MSc in Computer Science

Department of Computer Science

MScThesis

“Almosino Skeleton:
A software engineering framework for question answering systems”

Iason Andriopoulos

February 2017
Acknowledgments

I would like to thank my supervisor Prof. Ion Androutsopoulos for providing me the chance to get involved in a very interesting Computer Science domain. I would also like to thank Prodromos Malakasiotis and Ioannis Pavlopoulos for their valuable guidance throughout the development of the thesis. Finally, I dedicate this thesis to my family and my other half Despoina for their patience and their support.
Table of Contents

1. Introduction
   1.1. Question Answering Systems
   1.2. Responses of Question Answering Systems
   1.3. Structure of Question Answering Systems
   1.4. Contribution
   1.5. Notation
   1.6. Outline

2. Software Engineering perspective
   2.1. Challenges
   2.2. Design targets
   2.3. Solution

3. Almosino Skeleton
   3.1. Functionality
   3.2. Structure
   3.3. Core class model
   3.4. System components

4. Almosino QA
   4.1. Target
   4.2. Baseline components
   4.3. System architecture
   4.4. API
   4.5. Experiments

5. Conclusions
   5.1. Challenges
   5.2. Design Targets
   5.3. Future work

6. Appendix

7. References
1. Introduction

1.1 Question Answering Systems

Question Answering is one of the many interesting and challenging applications of Natural Language Processing and Information Retrieval. In general the purpose of Question Answering systems is to provide easily understandable and precise answers to natural language questions submitted by human users. An example of the responses a Question Answering system aims to provide is the following:

**Question:** “Is fever a symptom of flu?”

**Answer:** “Yes. Fever is a symptom of flu. Other symptoms include headache, sore throat, coughing.”

As illustrated in this simple example, Question Answering systems do not return lists of possibly relevant documents, unlike traditional Information Retrieval, but (ideally) direct answers. This functionality can prove useful in a wide range of use cases, since searching in large document collections is a problem that concerns many scientific and industrial domains, as well as everyday life applications. Question Answering systems may also help users find information in structured data (e.g. databases, ontologies) rather than documents, but this thesis is primarily concerned with Question Answering systems for document collections.

1.2 Responses of Question Answering Systems

Following the terminology of the BioASQ challenge [1], we consider two types of answers, exact and ideal. Ideal answers are summaries of the information relevant to a query while exact answers aim to answer directly the user’s question. Exact answers depend on the type of the question while ideal answers can be generated for any type of question. Considering the 4 question types of BioASQ, we present examples of the ideal and exact answers that should be generated by a Question Answering system:

- **Yes/No:** A question that can be answered with a simple Yes or No.
  - Question: “Is fever a symptom of flu?”
  - Exact Answer: “Yes.”
  - Ideal Answer: “Yes, fever is a symptom of flu. Other symptoms involve headache, sore throat, cough.”
• **Factoid:** A question whose answer is a single entity.
  - Question: “Which science’s field is Machine Learning?”
  - Exact Answer: “Computer Science.”
  - Ideal Answer: “Machine Learning is a field of Computer Science that involves the study of algorithms that can learn from and generate predictions on data.”

• **List:** A question whose answer is a list of entities.
  - Question: “What are the symptoms of flu?”
  - Exact Answer: “Fever, headache, sore throat, cough.”
  - Ideal Answer: “Common symptoms of flu include headache, sore throat, fever, cough.”

• **Summary-only:** A question that has no exact answer (e.g., it cannot be answered by returning an entity name or a list of entity names) and can only be answered by an Ideal Answer (summary of relevant information).
  - Question: “What do you know about the flu?”
  - Exact answer: -
  - Ideal Answer: “Influenza, also known as the flu, is an infectious disease whose common symptoms are fever, headache, sore throat, and cough.”

