource was created. The *from_timestamp* and *to_timestamp* fields are used to mark the time span of the diagnoses. The *diagnosis* field is the actual text of the diagnosis. The *patient* and *doctor* fields are the ids of the patient and doctor accordingly. Listing, creating, updating and deleting of resources (diagnoses) is done in a RESTful fashion.

The *Comments* model is composed of *id*, *created_timestamp*, *from_timestamp*, *to_timestamp*, *comment* and *patient*. The *id*, *created_timestamp*, *from_timestamp*, *to_timestamp* and *patient* fields are same as in the *Diagnoses* model. The *comment* field is the actual text of the anamnesis. Same as in the *Diagnoses* model, listing, creating, updating and deleting of resources (comments) is done in a RESTful fashion.

IV. Testing Methodology

The testing methodology contains description of the experiments, testing environment, test cases and test data for defining the load testing of the E-Ambulance application and its services.

For this testing we have measured data over two instances, the first one is the response of the SaaS application, and the second one to the annotation web service.

The goal of this testing is to find out how many concurrent users can simultaneously use the SaaS application and web service. This is based on the most common scenario in the system, where each doctor and patient will constantly provide details about the health condition, based on the analyzed ECG parameters. In such scenario, we want to keep the performance of the system optimal, no matter the number of the users actively using the system.

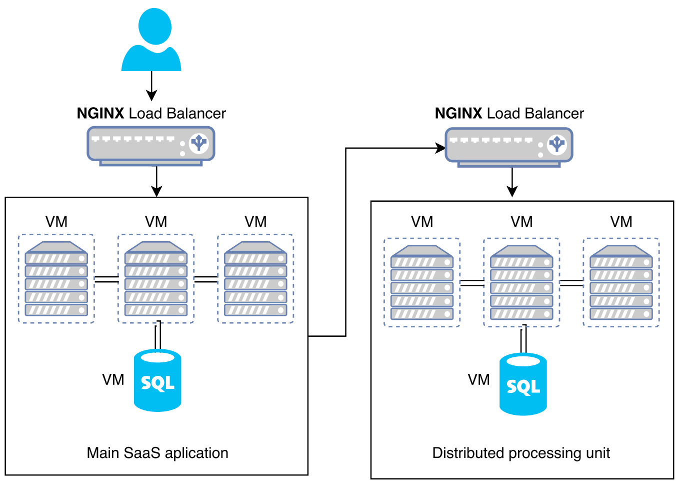

Fig. 2. Testing environment: Hosting of the E-Ambulance software modules

A. *Testing environment*

Figure 2 provides a visual presentation of the testing scenario. Each part of the architecture requires at least one virtual machine to be running; more virtual machines are added as the service demand increases by the users. Additionally, we have set the database server to be hosted on a separate virtual machine.

The database of the application was hosted on a single dedicated (virtual) server, while the application itself was hosted on 3 dedicated (virtual) servers. The overall architecture of the application included NGINX [12] load balancer, with three testing scenarios being simulated, where each application server used the same database server:

- *Scenario 1* using 1 application server and 1 database server;
- *Scenario 2* using 2 application servers and 1 database server;
- *Scenario 3* using 3 application servers and 1 database server.

All of the servers have 1 CPU, 1 vCPU with 2GB RAM. Windows Server 2012 was used as the main operating system for hosting both the application and the database.

The load testing experiments conducted on the main SaaS application were based on a standard user sign in scenario, based on both *GET* and *POST* methods (the former retrieves the sign in page from the server, while the latter posts the users' specific data to the server; an exact simulation of the real behavior of an user).

The experiments for load testing of the distribution processing unit were using only the service for diagnoses and anamnesis. Testing was composed of making *GET* request to either retrieve an anamnesis resource or a diagnoses resource and *POST* request to either create an anamnesis resource or a diagnoses resource by variable number of users.

