FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Mikko Kaasila
Mikael Pennanen

DECOUPLING BETWEEN LOVELACE'S CHECKER SERVER AND MAIN SERVER

Bachelor's Thesis
Degree Programme in Computer Science and Engineering
June 2022

Kaasila M., Pennanen M. (2022) Decoupling between Lovelace's Checker Server and Main Server. University of Oulu, Degree Programme in Computer Science and Engineering, 41 p.

ABSTRACT

Thesis provides an introduction to online learning environment Lovelace, defines its coupling issues, and narrates the process of decoupling those issues. The tools that are used in the Lovelace system's components main features and their role in the system are described and the term decoupling is explained to the reader. The system has a coupling issue between the main server and checker service; checker service has read and write access to the main server's database and that needs to be revoked. In addition, Lovelace system utilizes network file system to share files between the checker service and the main server.

For decoupling the system, a solution is designed and implemented that revokes those read and write rights and also strips the need for the network file system. Solution for the issue has a three-step design where iterations are in order of importance. Iterations of the design are: revoking the write access; revoking the read access; and lastly the consideration regarding the use of network file system. Implementation of the solution consist of creation of the development platform and each iteration of the implementation design.

Evaluation is structured in a similar manner as the implementation. Each iteration of the implementation is evaluated as its own and evaluation is also given to the development platform. The evaluation itself consists of discussion and observations that are made from the implementations and its outcomes.

Keywords: coupling, decoupling, system architecture

Kaasila M., Pennanen M. (2022) Lovelacen tarkastinpalvelimen ja pääpalvelimen irtikytkentä. Oulun yliopisto, Tietotekniikan tutkinto-ohjelma, 41 s.

TIIVISTELMÄ

Tutkielma esittelee verkko-oppimisympäristö Lovelacen, määrittelee oppimisympäristön kytkentäongelmat sekä kuvailee irtikytkennän vaiheet, että näiden vaiheiden ongelmat. Lovelace-järjestelmän komponentit ja niiden olennaisimmat ominaisuudet, sekä rooli järjestelmän toiminnassa on selitetty, ja termi irtikytkentä selitetään lukijalle. Järjestelmässä on kytkentäongelma pääpalvelimen sekä tarkistuspalvelimen välillä. Tarkistuspalvelimella on sekä luku- että kirjoitusoikeudet järjestelmän pääpalvelimen tietokantaan, jotka täytyy kumota. Järjestelmä jakaa myös tiedostoja lähiverkon yli palvelimelta toiselle, joka on osa palvelimien tiedonvaihtoa.

Irtikytkentää varten suunnitellaan sekä toteutetaan ratkaisu, jolla tarkistinpalvelimen luku- ja kirjoitusoikeus pääpalvelimen tietokantaan evätään ja tiedostojen jaon tarve verkon yli lopetetaan. Toteutuksen ratkaisumalli on kolmeportainen, tärkeysjärjestykseen listattuna. Ratkaisumallin vaiheet ovat: tarkistuspalvelimen kirjoitusoikeuden kumoaminen, tarkistuspalvelimen tietokannan lukuoikeuden kumoaminen sekä tiedostojen verkon yli jakamisen tarpeellisuuden arviointi. Projektin toteutus koostuu kehitysympäristön rakentamisesta, sekä ratkaisumallin jokaisesta vaiheesta.

Toteutuksen arviointi on jäsennelty alkuperäistä ratkaisumallia vastaavasti, arviointi tapahtuu erikseen jokaiselle vaiheelle sekä kehitysympäristö arvioidaan. Arviointi koostuu keskustelusta sekä havainnoista, joita toteutuksesta havaitaan.

Avainsanat: kytkentä, irtikytkentä, järjestelmäarkkitehtuuri

TABLE OF CONTENTS

FOREWORD

We would like to thank Mika Oja for supervising our work. The given subject was interesting to work with.

Oulu, June 29th, 2022

Mikko Kaasila
Mikael Pennanen

LIST OF ABBREVIATIONS AND SYMBOLS

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ID	Identification
I/O	Input/Output
JSON	JavaScript Object Notation
ms	Millisecond
NFS	Network File System
RAM	Random Access Memory
Redis	Remote Dictionary Server
RESP	Redis Serialization Protocol
REST	Representational state transfer
SoC	Separation of Concerns
SQL	Structured Quary Language
TCP	Transmission Control Protocol
YAML	YAML Ain't Markup Language

1. INTRODUCTION

Software systems' maintainability along with relative performance is highly dependent on systems' architecture i.e. how the components are built and what kind of relations they have with each other. Coupling measures how tightly system components are connected. When components are dependent and their changes directly affect other components you would call system's coupling tight. If that is not the case and the components are not very dependent on each other the system is loosely coupled. When tight coupling occurs in software's architecture, it is harder to make changes to the parts of the system without affecting those strongly coupled parts. In this kind of tightly coupled architecture components must know other components' implementations so there would be no malfunctions during component changes and syncing, reducing software's maintainability. To fix the issue the system should be decoupled. The goal with decoupling is to make the components more independent. This increases the modularity of the complete software.

This Bachelor's Thesis focuses on decoupling the Lovelace learning environment's checking processes from the main server. The goal is to eliminate the need for database access for the checking service. Implementing the decoupling requires us to familiarize with the requirements of the system environment. Once sufficient system knowledge has been acquired, we can start designing a new protocol for the communication between the main server and the checker service. This protocol should include a format for both requests as well as responses between these two services. To implement this, we must study the checker tasks and find what information is needed for the checker to be able to work.

Another goal is to examine and assess the need to decouple the connection between checking service and the main server's network file system (NFS). As we design the decoupling of the main Django server from the checker tasks, the requirement of decoupling the NFS from the checker will be assessed as well. In case the solution of decoupling the NFS is found practical, this implementation will be documented as well. In the next chapters Lovelace system and its components are introduced along with the related work regarding decoupling. The design section contains the process of designing the implementation together with the actual design of our planned decoupling. Analysis of the design will be included, where we evaluate and reflect the plan.

2. RELATED WORK

2.1. Decoupling

When talking about decoupling a client and server, the basic idea is to add more independence to their configurations, so that every change in configuration does not need to be done for both of the client and server[1]. This elevated state of independence also gives an opportunity to implement more comprehensive mandatory testing to each individual component as needed more easily. Increased independence on the main server enables it to be updated individually without influencing the client directly. It can continue communicating through the same interface as before agreed between these services. This means that the server takes the responsibility of the changes implementation and compatibility.

Benefit of the immutable Application Programming Interface (API) between components is interchangeability. As long as communication is handled gracefully and with the right syntax considering the current situation, the different components do not need to know the inner workings of each other. Figure 1 represents the basic idea of communication between two servers via API. As long as the request of Server 1 is done with correct formatting, for example a JavaScript Object Notation (JSON), it does not matter if the Server 2 has been modified or replaced entirely with new component, as long as it can handle the request like before. This greatly reduces programming overhead, as changes are made only to independent components communicating with other parts of the system.

The act of decoupling continues a popular idea about Separation of Concerns (SoC). This means that a software implements separation between parts of itself, and these isolated sections manage the problem in hand[2].

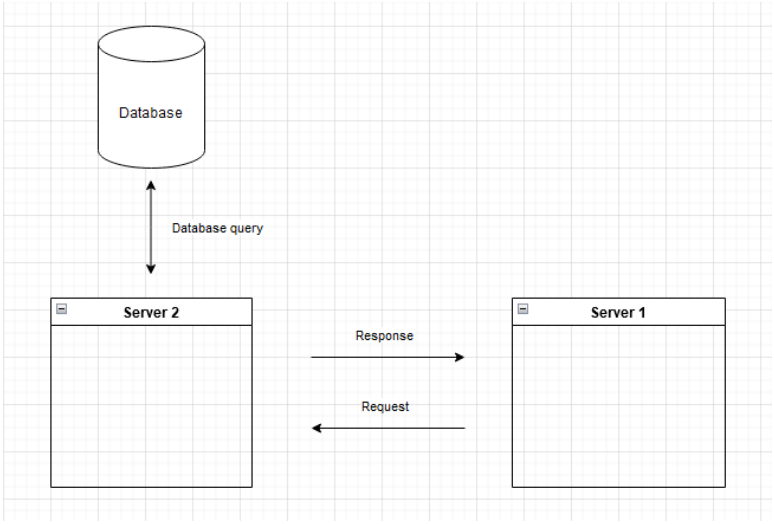

Figure 1. Request for a database query between two servers through REST API

2.2. Lovelace

Lovelace is a learning environment aimed for students to learn programming through different courses and exercises[3]. The platform allows a user to log in and attend courses of their liking. Exercises where students need to return code files as answers are offered in the courses and users can submit their answers in the website. When submitted, the answers are processed by automated checkers and the user will get an evaluation of the answer. Automated checkers are run by the checking server.