### 1.3 Structure of Question Answering Systems

In general, Question Answering is a process that involves many different components and several stages of execution in order to generate the final results. As a first step, the query is processed (removing stop words, special characters, possibly adding synonyms etc.) then used in order to retrieve documents relevant to the query. Next, the most relevant to the query snippets (parts of text composed of a single or few sentences) are selected from the relevant documents. The snippets (or the documents, depending on the approach) are then used to extract exact and ideal answers. A representation of the process can be viewed in Figure 1.1
1.4 Contribution

The implementation of Question Answering systems is a complex problem since the final result is the outcome of a process that involves four (or maybe more) components that are related to Machine Learning, Information Retrieval and Natural Language Processing. One of the many challenges of Question Answering is the software engineering task to combine all these different components in a single, robust system. We address this issue by providing Almosino Skeleton, a software engineering framework that can be used as a backbone to develop Question Answering systems.

1.5 Notation

The rest of this thesis uses the following abbreviations.
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>QA</td>
<td>Question Answering</td>
</tr>
<tr>
<td>NLP</td>
<td>Natural Language Processing</td>
</tr>
<tr>
<td>IR</td>
<td>Information Retrieval</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
</tr>
<tr>
<td>SE</td>
<td>Snippet Extraction</td>
</tr>
<tr>
<td>EAE</td>
<td>Exact Answer Extraction</td>
</tr>
<tr>
<td>IAE</td>
<td>Ideal Answer Extraction</td>
</tr>
</tbody>
</table>

1.6 Outline

The remainder of the thesis is organized as follows:

**Chapter 2 - Software engineering perspective:** The software engineering challenges of the task we address and the design targets of our solution.

**Chapter 3 - Almosino Skeleton:** A description of the framework’s functionality, basic logic and structure.

**Chapter 4 - Almosino QA:** Description of an example Question Answering system developed using the framework and techniques used to test the system’s functionality.

**Conclusion:** Summary of the results of the thesis and future work.
2. Software engineering perspective

From a software engineering perspective, the challenge is to bind the set of NLP/IR components described so far into a single robust logic. That is, we need to provide software that easily connects one component to another generating a complete QA system. In this chapter we will examine the challenges of this task and we will introduce our approach and design targets.

2.1 Challenges

Programming language heterogeneity

Programming language heterogeneity in an ecosystem of projects (e.g., student projects each developing a QA component) is something we should expect since software engineering trends come and go. Even if we imposed a rule to develop every component in a specific language there is no evidence that this decision would not be rendered obsolete in a few years. For example, while our current QA components are developed in Java and Python, in the near future several components could be developed in R or Scala. Therefore our first challenge is to define a communication standard between heterogeneous components.

System cost and efficiency

ML/NLP/IR methods, including those used in QA systems, usually have demanding system requirements since they process large volumes of information. However their requirements differ a lot. For example, the speed at which data are retrieved from external storage (e.g., hard disks) is important for document retrieval but not so for snippet extraction; or the memory requirements of a snippet extraction component might be much more demanding than those of a summarizer if the former has to process a lot more information than the latter (e.g., documents vs. selected snippets). Even the same component may have highly variable requirements depending on the algorithms it uses (for example, a deep neural network would require a sufficient GPU, while for example a Logistic Regression classifier would not). A system combining heterogeneous components, unless designed carefully, would result in demanding lots of resources but rarely using them in a sufficient rate. Imagine, for example, a system that demands the hard drive specifications of an Information Retrieval engine and the GPU/CPU specifications of a neural network, but actually uses them 50% of the time or less. Efficient usage of the system’s resources is a critical problem since most applications nowadays aim to be deployed on the cloud and served as a service.

System agility

In a non dynamic ecosystem of components, it is easy for a software developer with knowledge of each component’s input and output to bind them in a single robust system. However, in research-oriented QA systems the selection of components is dynamic. For example, a new
component that re-ranks the retrieved documents could be added, or a new summarizer could replace an older one. Such changes, unless we design the system carefully, may require a lot of re-engineering.