The service architecture consists of a load balancer (reversed proxy) for routing the requests, (virtual) servers which were hosting the application service and serving the requests and one (virtual) server which hosted the MySQL database server. The architecture of the service for diagnoses and anamnesis is shown in Figure 2. As a load balancer we were using the NGINX reverse proxy server and it was running on a (virtual) server with 2 CPUs, 4 vCPUs and 2GB RAM (same configuration is used for the SaaS architecture). As a routing logic, we were using *least-connected* logic, which means that the next request will be routed to the server with the least number of active connections.

The application service was running on a (virtual) server with 1 CPU, 1 vCPU and 2GB RAM. Finally, the MySQL database server was running on a (virtual) server with 1 CPU, 2 vCPUs and 2GB RAM. The service architecture experiments are based on the same scenario (setup) as the SaaS architecture.

Experiments were conducted using the Apache JMeter software. The testing environment was created and is being hosted on the cloud at the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University in Skopje, Republic of Macedonia.

B. *Test cases*

The load testing simulation of the SaaS application was done based on different number of users trying to sign in, sending requests (both *GET* and *POST*) for one minute, for each architectural scenario. And the load testing simulation of the service for diagnoses and anamnesis was done based on different number of users making both *GET* and *POST* requests for five minutes, for each architectural scenario.

We started the test off with 10 users, then increased to 50 (the service for diagnoses and anamnesis directly started from 50 users) and 100, 200, and so on until 1000 users (increasing by 100 users each time). All the requests that were created by these users were sent simultaneously.

C. *Test metrics*

In order to figure out how the architecture will scale based on the different test scenarios and to be able to compare the results, we were measuring the minimum, maximum and average response time in each of the given scenarios for both the SaaS application and web service. In addition, we were measuring the throughput expressed in requests per second.

We have completed at least 5 measurements for each test case and calculated the average values. Differences in the measured values are expected due to various behavior and current state of the overall cloud where our testing environment was hosted.

The measurements included the response time T in seconds and server throughput V in requests per second. Values for each scenario are determined by its index 1, 2 and 3. The overall speedup of scenarios 2 and 3 against scenario 1 is given by (1) when response time is analyzed and in (2) for throughput (velocity). The normalized speedup is given by the equation (3), where m and t are the indexes of the scenarios, for both the response time and velocity. Equations in (4) provide the normalized speedup of scenario 2 and 3 against scenario 1, measured by the response time. The equations in (5) provide the normalized velocity speedup, accordingly.

$$S_{12}(T) = \frac{T_1}{T_2} \qquad S_{13}(T) = \frac{T_1}{T_3} \tag{1}$$

$$S_{12}(V) = \frac{V_2}{V_1} \qquad S_{13}(V) = \frac{V_3}{V_1} \tag{2}$$

$$NS_{mt} = \frac{S_{mt}(T)}{t} \qquad NS_{mt} = \frac{S_{mt}(V)}{t} \tag{3}$$

$$NS_{12} = \frac{S_{12}(T)}{2} \qquad NS_{13} = \frac{S_{13}(T)}{3} \tag{4}$$

$$NS_{12} = \frac{S_{12}(V)}{2} \qquad NS_{13} = \frac{S_{13}(V)}{3} \tag{5}$$

The higher the speedup, the better the performance is.

V. Results

The results will be presented in charts where the blue line with circle markers corresponds to Scenario 1 (1 application server), the orange line with square markers to Scenario 2 (2 application servers) and finally the gray line with triangle markers to Scenario 3 (3 application servers).

A. *Performance of the SaaS application*

Figure 3 provides a visual representation of the average response time of the three testing scenarios, and Figure 4 for the test scenarios based on requests per second.

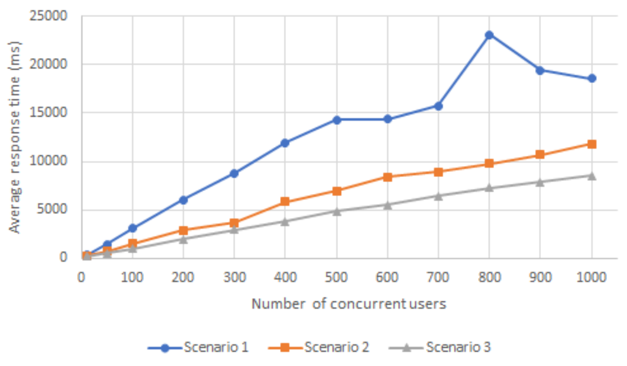

Fig. 3. Average response time of the front-end main SaaS application.