The checking process starts with the server getting information from the main Django server regarding the specific answer submitted. This information consists of several types of data, including but not limited to, identification (Id) information of the returned exercise, Id of the answer for this particular exercise as well as revision number. Id of the exercise is used for obtaining the information of the currently evaluated exercise. Id of the sent exercise tells the checker server what exercise files to gather for evaluation. Correct versions of the answers Id along with exercises Id are obtained using the revision number. The aforementioned information is gathered from the database. The result of this database query contains instructions what are the file paths for the needed files, as well as the stages of the evaluation process along with the needed commands. The required files are copied into a temporary folder in the checking server. After copying the files, the configured commands are run to perform the evaluation of the exercise. At the end of the evaluation process the checker server writes the result of the run tests to the database.

Checker requires required file paths from the database to function so reading rights are required to the database. Also, saving evaluations requires writing rights to the database. In addition to the checkers, Lovelace operates on top of several components in order to make the platform function properly. At current state components of the checking server and the main server are in a need of decoupling. Figure 2 shows the current system architecture of the Lovelace environment. As the figure shows, Django checking service has an access main server's database. The decoupling will be introduced later in this chapter. Essentially for the project this means that a new design of connection should be made between the two components to ensure better security and maintainability. Most relevant components and principles for this project are listed below.

2.3. Django

Lovelace is built with web development framework Django, which is implemented on Python programming language [4]. Django is a popular framework for web development that offers manageable up scaling by having highly independent architecture. This enables adding or removing various parts of a system, such as a caching server for high activity sites where there is heavy traffic. This independence also makes Django highly modifiable so it can be used for a large number of different applications and services.

Django framework offers reasonably good security by default[5]. Notably for this project Django features a Structured Query Language (SQL) injection protection. This

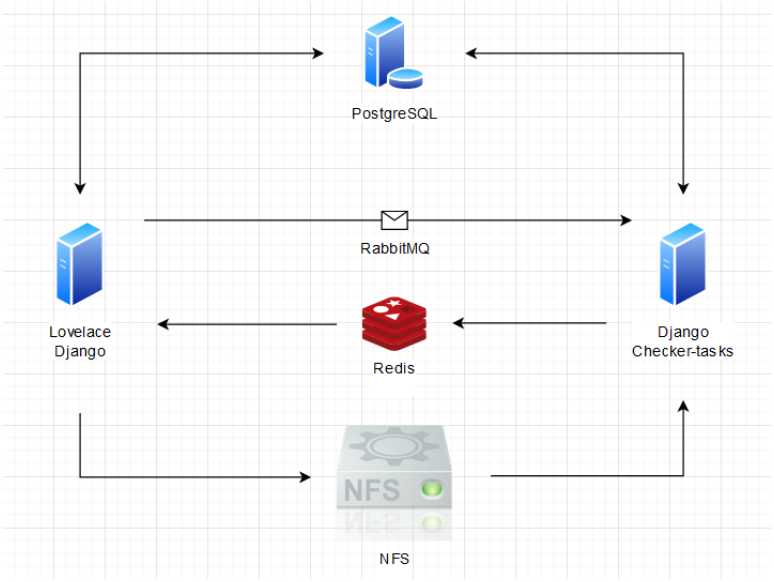

Figure 2. Current visualization of the system architecture presented as flowchart.

means that an attacker can not run hostile SQL code against a database, which could result in deletion of data or lost integrity by illegally fetched data. Django also has a security solution for Cross site request forgery (CSRF) that can be enabled. This solution generates a CSRF token, each time a form is created, to prevent malicious parties from sending unauthenticated data in behalf of the legitimate user.

Essential feature of Django to get Lovelace up and running are tasks that are run with Celery. With them Lovelace runs its courses' checkers, tools for teachers and statistics about tasks. As said, tasks are run with Celery and are described in more detail in the Celery sections.

2.4. Remote Dictionary Server

Remote Dictionary Server (Redis) provides a way for clients to gain unified access to data structures in a form of memory caching server. This is implemented with usage of Transmission Control Protocol (TCP) sockets combined with REdis Serialization Protocol (RESP). RESP is a protocol created for Redis, featuring simple and human readable format[6]. Redis can serve content from Lovelace more quickly, since the pre-rendered static Hypertext Markup Language (HTML) of the website can be cached so that they do not have to be loaded every time a client asks for a specific site,if loaded before. Since static HTML forms the majority of the Lovelace's web pages, the caching works well. This boost of performance can be applied to other things, such as managing client sessions and fetching data from database.

Offering great performance, Redis is a popular choice of framework[7]. What allows Redis to increase performance is the caching abilities of the server. This means that Redis can save the database queries and other type of requests to the servers Random Access Memory (RAM). Fetching this information from the RAM of the server is quicker than having to do computationally expensive database queries every time information is needed from the main database. The Lovelace checker server uses

Celery that is configured to use Redis to communicate results of the exercises back to the main Lovelace Django server after checking the user submitted exercise.

2.5. RabbitMQ

RabbitMQ is a multi-protocol supported messaging broker[8], which is a software designed to deliver and receive messages between consumers and producers. Lovelace delivers messages from the main server to the checker server using RabbitMQ. Using RabbitMQ these servers have a common ground to pass messages and to await responses from one another. RabbitMQ's supported messaging protocols are Simple Text Oriented Messaging Protocol, MQ Telemetry Transport, Advanced Message Queuing Protocol 1.0, Advanced Message Queuing Protocol 0-9-1[9] and the protocol used in Lovelace is AMQP 0-9-1. The protocol gathers the messages into a exchange where they are distributed within given rule-sets for consumers to fetch or subscribers to be delivered [10]. To counter the anomalies and networks' unreliability, AMQP has message acknowledgements implemented. Acknowledgements are sent back to broker from the receiving end of message transfer to notify that the message is received. When notified, the message can be removed. If any anomalies occur in message delivery and acknowledgment is not receive by the broker, the message will not be removed. It will be returned back to the queue for re-sending. In Lovelace RabbitMQ is used via Celery which is used as a worker management tool for python tasks and will be explained in more detail in the next section.

2.6. Celery

Celery is an asynchronous task queue that is used with RabbitMQ. Celery tasks are defined by callable classes, which may be used like regular functions. Tasks send messages to workers and it can define what is happening when the message is sent and when the message is received. In Celery, every task is uniquely defined. Workers are parent processes that are run when a worker is run. These workers handle and process the given tasks that are queued. Tasks stay in the queue until the message that has been sent is acknowledged by a worker.[11]

Celery carries out tasks that are given to the broker. To be more precise, Celery needs a broker so it can send and receive messages. Celery supports multiple brokers so a choice between them must be made[12]. There are two brokers that are feature ready for Celery and they are RabbitMQ and Redis which are both used in Lovelace. With Celery, tasks can be queued and scheduled efficiently and therefore main server can well maintain connection to the checking server[13].

Django-celery is a package that integrates the use of Celery with Django. Newer versions work out from the box with Django but since the system is using an older version, a separate library is needed. With the package installed it is possible to run tasks with Celery in the Django app. Celery instance is initialized within a celery.py file. In Lovelace Celery gets its configurations from the configuration file where are the Django configurations imported from app's own settings. The tasks that Celery are

given to run are also configured in the file. With given information Celery auto-detects the task directories where to run the discovered tasks. Tasks are located inside the app directories in a tasks.py file. [14]

In Lovelace our installed apps are "routine_exercise", "courses", "teacher_tools" and "stats". Each apps' tasks are quite self-explanatory. Routine_exercise's and Courses' tasks handle the users' answering to different types of exercises. There are tasks considering the returning of answers, handling the different exercise stages and database interactions which is a key point of interest in our project. There are tasks that are running writing commands on the main server database and that must be changed. Teacher_tools offer key tools to course staff for monitoring and communication. Stats' tasks provide statistics about exercises and students.