**Robustness**

We cannot afford unexpected behaviour to cause error propagation throughout the system, even if unexpected behaviour comes up at some point of the execution. For example, let us consider the scenario where we use a set of remote services, some of which become unavailable temporarily due to a server failure, or loss of internet connection. Under no circumstances should this failure cause additional failures to other components or even worse propagate an error to the user. Instead, the failure should be caught and handled at the point of its generation. However, a system with a heterogeneous set of components is expected to be error-prone. We cannot guarantee that multiple, dynamically selected NLP/IR components will always follow a standard behaviour, since we view them as black boxes of execution. Secondly, should we choose to decentralize our model by using remote services we cannot always guarantee those services will be accessible. So, we consider the NLP/IR components to be error prone, and at a the system level we focus on handling any errors appropriately.

**2.2 Design Targets**

Besides providing solutions to the challenges discussed in the previous section, we would ideally like the system to address the following set of design targets:

**Additional Functionality**

While our primary target is to combine a set of components into a single logic, we would also like to manage and monitor the information generated, optimize the operations and extract useful information. Therefore we should also provide the following functionality:

- **Logging**: Keeping track of the system’s events will help us trail unexpected or verify normal behaviour in post-execution analysis.
- **Database Management**: Storing the system’s results will help us maintain information about the system’s execution, which is useful for post-execution analysis (e.g. how many queries were served in a single day, with what success rate, etc.).
- **Evaluation metrics**: The components of the system need to be constantly evaluated with appropriate metrics, so that we can assess the quality of their outputs.
- **Statistics**: Managing a complex system is in general a difficult task. We can make this task easier by providing statistics on stored data that inform the system’s administrator about the system’s health, usage etc.
- **Management tools**: Offering tools that automatically handle system operations is in general a user-friendly practice.
Native in Python

We aim to develop our framework natively in Python, as we consider it the most appropriate choice for the following reasons:

- Python is a major trend in Data Sciences with a wide selection of packages available (NLTK, Scikit-Learn, spaCy etc).
- Being a dynamically typed language with native support in semi-structured data, Python helps us skip software overhead in data manipulation.
- Python’s open community and wide selection of available free and open source packages matches our state of mind as a research group.
- Python’s excellent readability matches our need for “easy to understand” source code.

Simplicity

We consider two types of users of the system. The ones that will try to extend it and the ones that will try to use it to build their own Question Answering system. Both should be able (with a basic software engineering background) to comprehend the system’s basics and how it works. Thus we need to provide a simple architecture, class model, and file structure, as well as management tools. Our aim is not to provide a system with a wide selection of available tools for the user to choose from. Instead we aim to offer a system that can be easily mastered, that comes with basic options, but that can be easily extended with additional functionality.

Configurability

The system should be designed in a way that options and tweaking parameters are defined by configuration files. Changes in options should never reflect to changes in source code as changes in the code are much less user-friendly and have to be thoroughly tested to verify that they do not cause errors.

2.3 Solution

In order to address the aforementioned challenges and design targets, we implement a Python framework named Almosino Skeleton that can be used to construct QA systems. In the following sections we will thoroughly go through the framework’s structure and how it is used. Next we mention the main software engineering strategies followed:

- **Graph structure**: The systems created with the framework will follow the structure of a directed execution graph. Each node in the execution graph represents an autonomous

---

1. The popularity of Data Analysis software
2. The python package index
component and each edge represents a data flow from the parent node to the child node.

- **Remote components**: The functionality contained within the framework will not be tailored to any particular NLP/IR/ML component at any point. Such components will be viewed as remote black boxes and will be used with appropriate web APIs.

- **Message oriented communication**: The autonomous nodes communicate via messages that encapsulate all required information.

Following these three basic strategies, our introduction’s question answering paradigm can be presented in the following figure.

![Figure 2.1](image)

*System components communicate with each other by exchanging messages. Notice that system components related to NLP/IR functionality communicate with remote services, that provide the corresponding functionality. In accordance with our introduction’s paradigm we expect the remote IR service to retrieve relevant documents, the remote SE service to extract relevant snippets, and so on.*
3. Almosino Skeleton

We will approach our software engineering task by implementing a lightweight Python framework named “Almosino Skeleton”. Almosino Skeleton can be used in order to combine heterogeneous NLP, IR, ML components into a single robust system. In this chapter we will present the framework’s functionality, structure, core class model and basic logic.