From the results presented in Figure 3, we observe that the number of application servers impacted the response time of the SaaS application. The response times when using only one application server are obviously larger than the response times when using two application servers which are also larger than the response times when using three application servers.

Next, we can observe similar improvements in the Figure 4 where the requests per second are presented for the different test cases. The lowest number of requests per second is observable when there is only one application server (blue line in the figure). When using two application servers (orange line in the figure) there is clearly improvement over using one

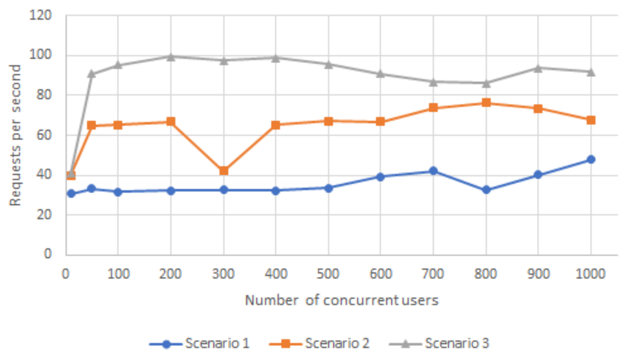

Fig. 4. Requests per second of the front-end main SaaS application.

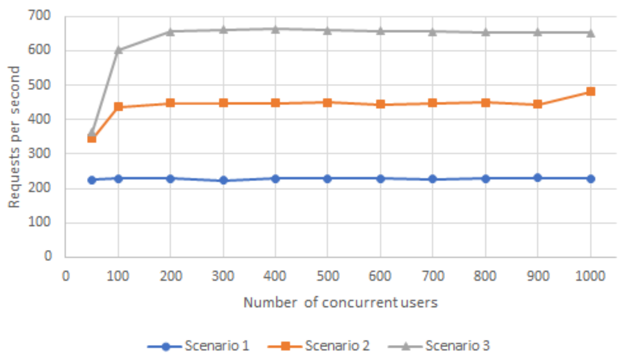

Fig. 6. Requests per second of the distributed processing unit.

application server and also we can see the same when using three application servers (gray line in the figure). For example, when there are 1000 concurrent users, there are 48 requests per second when using only one application server, there are 68 requests per second when using two application server which is 1.42 times improvement over one application server, and there are 92 requests per second when using three application servers which yields an improvement of 1.92 times over one application server.

B. *Performance of the services*

Figures 5 and 6 provide visual representation of the average response time of the service and the requests per second respectively for the three testing scenarios.

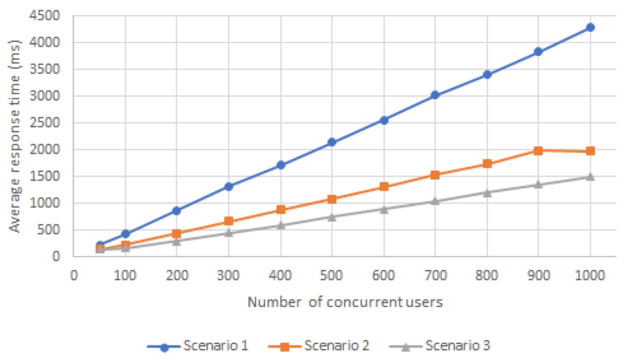

Fig. 5. Average response time of the distributed processing unit.

Improvements similar to the main SaaS application are observed for the web service, as presented in the figures (Figure 5 and Figure 6) when load testing was performed on the service for diagnoses and anamnesis.