2.7. PostgreSQL

PostgreSQL is an open-source object-relational database. Object-relational database means that usage of objects and inheritance is supported in the database queries and database schemes. It is commonly used and popular database[7], following the SQL standard closely, but does not fully conform to this standard[15].

PostgreSQL is the database used by Lovelace. Django is officially supporting PostgreSQL database, this makes it a good choice, since the support is not dependant by a third party support. Due to Django's automatic connection management to a database, it is advisable to use fully supported database, which PostgreSQL belongs to, to not lose any of these functions. Benefits of using PostgreSQL include ability to set custom data types, which can be defined by the user, along with great variety of regular supported data types such as JSON and basic primitive types like Integer and Numeric values[16]. PostgreSQL also supports many security features, like support for authentication and multi-factor authentication[16].

2.8. Network File System

Network File System is a distributed file system[17]. NFS is implemented as client-server model[18], which means that the NFS server can manage authorization and authentication of its' clients. This allows resources to be shared within a network, and makes management of the file system centralized across the clients. Using NFS, clients can use shared folders like the folder was located in the client machine[17]. Clients are the machines that consume the shared resources from the models other party, which are called servers[18].

In the current version of the system architecture of Lovelace the main server is hosting the NFS and the Checker task server has access to the NFS. NFS offers access to some essential directories from the main server disk to the checker service. If there will be need to decouple these two services later on in this project regarding the NFS and the state of coupling, the implementation will be documented. As the only client of the main NFS server is currently the checker service, this would render the hosting

and sharing of the main server resources useless, thus removing the need to share the disk all together.

3. DESIGN

3.1. Design Goal and Priorities

Goal of the design was to find a solution where the checker service can operate and complete its tasks without having an access to the main server's database. To accomplish this we had to study how the checking service operates and uses the database. The main point was to isolate what data the service needs from the database to complete its tasks. When that data was isolated our team had to design a format for the data request which was suitable for all tasks. We had to find a solution to request that data from the main server. We wanted to implement that solution using existing technologies that were used in Lovelace. Main server communicates with the checking server with Redis and Celery with RabbitMQ as its engine so we considered using them for the task data transferring. When sending the evaluation request to the checker service from the main server, the request should already contain the needed information for the checker server to be able to evaluate the user submitted exercise. After evaluation process is complete, the checker service returns the results to the main server via Redis, instead of writing results to the database. This implementation is divided to several stages, explained fully in the Implementation Design section.

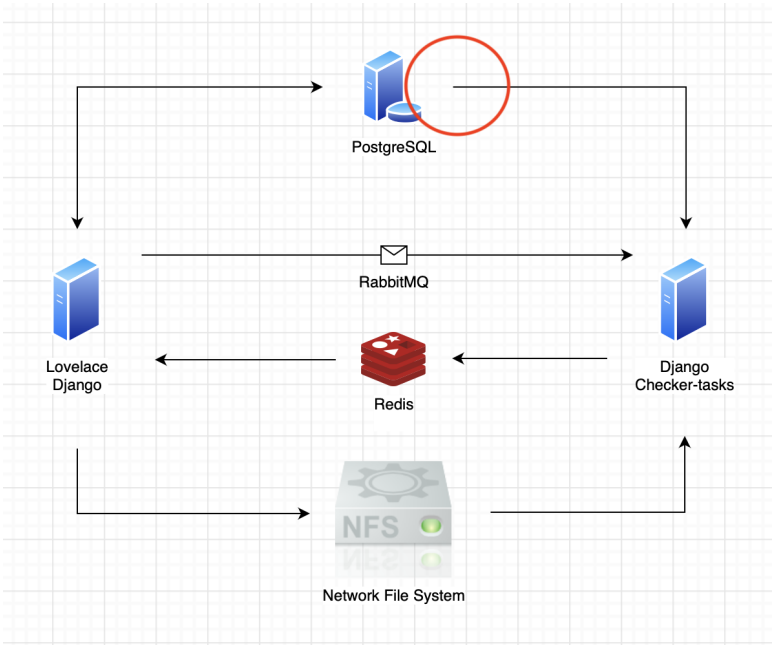

Figure 3. Visualization of first priority implementation; checking server loses its write access to database.

As seen on figure 3 we can see the first priority of decoupling. We need to revoke checker's write access to database and implement same feature through the main server via requests. Figure 4 visualizes the second priority which is to eliminate the reading right of the checker service. That feature needs also to be implemented through the main server. The last priority was to examine the need for NFS and possibly eliminate the access of checker to it. Same files would be offered from the main server as seen in figure 4. If seen plausible its implementation will be documented within given time.

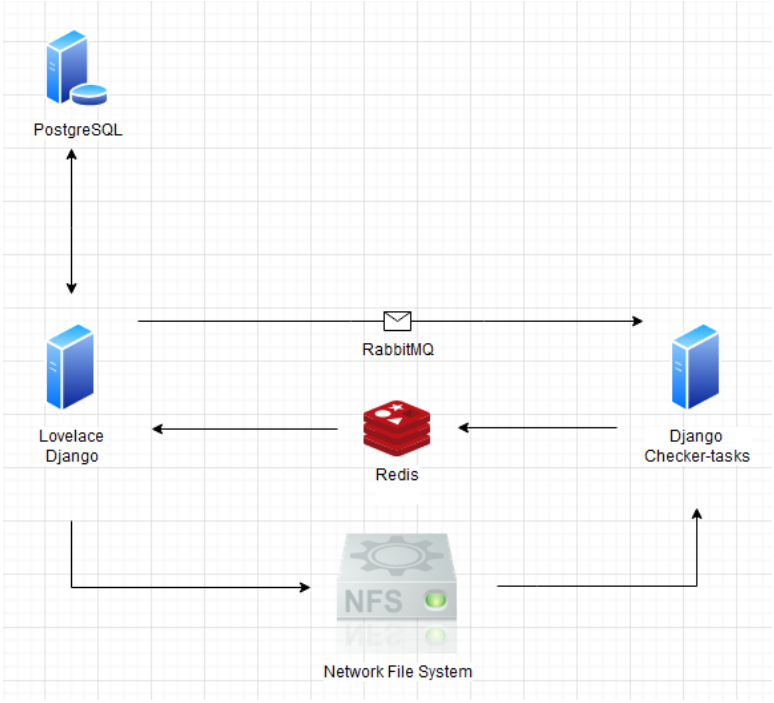

Figure 4. Visualization of second priority implementation; checking server loses its read and write access to database.

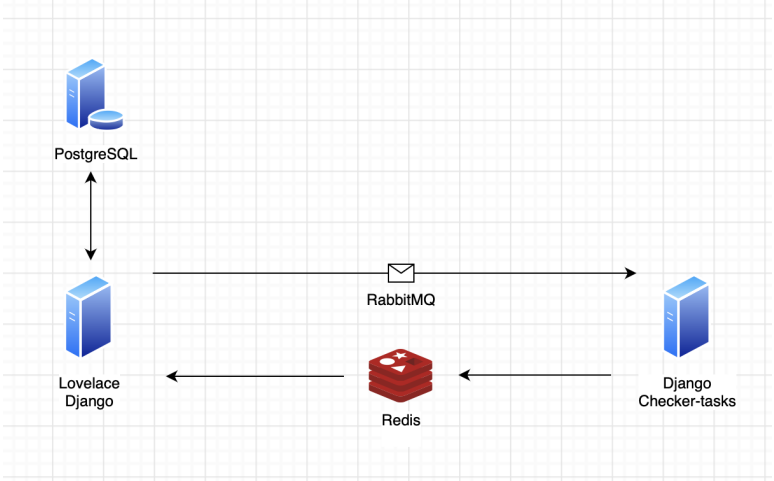

Figure 5. Visualization of third priority implementation; NFS is eliminated and the needed files are provided by the main server.