Figure 5 presents the average response times of the service for the different test cases. It is obvious that the lowest response time is for scenario three.

A similar behavior is observed in Figure 6 that presents the requests per second of the service for the different test cases. The highest number of requests per second are achieved when there are three application servers (gray line in the figure) and the lowest number of requests per second are achieved when there is only one application server (blue line in the figure). For example, when there are 1000 concurrent users, one application server can process 228 requests per second, two application servers can process 481 requests per second which is 2.1 times higher than one application server and finally, three application servers can process 652 requests per second which is 2.86 times higher than one application server.

VI. Discussion

In this section we will present and analyze the normalized speedups we observed from the experiments that we have conducted. Additionally, we will give brief discussion of the results presented in Section V.

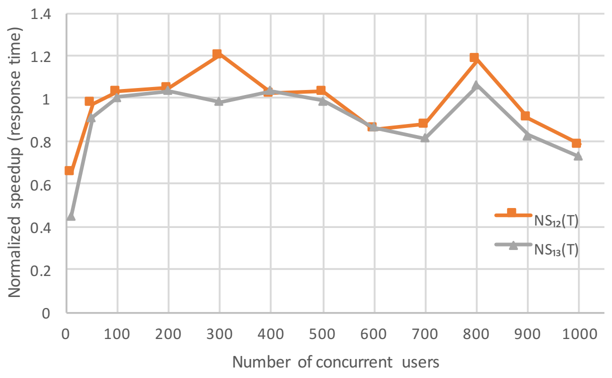

Fig. 7. Normalized speedups $NS_{12}(T)$ and $NS_{13}(T)$ of the front-end SaaS application (comparison of scenarios 2 and 3 against scenario 1).

The normalized average response time speedup for the SaaS application is presented in Figure 7 and Figure 8 gives visual representation of the normalized requests per second speedup. From the figures can be observed that the speedup is same for both of the scenarios until 500 concurrent users. When there are 500 or more concurrent users, Scenario 3 has worse speedup than Scenario 2. This is expected behavior, because all of the application servers are using the same database.

Similar behavior can be observed for the service as well. Figure 9 gives visual representation of the normalized average

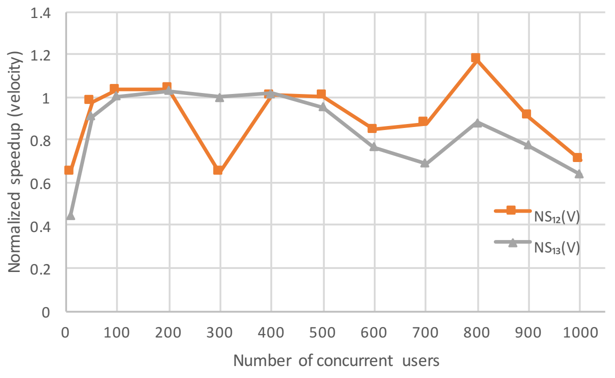

Fig. 8. Normalized speedups $NS_{12}(V)$ and $NS_{13}(V)$ of the front-end SaaS application (comparison of scenarios 2 and 3 against scenario 1).

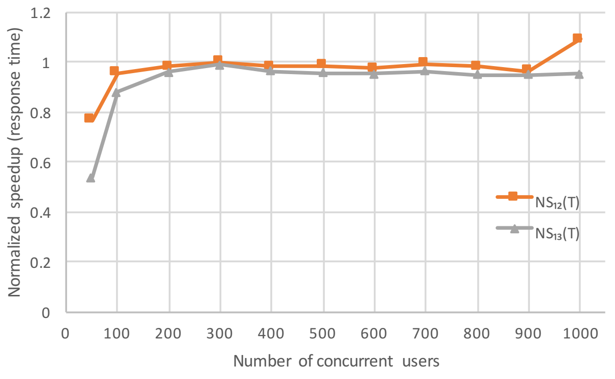

Fig. 9. Normalized speedups $NS_{12}(T)$ and $NS_{13}(T)$ of the distributed processing unit (comparison of scenarios 2 and 3 against scenario 1).

response time speedup for the service, and Figure 10 gives visual representation of the normalized requests per second speed up of the service. As we can see from the figures, the speedup is same for both of the scenarios until 900 concurrent users, and after that, Scenario 3 has worse speedup than Scenario 2.

From the results presented in Section V we can also conclude when a new application server can be added so we would get better response times and better requests per second. From Figure 3 and Figure 4 we can figure out that when there is only one application server, a new application server should be added when there are 50 concurrent users, and when there are two applications servers, a new application server should be added when there are 200 concurrent users. From Figure 5 and Figure 6 we can see that when there is only one application server, a new application server should be added when there are 50 concurrent users, and when there are two application servers, a new application server should be added when there are 100 concurrent users.

VII. Conclusion and Future Work

In this paper, we have presented the scalability performance evaluation of the E-Ambulance application as a system for

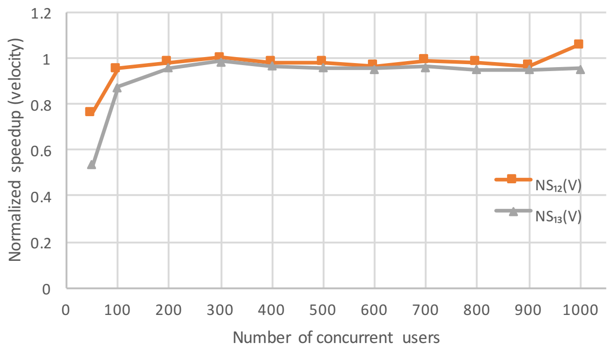

Fig. 10. Normalized speedups $NS_{12}(V)$ and $NS_{13}(V)$ of the distributed processing unit (comparison of scenarios 2 and 3 against scenario 1).

computer assisted diagnoses of ECG recordings, by describing the software application architecture, methodology for conducting experiments, presenting the testing results and their evaluation. The testing tool for conducting load tests was Apache JMeter. From the discussion in Section VI, we can clearly conclude that the overall architecture scales well, while preserving the optimal performance for the users.

As our future work, we would like to test how different storage technologies for preserving the ECG data can impact on the overall performance of the system. We will conduct experiments in order to find out the scaling of different storage architectures, based on different number of concurrent users.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] E. H. Halili, *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites.* Packt Publishing Ltd, 2008.

[3] B. Almadani, M. Bin-Yahya, and E. M. Shakshuki, "E-ambulance: Real-time integration platform for heterogeneous medical telemetry system," *Procedia Computer Science*, vol. 63, pp. 400–407, 2015.

[4] A. Ristovski and M. Gusev, "Saas solution for ecg monitoring expert system," in *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2017 40th International Convention on.* IEEE, 2017, pp. 308–313.

[5] S. Pandey, W. Voorsluys, S. Niu, A. Khandoker, and R. Buyya, "An autonomic cloud environment for hosting ecg data analysis services," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 147–154, 2012.

[6] C. Vecchiola, X. Chu, and R. Buyya, "Aneka: a software platform for .net-based cloud computing," *High Speed and Large Scale Scientific Computing*, vol. 18, pp. 267–295, 2009.

[7] A. Fox and D. Patterson, *Engineering Long-lasting Software: An Agile Approach Unsing SaaS and Cloud Computing.* Strawberry Canyon LLC, 2012.

[8] G. Van Rossum *et al.*, "Python programming language." in *USENIX Annual Technical Conference*, vol. 41, 2007, p. 36.

[9] D. W. F. Team, "Django web framework."

[10] T. Christie, "Django rest framework," 2015.

[11] A. MySQL, "Mysql database server," *Internet WWW page, at URL: http://www. mysql. com (last accessed/1/00)*, 2004.

[12] C. Nedelcu, *Nginx HTTP Server: Adopt Nginx for Your Web Applications to Make the Most of Your Infrastructure and Serve Pages Faster Than Ever.* Packt Publishing Ltd, 2010.