3.2. System Design

Our design goal for the decoupling of the main server and the checker server is to further isolate the checker server to be more independent and less dependant of the other changes to the system. The checker currently is a kind of microservice, so we should further refine this design and make it capable of sending and receiving information about the evaluation process with less dependencies to other system modules. With this kind of implementation there is no need for the checker server to write the result of the evaluation to the database, but rather to send it back to the main server. This communication will be handled with API. The design of the API

needs to be appropriate, in order to not limit either the main server or the checker server too much, maintaining the wanted independence between the servers. This kind of API design and communication between the two servers will also be able to include the other variations planned for the implementations, which will be covered in the Implementation Design section. Such as being able to handle the communication needed for the checker server to get the read information from the database as the main server being the information serving middleman.

A good solution would include communication between servers which does not include alternating the inner workings of a server, rather just the protocol used for communication between servers. An example of this may be a change to evaluation, where a partial evaluation score is returned if the evaluated exercise is not fully correct. This change would require modifications inside the checker service, but only a small change to the protocol used to carry information about the evaluated exercise. The main server would be able to choose if to use the partial score, or just process the complete exercise either as a fail or success. A nicely designed protocol could then provide freedom for both parties involved, with shared information not being too binding for either one.

By having a central implementation for writing to a database we also make the code base more modifiable[19]. Since writing to a database requires the client to know the structure of the tables and relations inside the database, a change to this logic would then mean a change to several places. If we have a centralized task that operates the interaction with the database, we will not have to deal with several modifications to multiple places. This again decreases the coupling of the system architectures parts.

3.3. Implementation Design

Our design for implementing the decoupling will be to replace the direct writing to the main database of the checker service task with a JSON object either stored to Redis transaction or inside RabbitMQ payload between the main server and checker service. The transaction contains all of the necessary information for the main server to be able to handle the database interaction. This kind of solution will comply with the first iteration of the initial design choices, with only the write access being revoked and no other functionalities being modified. The implementation is also easy to expand to cover the needed information in order to evaluate an exercise directly from the main server without the need of the checker service having to first read the database. This section will cover all of the iterations of the different design choices and modifications needed between these implementations.

At the moment checker service can write straight to the database and it uses a Celery task to save course exercise results to the database. Task creates an evaluation object that is then linked to the user's answer object and saved to the database. The evaluation object has the following fields visualized in JSON and this format could be used when sending the database write request to the main server:

```
{
  "answer":
    {
```

```
    "exercise_id":      id ,
    "user":             user ,
    "answerer_ip":      ip ,
    "instance":         instance ,
    "revision":         revision
    "evaluation":
    {
     "correct":              boolean ,
     "points":               integer ,
     "evaluation_date":      datetime ,
     "evaluator":            User ,
     "feedback":             text ,
     "test_results":         text/json
    }
   }
}
```

These fields determine is the answer correct, how many points answer has, when the evaluation occurred, user of the answer, handwritten feedback from the teaching staff and the checking logs of testing. This object is then added to the answer object and then saved into the database. The simplest and most straightforward solution would be to modify the current task to send the objects to the main server via Redis and save the data to the database from there. Figure 6 displays the data flow in this implementation.

In the second iteration of the design, where the Django checker server database access considering both read and write rights are revoked, the main server sends the needed information for the checker tasks to be able to complete. This includes information such as list of tests and their phases and run commands for test's different stages. These need to be read from the main server, which now acts as a primary database inter actor, since the other client is now revoked. Information will be send in a similar manner as in first iteration of the implementation, which is JSON object containing this necessary data.

Since the PostgreSQL database is hosted at the same server as the main Django server, the NFS will still remain to share the needed files between Django main server and checker service. This functionality is altered in the third iteration of the implementation. The checker server can now search the needed files from the main server's hosted NFS share, and complete the exercise audits as before. The main difference compared to the original database access being the information is now already provided with the request as the checker server is ordered to perform an evaluation of a certain exercise. This way the checker server does not need to even know of the existence of the database, as it obtains all of the necessary information from the main server. This implementation iteration is a major step towards decoupling of the Lovelace architecture, since now the checker server will not stop functioning due to database modifications, since there is only interaction between the main server and the checker service.

In the figure 6 the isolation of the checker server is visualized. In this implementation, the checker service will not interact with the database, but only consumes the information from the Django server and accesses the NFS, shared by

the main server process. Checker sends writable data back to the main server where it is written to the database. The figure 7 visualizes the checker server that has been stripped from both of the read and write access to the database. The main server handles all of the interaction to the database in the second iteration.

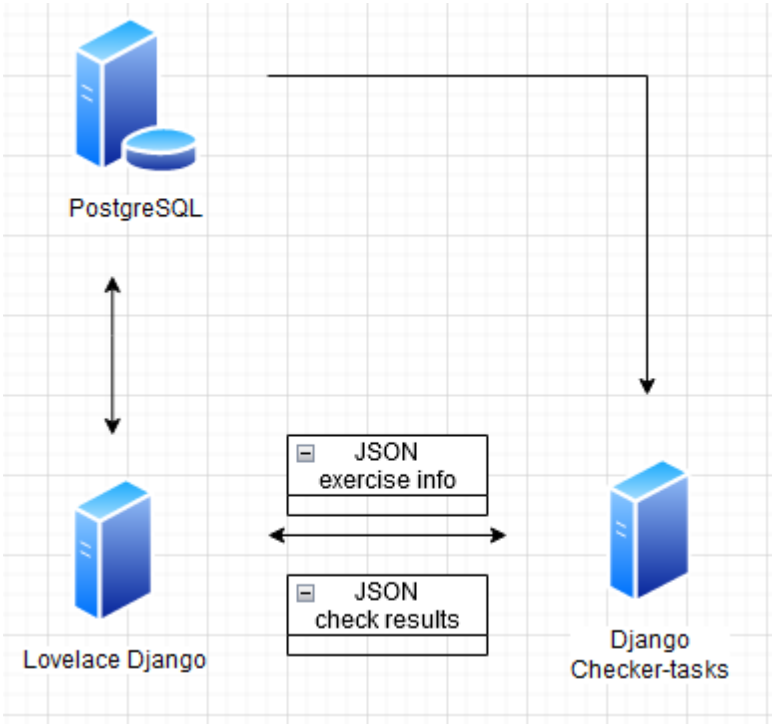

Figure 6. Visualization of the data flow in implementation, where the checker service does not write the database object to the database. The NFS is not portrayed in the chart.

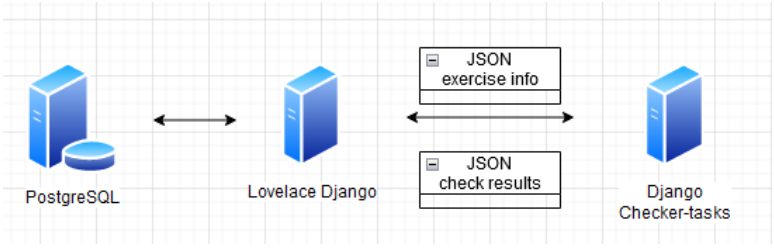

Figure 7. Flowchart of the second iteration of the implementation design. The NFS is not portrayed in the chart.

3.4. Design Analysis

The design of our implementation aims to reduce the coupling of the different parts of the Lovelace platform. By incorporating the API communication between the main server and checker service, we enable greater independence between these parts of the system, since their communication happens by HTTP requests in our design implementation. Our proposed JSON object, described in the Implementation Design section, which is used for transportation of information, only requires both parties

involved in the communication to be able to process JSON information and to send message back to the original request provider using the predefined format. Essentially this means that the inner requirements of the services are not so tightly regulated, allowing even different kind of programming language to perform the checker server process. This follows the idea of decoupled system, where parts of the system may be changed without significant effect on other parts.

Of course this implementation still contains limitations, as with any other design choice. As the main server handles the responsibility of writing, and possibly reading depending on the implementation choice of our designs, the monolith design grows on the main server. Considering the current size of the Lovelace platform this is not a major problem, but if it was to grow larger, a dedicated service to handle communication with the database should be considered. Having too much functionality in one service makes it difficult to maintain and change.

Benefits of sending a JSON object is the freedom to use only necessary data from it. If the JSON object contains extra data fields, they are not mandatory to be read or used in any way, but this allows more ways to process the data in different scenarios and alternating the saved information with low effort. Like the visualization JSON object displayed in the Implementation Design section for example, an insertion of a new data field would not affect the existing processing of current data. But the extra field of information could be handled as well if needed.

The benefits of either sending the JSON in a Redis transaction with an id or directly inside a RabbitMQ payload will be investigated further in the implementation section. This way we can test both solutions and have more insight of the benefits and drawbacks of the two aforementioned ways when applied to the data transfer. The Redis transaction allows us to store the JSON object with a transaction id, which the checker server will receive, and can fetch the data with this id. Inside RabbitMQ payload the JSON object is placed to a task queue where the checker service can consume it.

4. IMPLEMENTATION

4.1. Development Platform

Our solution to manage the cumbersome installation was to implement an automated process that carries out the needed configurations and installations of the system. We thought that the tedious process should be done only once. This automated deployment is done with the use of Vagrant and Ansible. With Vagrant we are able to create virtual machine environments where we can run the installation instructions automatically with Ansible. Ansible is an automation platform where is possible to produce automatic deployments with given configuration files and playbooks. In our implementation we used Ansible to run playbooks where we have our Lovelace main server and checker server installation tasks. Doing this the development platform can now be installed with a single vagrant command. The only part that is left to be done manually is the addition of data to the server so we can do testing with it.

4.1.1. Vagrant

Vagrant is a virtual development environment orchestration tool, which allows us to create and use virtual environments with low initial configurations needed. Vagrant uses virtual machine providers such as VirtualBox, which we are using, to create the virtual environment. To do this automatically Vagrant needs an configuration file which is called in this context as a Vagrantfile. In Vagrantfile we are defining our needed configurations for our virtual machines such as networking, operating system and hardware performance. [20] As an example here you can see our Vagrantfile configuration for our development platform:

Listing 4.1. Vagrant example

```
Vagrant.configure("2") do |config|

  config.vm.define "Lovelace" do |server|
    server.vm.box = "ubuntu/focal64"

    server.vm.hostname = "Lovelace"

    server.vm.network "public_network", ip: "192.168.1.50"

    server.vm.provider "virtualbox" do |vb|
      vb.memory = "2048"
      vb.cpus = "2"
    end

  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "ansible/lovelace_main.yml"
    end
end
```

```
config.vm.define "Auxchecker" do |aux|
  aux.vm.box = "ubuntu/focal64"
  aux.vm.hostname = "Auxchecker"

  aux.vm.network "public_network", ip: "192.168.1.51"

  aux.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
    vb.cpus = "2"
  end

  aux.vm.provision "ansible" do |ansible|
    ansible.playbook = "ansible/checker_server_main.yml"
  end
end
end
```

This configuration creates two virtual machines, which behave as both of the machines were an actual computer. This kind of reproducible and life-like setup is in many ways beneficial for our developing needs, since with this kind of automatic environment, we can start from scratch easily, and be back testing with freshly created environment within matter of minutes. Our final version of development platform runs on single virtual machine so the installation process is simpler and developing takes less effort.

4.1.2. Ansible

Ansible is the deployment tool that can be configured to automatically run given tasks. Given tasks are written in a playbook that Ansible uses to carry out deployments. Playbooks are files written in YAML ain't a markup language (YAML) which is a very user-friendly and simple. In our situation Ansible connects to the virtual environments via ssh we created with Vagrant and performs the created playbooks each on its own environment. [21] The next snippet of YAML shows an example how playbooks are written. In the snippet we are updating repositories so we can install PostGreSQL server and then it is installed with its dependencies:

Listing 4.2. Ansible example

```
- name: Start Lovelace server and install packages
  hosts: Lovelace
  become: yes
  become_user: root
  vars_files:
    - group_vars/all

  tasks:

    - name: Update repositories
      apt:
```

```
    update_cache: yes

- name: Install PostGreSQL Server
  apt:
    name:
      - postgresql-12
      - postgresql-contrib
      - python3-pip
      - python3-dev
      - virtualenv
      - python3.8-venv
      - postgresql-client-12
      - apache2
      - apache2-dev
      - nfs-kernel-server
      - rpcbind
```

Automation with Ansible is both easy and powerful, compared to the traditional alternative, shell scripting. Ansible automation playbooks are both shorter and less error prone, than shell scripts, at least with no extensive error handling being done to the shell script. Using Ansible, we can leave most of the handling to the tool, since Ansible is idempotent, which means that unless something has changed, there is no need to re run the command.

4.1.3. Lovelace Git Repository

We have created a fork of the original Lovelace repository, where we can develop our implementation of the Lovelace decoupling and control the versions of the different stages of the process. This fork allows us to pull this repository branch with the made changes and set it up automatically to reflect the modifications in a live environment. With different development branches for each of the stages of the decoupling we can compare the effects of each implementations and compare the differences. For the project management, using git is highly beneficial, since all the changes are made and contained within their respective branches. This version control enables us to inspect the differences more effortlessly compared to using only different files for example, and switching between them.

4.2. Revoking Write Access

Our first priority was to revoke the checker service's write access to the database of the main server. This was accomplished by reformatting the existing code in the repository. Modified python files were tasks and views. In tasks file we modified shared Celery tasks called run_tests. In this task originally, the exercise results are saved in the database by worker. Also the rendered results are saved into Redis. Because of that we decided also to use Redis in this instance to store the needed data that the main server needs for saving the evaluations. The needed data consisted of object ids, result strings

and exercise points. Main server needed ids for the user, answer, and exercise. In the task we removed the actual saving and instead of that we stored the data to Redis. Now when the main server orders the given celery task the data will be stored and then fetched by the main server. We added the saving function into the views file in the file_exercise_evaluation function. With this addition and the data from Redis, main server is now able to save the evaluation to the database without the worker. It can query the right objects regarding the evaluation with the ids that are stored in Redis in the task by the worker.

With this modification to the Lovelace, we have achieved the first objective of the initial design goal, which was to removal of checker servers need to write the results to the database. By revoking the writing access, the checker service needs significantly less rights to the database, since after the modification, it only needs to read from the database. This requires less trust, and is therefore already a significantly better alternative, compared to the writing happening as well. However, with this option the problem of needing to sync the changes to the database between the main server and checker service persists. Since the checker still needs to read the database, the changes need to be made for both of the services, maintaining the high level of development overhead within the system. This is why the second iteration of the project plan is needed, to decrease the coupling between these two services.

4.3. Revoking Read Access

The second stage priority of the project consists of removing the need for the checker server to read the files and all the necessary data from the database. This is substituted by providing the data ready from the main server, containing the needed files and other information for successful evaluation process within the checker server's tasks. As the evaluation process advances in the original design, when receiving a number of id's from the main server, the checker can fetch the needed information along the execution of functions, however this is not possible if the read access is revoked. Essentially this means, that all of the required information must be handed to the checker server from the main server's side when initiating the evaluation service.

As stated in the Design Analysis section, we decided to implement a JSON object, in which we gather the needed information before sending it to the checker server. During the testing of the information gathering, the benefit of using JSON as the base of information was distinct. It allowed for easy gathering and sending of several kinds of experimental test data, which had no effect on any other part of information, since they are separated in python dictionary object. Transferring of the data was done with modifying models.py file, where we added the JSON object as parameter for checker service, containing processed information of files and needed commands, and fetched this object from the checker server in the run_tests function as a normal parameter. The possibility of using Redis as storage for the JSON object, as mentioned before in the Design section, was tested, but it was discarded as being impractical, since we had to modify the database model anyway to accept an extra argument. This meant that if we decided to store a Redis transaction id, only an extra unneeded software layer would be introduced, since adding the JSON directly as an argument is a valid solution.

Additionally, adding the JSON parameter does not need generation of transaction id's and we can be sure the data is always transferred correctly, since the data is part of the starting parameters of the checker servers evaluation process.

Due to several kinds of database queries needed throughout the evaluation process of the checker service, this meant that the data object JSON was going to be quite large in size. At first the nested nature of the queries during the evaluation process presented a challenge on how to organize the data within the created JSON. However, it was pleasing to quickly notice, that the structure of the JSON would not need to be highly nested, since python dictionary objects are easy to divide and reference later on in the functions. When all of the necessary data was confirmed to be correctly received by the checker service, the design of the modified evaluation process could be started. Here is an example of the JSON, which is passed to the checker server, containing the information of the evaluation:

Listing 4.3. Json data for the checker

```
{
    "tests": [
        "test1",
        "test2",
        "test3"
    ],
    "commands": [
        "command1",
        "command2",
        "command3"
    ],
    "files_to_ch": [
        "file1",
        "file2",
        "file3"
    ],
    "exercise_list": [
        "exfile1, exfile2, exfile3"
    ],
    "instances_list": [
        "instance1",
        "instance2",
        "instance3"
    ],
    "instance_file_links": [
        "instance_file_link1",
        "instance_file_link2",
        "instance_file_link3"
    ],
    "files_to_check_correct": [
        "file1",
        "file2",
        "file3"
    ]
```

}

We studied the structure of the nested function calls when initiating the evaluation process and discussed the flow of operations with our project supervisor. A decision was made to reduce the number of functions, by skipping some unnecessary stages and operations for the evaluation process, which both simplified the development of the new checking server work flow, and reduced the amount of needed data, since some database queries were discarded in the process. The changes made do not have impact on the evaluation itself, but rather help to simplify the ongoing operations, by striping the excess stages around the evaluation. After reducing the needed steps and having the correct data on hand from the main server, it now was more straightforward to combine the needed steps from the different functions to form a checking process to match the ready given data. After completing the evaluation process building, we got an correct answer displayed in the checker servers worker output. Unfortunately, the final step of evaluation process is generating an feedback object, consisting of the data from the evaluation. Since our evaluation process skips some of the stages, which the last generator is expecting, the celery worker does not respond with an evaluation passed response back to the main server. However, for the purpose of testing and implementation, we could check that the evaluation process was successful by investigating the terminal output of the worker, which displays the result of the tests run. After investigation of the result-logs, we could confirm the run tests had been successful, and the evaluation had been completed without needing to read information from the database. This concludes the stage 2 implementation, since we had revoked the need of reading from the database to perform an evaluation.

4.4. NFS Implementation

Because of the modifications done in the read access revoke the use of NFS is now retired. The elimination was done by our implementation of the read access revoke. NFS allowed the use of simple sharing of files for exercise evaluation. Files are not needed to be shared via NFS anymore because the files are read on the main server when the needed data for the evaluation process is created for the checker service. We transferred the same function used to read the exercise files contents from the checker server back to the main server, where this function may read the data straight from the disk, instead of relying on NFS sharing the folder over network. The contents of files are Base64 encoded in the evaluation JSON, to ensure they get to the destination of checker server intact.

5. EVALUATION

5.1. Evaluation Plan

Creating the development platform for Lovelace had a significant and time consuming role in our project so small comparison of different platform setups is one subject of our evaluation plan. Our original development platform consisted of multiple virtual machines where each part of the system operated, but we decided to change to only one machine where everything operated. Building and rebuilding the system when needed always took a lot of time so we spared a significant amount of time when we moved the automation to create a single virtual machine.

Considering the main goal which was the actual modification of the code repository we can compare how each iteration of the implementation goal improves the status of coupling in the system. We can also consider the security aspects of the iterations. Both write and read access and their revoking has an reasonable effect on the state of coupling in the system. We can also try to measure modifications' effect on the processing performance of the system if we can create an eligible situation to measure it. With results we can asses that are done modifications justifiable security wise if there is a minor performance drop found.

5.2. Evaluation Execution

After discussing on possible evaluation implementations with the supervisor we came to the conclusion that we should continue with a more heuristic approach on evaluation. Since the lack of proper environment for comprehensive performance testing we cant perform testing that would measure properly real-life-like traffic on the system. While we are not measuring performance we can still discuss and observe the potential performance issue along with the implementation iterations and the changes that are brought with them.

5.3. Evaluating Iterations

5.3.1. Write Access

Revoking the write access from checker service to the main server decreases the amount of processing in the checker service though it increases on the main server side. Though the increased load is quite insignificant our implementation still centralizes checker services' processes to the main server as checker service does not perform write operations to the database but stores them to Redis for the main server to write them to the database. So the solution lightens the processing load on the checker service and transfers it to the main server. State of the coupling has been reduced with this iteration, since the write access is removed. For checker server this means it must not know the exact format of the database models and schemas anymore, since the remaining operations focus on querying the database. These operations can be done

Table 1. Lovelace original I/O flow

I/O operation	Service	Location
Start evaluation with exercise ids	local disk, Postgres	Main server
Redis task start	Celery, RabbitMQ	Main server
Read needed files from database	Postgres	Checker server
Get files from NFS	NFS	Checker server
Perform evaluation	local disk	Checker server
Save evaluation to database	Redis	Checker server
Save rendered results to Redis	Redis	Checker server
Fetch results from Redis	Redis	Main server

Table 2. Write access revoked

I/O operation	Service	Location
Start evaluation with exercise ids	Django	Main server
Redis task start	Celery, RabbitMQ	Main server
Read needed files from database	Postgres	Checker server
Get files from NFS	NFS	Checker server
Perform evaluation	Redis	Checker server
Save evaluation object to Redis	Redis	Checker server
Fetch results from Redis	Redis	Main server
Save evaluation to database	Postgres	Main server

Table 3. Read and Write access revoked

I/O operation	Service	Location
Start evaluation with exercise ids	Django	Main server
Read needed files from database	Postgres	Main server
Form JSON containing needed data	Django	Main server
Pass exercise JSON to Redis	Redis	Main server
Redis task start	Celery, RabbitMQ	Main server
Perform evaluation	Redis	Checker server
Save evaluation object to Redis	Redis	Checker server
Fetch results from Redis	Redis	Main server
Save evaluation to database	Postgres	Main server

with provided id's from the main server, so overall the state of coupling regarding the database access has decreased. Still this option leaves the maintaining overhead relatively high, since if some new table is added to the database, the changes has to be updated to the checker server as well, regardless of only polling the database. This problem is addressed in the read access revocation iteration in the next section.

Security-wise revoking the write access to the main server database is well justified. No other than the main server should have a right to write to the database so isolating external entities from write rights such as the checker service should be implemented. Due to this isolation now taking place we can also keep executable code not being able to have access to the database so no malice could take place.

5.3.2. Read Access

Now when the tasks on the checker service are not reading the database, more load is transferred to the main server where the needed data for evaluation is read. This change is not really significant because we are merely changing the place of the processing. This is because of the operations performed on the main server, these including the added database interactions, are not heavy on the system CPU utilization. The allocated time for processing comes mainly from the system waiting for Input/Output (I/O) data transfer, rather than demanding heavy CPU loads. Due to this I/O utilization changing place from the checker server, to the main server, we do not significantly increase the CPU load of the server, but rather have impact on the unloading speed of the user submitted exercises. This queue has also been present in the previous iterations, but now it is more centralized due to the fact the unloading takes place in the main server. This centralization can bring up the question of the need to add a dedicated database wrapper, to bring down the time for unload the database read operations queue. Centralizing processes to a single component in a system fundamentally adds stress to the given component and brings out the thought of potential bottleneck. This kind of addition would also comply with the introduced concept of micro-services, which the main server does not comply anymore with the added functionalities.

Now when the checker service is isolated from the database, the system could be considered more secure because we have decreased the number of actors interacting with the main database and the executable code that runs in evaluation processes in the checker service which is now isolated from the main database. Now the main server is also isolating the checker servers needs to read or write to the database. For the systems state of coupling, the change is significant. Now the database changes need to be synchronized only to the main server, which then sends the evaluation JSON to the checker, when initiating the evaluation process.

5.3.3. Eliminating NFS

The main objective of NFS was to offer the needed files for evaluation process for the checker service. Elimination of this file system adds more I/O load to the main server just as we discussed the fact on the earlier section. Still the complete I/O load may be

considered to be less than with the NFS system enabled, since the file read operations do not occur over a shared network, but within the same component, so there is no stress and traffic within the network. When the NFS service is not needed anymore, the checker server does not need to be in the local network anymore, therefore the location of checker servers could be transferred anywhere, accessible through HTTP requests.

5.4. Performance Evaluation

Performance testing for the Lovelace system iterations can be estimated, even though we do not perform actual benchmarks with generated artificial traffic. This is because of the main load and time from the processes comes from the I/O load and waiting time between requests. This enables us to review the time for a complete evaluation cycle of user submitted exercises and estimate the affects of altered stages. In the original Lovelace system, the checker server handles the read and write operations to the database. Unlike in the read and write revoked iterations, where these operations are spread before and after the Redis evaluation task. Since the database operations are not CPU intense, they can therefore be executed rapidly in succession. After evaluation is complete, the main server receives a complete status for the current task by polling the tasks state and returns the evaluation to the user in browser. This polling is done in 100 milliseconds (ms) cycles, plus the additional time of processing, so the effective time for complete cycle is 100 ms + processing time. What this polling cycle does, is that it creates waiting time when the main server is awaiting the complete status of the Redis task. Since in the original Lovelace, the database interactions are done from the checker server, they can be done quickly sequentially, and after receiving the complete status, the processing is complete and can be displayed to the user faster. This is because of the result is already stored in the database, and processing is not needed after receiving the complete status from the checker.

With the added changes to the I/O operations taking place in the main server, the time for waiting the result of the Redis task becomes a little more nondeterministic. This is because of the waiting time of 100 ms between polls, which can create gaps to the evaluation process. For example if the checker server would complete the otherwise quicker operations, without needing to write the results in the database, in 101 ms, the complete time for the process would still be at least 200 ms, due to polling intervals of the main server. And because of the reduced operations of checker server's revoked read access, the time would be increased even further by needing to save the evaluation in the main server. As a result the overall time may in some cases increase compared to the original. While the increase in time is not critical, this could be considered by reducing poll time from 100 ms to a smaller window. This nondeterministic behavior depends on the timing of the operations, the speed of the network and the performed testing of the exercise files.

As can be seen from comparing the Table 1 with Table 2 and Table 3 the amount of processing keeps on increasing within the main server, with each iteration. These moved steps used to be part of polling cycle, which the main server creates, however with reduced steps within the checker server, more time is allocated to just waiting the

result of evaluation with current polling, since the database operations are not part of the poll cycle anymore.

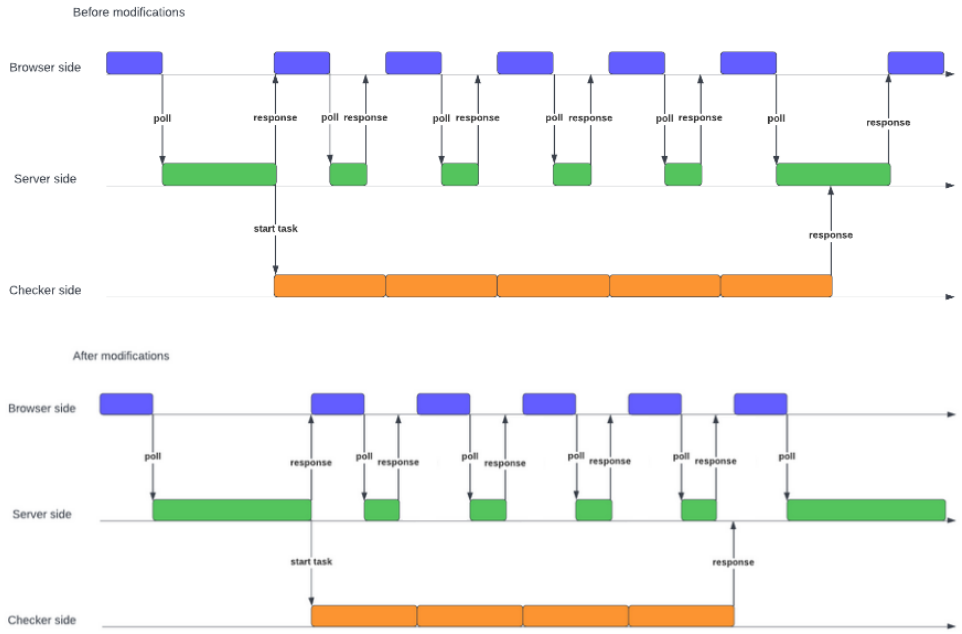

Figure 8. Visualization to represent the changes in operation distribution.

As the figure 8 shows, before modifications more operations were done in the checker side, after modification operations are moved to the server side. The Checker line in the lower graph is shorter, due to transferred operations into the main server side. The total amount of operations remains the same. We are merely switching the place of operations in the different iterations of the system.

5.5. Development Platform

Having two different versions of the development platform, we ought to observe both and discuss on the reasons why a certain version was decided to be used. For effective and fast development we needed a platform that was easy and fast to setup. Both versions of the platform were very easy to setup. The system that used multiple VMs for the system resembled more the real Lovelace system but the setup took much more time. This was because of two virtual machines needed to be built and configured each time a new iteration was tested, leading to regular long build times. The second version installed everything on the same VM and therefore reduced the setup time drastically. The latter one was chosen to be used as the main platform though its structure was different to Lovelace. It still had the same functionality in the modifications we made in the implementation. Setup and installing the development platform took so much time and restarting the platform so it was pretty clear to switch to the mono VM version to save time and effort. Every Lovelace development iteration was tested on both platforms to ensure functionality, especially since the first testing was on the single VM environment, so it was important to test the system functionalities on the more life like

system as well, where the similar connections had to be made between components, as in the real Lovelace system.

6. DISCUSSION

When reviewing the project's and the initial goals, all of the requirements are met with our current implementation. Design and planning of the different implementations were successful and well contained, therefore we could follow the plans without hardly any changes to the initial design. We are pleased to have met all of the set requirements at the start of the project. This means that we can consider the made changes successful. One of the key aspects for this project was the development platform, which allowed us to effortlessly set up environments for different development stages and to test different variations of solutions to find the best one for respective tasks. The development of this platform took quite some time, but it was well worth the time, because of the convenience it brought for our development process. The development of the automation was not always straightforward, due to need of adapt and change software versions and behaviour of some configurations compared to the setup and installation guide provided with the Lovelace system. We also needed to study the used tools, Ansible and Vagrant a lot, to learn how to convert some of the manual and shell script steps to comply with our setup. The development platform had some changes made for it during the project, this was to speed up the creation of new development instances. Because of the increased speed of the instance creation with the single container virtual machine, we were able to speed up the development process, and as a side product it was useful also because of lower threshold to spin up a new environment due to decreased build times. This in turn, allowed for testing of even smaller changes, with completely new environments, just to make sure the implementation works as intended.

Our project first priority, revoking the write access of the checker server, was completed first. When inspecting this implementation, the requirement of the checker not writing the result of the evaluation to the database, is easy to confirm. Since this iteration of the project was quite simple, not requiring too many changes, but rather just sending of the JSON object back to the main server, we are pleased with the result. Our first tested solutions were more complicated, but for this kind of solution we wanted to find a simple answer. The change could be made with minimal changes to the original source. This is good, since even with just implementing this step, the state of the coupling can be reduced within the entire Lovelace system. This is turn reduces the development overhead of the system administrators.

Regarding the next iterations, revoking the read access and removal of NFS from the checker server, were handled at the same time. This was not planned initially, when designing the solutions and priorities. It was discovered when we took closer look at the state of the NFS service after verifying the read access revocation. This change, containing both of the remaining steps of the project, meant that the state of the checker server was quite detached from the main server. The checker now only needs to be accessible trough HTTP requests, the deployment could be done even to a remote server, since the NFS is not used for file transfer. This is good news for system flexibility and could be combined with cloud services for example, to provide the checking server as a service.

The current version of Lovelace does not make use of micro-services. Micro-service based system consists of small and autonomous little parts, handling small tasks by

themselves. This kind of approach to Lovelace would make it easier to build additional services, since addition of new software is easier to micro-services based platform, rather than current monolithic code base[22]. Usage of micro-services would continue the trend of slicing the functionalities of a service to smaller, more contained one, like we have done in our project by stripping the checker server of its read and write access from the database. Next step could be to integrate a dedicated database inter actor, to handle all the database interactions for clients. This would reduce the system coupling by having only one component tied to the database schemas and exact format of interaction with it.

In the future there are some adaptations which could be beneficial for the system. One of these changes is migration to Representational state transfer (REST) based system, where the services could offer HTTP endpoints to communicate with each of the other parts of the system. Combined with the microservice design, where every part of the system is dedicated to one small task, to decrease the coupling within the system. Communication over REST would allow to choose the stack of technologies more freely, since the communication happens over HTTP, which almost all of the modern programming languages and platforms support. This approach could be combined partly with the existing solutions, for example the usage of JSON for sending the data to checker server and returning the evaluation back to the main server. JSON is often used within REST, because of its universal nature over modern programming languages[23].

Following the microservice implementation, would be to implement a cloud based checker server as a service type of integration. Moving the checker server to cloud could mean improved scalability and easier updates for the system.

7. CONCLUSIONS

The topic of the thesis and the subject of the project was to reduce the state of coupling between the Lovelace's main server and the checker server. This was done due to the need of sync changes between both of the servers, which equals to a lot of development overhead. The project offers a solution for this problem with decoupling the checker server from the database and sending and receiving all of the data from the main server, which previously was got and written to the system database. This reduces the amount of system components interacting with the database, and therefore reduces the need for syncing the database changes for two servers. During the process our team learned a lot from automating a system, from creating the development platform. As well as Django and Redis, since these were the main components we had to learn how to use to make the proposed changes happen. The project was really educational overall. The initial studying of the Lovelace system was hard for us, but with time and resilience we got trough and finished with result we are happy with. The system could be further improved with some changes proposed earlier, but our thesis project goals were achieved.

8. REFERENCES

[1] Gagliardi V. (2021) Decoupled django with the django rest framework. In: Decoupled Django, Springer, chap. 1.

[2] De Win B., Piessens F., Joosen W. & Verhanneman T. (2002) On the importance of the separation-of-concerns principle in secure software engineering. In: Workshop on the Application of Engineering Principles to System Security Design, Citeseer, p. 1.

[3] Lovelace ohje opettajille. URL: https://lovelace.oulu.fi/lovelace-ohje-opettajille/lovelace-ohje-opettajille/. Accessed: 2022-01-27.

[4] Django framework. URL: https://www.djangoproject.com/. Accessed: 2022-01-27.

[5] Security features of django. URL: https://docs.djangoproject.com/en/4.0/topics/security/. Accessed: 2022-01-27.

[6] Protocol and connection spesification of redis. URL: https://redis.io/topics/protocol/. Accessed: 2022-01-31.

[7] Rabbitmq is the most widely deployed open source message broker. URL: https://insights.stackoverflow.com/survey/2021#section-most-loved-dreaded-and-wanted-databases. Accessed: 2022-01-31.

[8] Most wanted databases listing of stack overflow. URL: https://www.rabbitmq.com/#features. Accessed: 2022-01-31.

[9] Which protocols does rabbitmq support? URL: https://www.rabbitmq.com/protocols.html. Accessed: 2022-01-31.

[10] Amqp 0-9-1 model explained. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html. Accessed: 2022-01-31.

[11] Celery - tasks. URL: https://docs.celeryproject.org/en/v5.2.3/userguide/tasks.html. Accessed: 2022-02-16.

[12] First steps with celery. URL: https://docs.celeryproject.org/en/stable/getting-started/first-steps-with-celery.html#choosing-a-broker. Accessed: 2022-01-31.

[13] Celery - distributed task queue. URL: https://docs.celeryproject.org/en/master/index.html. Accessed: 2022-01-31.

[14] Celery - django. URL: https://docs.celeryproject.org/en/stable/django/first-steps-with-django.html. Accessed: 2022-02-17.

[15] Postgresql sql conformance. URL: https://www.postgresql.org/docs/current/features.html. Accessed: 2022-01-31.

[16] Postgresql features. URL: https://www.postgresql.org/about/featurematrix/. Accessed: 2022-01-31.

[17] Nfs overview. URL: https://www.ibm.com/docs/en/aix/7.1?topic=management-network-file-system. Accessed: 2022-02-02.

[18] Nfs services. URL: https://www.ibm.com/docs/en/aix/7.1?topic=system-nfs-services. Accessed: 2022-02-02.

[19] Newman S. (2021) Building microservices, " O'Reilly Media, Inc.". pp. 41–42.

[20] Vagrant intro. URL: https://www.vagrantup.com/intro. Accessed: 2022-03-15.

[21] Ansible howto. URL: https://www.ansible.com/overview/how-ansible-works?hsLang=en-us. Accessed: 2022-03-15.

[22] Newman S. (2021) Building microservices, " O'Reilly Media, Inc.". pp. 6–7.

[23] What is rest, redhat. URL: https://www.redhat.com/en/topics/api/what-is-a-rest-api. Accessed: 2022-04-28.

9. APPENDICES

Appendix 1

Work distribution

Mikko Kaasila	Stage 1	35h writing, research on system components, initial design of implementation
Mikael Pennanen	Stage 1	45h writing, discussion with the supervisor, study components
Mikko Kaasila	Stage 2	70h writing, studying code repository, discuss development platform design, code implementing and debugging
Mikael Pennanen	Stage 2	96h writing, discuss development platform design, development platform installation and debugging, code implementing and debugging
Mikko Kaasila	Stage 3	25h writing evaluations, evaluation design, visualizations
Mikael Pennanen	Stage 3	40h writing evaluations, evaluation design, discussion with the supervisor,
Mikko Kaasila	Stage 4	20h writing discussion, corrections and fixes to text
Mikael Pennanen	Stage 4	20h writing discussion/conclusion, discussion with the supervisor
Mikko Kaasila	Total	150h
Mikael Pennanen	Total	201h

Appendix 2

Listing 9.1. Vagrant example

```
Vagrant.configure("2") do |config|

  config.vm.define "Lovelace" do |server|
    server.vm.box = "ubuntu/focal64"

    server.vm.hostname = "Lovelace"

    server.vm.network "public_network", ip: "192.168.1.50"

    server.vm.provider "virtualbox" do |vb|
      vb.memory = "2048"
      vb.cpus = "2"
    end

  config.vm.provision "ansible" do |ansible|
    ansible.playbook = "ansible/lovelace_main.yml"
    end
end

config.vm.define "Auxchecker" do |aux|
  aux.vm.box = "ubuntu/focal64"
  aux.vm.hostname = "Auxchecker"

  aux.vm.network "public_network", ip: "192.168.1.51"

  aux.vm.provider "virtualbox" do |vb|
    vb.memory = "2048"
    vb.cpus = "2"
  end

  aux.vm.provision "ansible" do |ansible|
    ansible.playbook = "ansible/checker_server_main.yml"
  end
end
end
```

Appendix 3

Listing 9.2. Ansible example

```
- name: Start Lovelace server and install packages
  hosts: Lovelace
  become: yes
  become_user: root
  vars_files:
    - group_vars/all

  tasks:

    - name: Update repositories
      apt:
        update_cache: yes

    - name: Install PostGreSQL Server
      apt:
        name:
          - postgresql-12
          - postgresql-contrib
          - python3-pip
          - python3-dev
          - virtualenv
          - python3.8-venv
          - postgresql-client-12
          - apache2
          - apache2-dev
          - nfs-kernel-server
          - rpcbind
```

Appendix 4

Listing 9.3. Json data for the checker

```
{
    "tests": [
        "test1",
        "test2",
        "test3"
    ],
    "commands": [
        "command1",
        "command2",
        "command3"
    ],
    "files_to_ch": [
        "file1",
        "file2",
        "file3"
    ],
    "exercise_list": [
        "exfile1, exfile2, exfile3"
    ],
    "instances_list": [
        "instance1",
        "instance2",
        "instance3"
    ],
    "instance_file_links": [
        "instance_file_link1",
        "instance_file_link2",
        "instance_file_link3"
    ],
    "files_to_check_correct": [
        "file1",
        "file2",
        "file3"
    ]
}
```