3.1 Functionality

It is crucial to define, first of all, the functionality of our framework: what it provides and what it does not. The functionality provided is of four types::

- **The core software**
  - The software responsible for creating the execution graph, its nodes, the way they communicate, and how the total functionality is bound together.
- **Data manipulation**
  - The software responsible for managing information that is the product or byproduct of each node’s execution. This information can be used to extract statistics, metrics or monitor the system.
- **Tools**
  - The software that helps a user get started with the framework. We provide the tools that help a user create a new component (node of the graph), define a graph (which nodes and how they are connected) and inspect the system (configuration checks etc.).
- **Evaluation**
  - The software that extracts evaluation statistics regarding the system’s performance, health, etc. The extracted statistics can be divided in two main categories: a) evaluation metrics that measure the performance of ML/IR/NLP components (e.g., precision, recall, etc.) and b) system statistics that estimate information regarding the system’s performance (e.g., successful execution rate, average completion time etc.).

3.2 Structure

The framework’s source code is structured in three main packages: Skeleton, Topology and Evaluation. In this section we will describe what type of functionality each package provides, also taking into account the four types of functionality described above.
- **Skeleton:** Being the core package, the skeleton package contains the implementations of most of the core software classes, the system’s tool selection, implementation of metrics regarding the system’s performance, and data manipulation software.

- **Topology:** Initially almost empty, this package contains only the topology core class (to be explained further in section 3.3). However, this is where the system’s components will be built using the framework, so eventually it is where the core software is extended and all the other functionality is used.

- **Evaluation metrics:** Software estimating the performance of the remote counterparts. Currently containing implementations of Machine Learning, Information Retrieval and Summarization evaluation metrics.

Given the type of the functionality contained within each package, we can visualize the dependencies between the packages with the following UML package diagram.

![UML package diagram]

Figure 3.1

*Package dependencies. Notice that the topology package depends on the Skeleton and Evaluation Metrics, but the other packages are independent.*

### 3.3 Core class model

Almosino Skeleton has a simple core class model of 5 classes that handle or encapsulate most of the information processed by the system in general. Understanding each class's role and functionality will help the user to comprehend how the framework basically works.
Message

Defining a communication protocol between different points of execution can prove to be a complex and error-prone task. In Almosino Skeleton we follow a much simpler approach: components communicate with each other by exchanging Message objects. Message objects are basic units that encapsulate information in four basic data structures:

- **Header:** Field containing global variables (such as the mode of execution) and application pass-through parameters (such as the id of the user that requested the query). The header is extended with error messages (if an error comes up) to propagate the warning in the next nodes.
- **Body:** Initially containing the question and the question type, the body is basically the system’s response gradually extended by each node (e.g., with documents, snippets, summary etc). At the end of the execution the body contains the response to be returned to the user.
- **Gold:** Evaluation of a component’s performance is crucial in Data Science systems. In order to evaluate a system properly we need the expected (gold) output to compare it with the system’s output, given the same input. This information is contained within the gold field.
- **Params:** Parameters concerning execution options that can be set on a per-message basis. For example such parameters are the number of documents to be retrieved or the maximum limit of words in summary. In contrast to the header parameters, these variables are oriented in specifying the execution options of one specific remote component.

Node (abstract class)

Nodes are the basic units of organisation in the system. That said, the nodes are more responsible for the control of the system and its evaluation rather than data manipulation and execution. Node objects are the exact equivalent of nodes in the execution graph described in the previous chapter. Their main responsibilities are:

- Inter-Node communication
- Statistics and metrics
- Component configuration
- Worker management

Worker (abstract class)

Workers are thread objects that practically do all the hard work within a component, being the ones that execute the remote call to corresponding remote API and providing the rest of the wrapping functionality. As soon as a Node receives a message, it appends it in queue where it
will be consumed by the first available Worker. While consuming a Message a Worker executes the following operations:

- Runs a message compatibility check routine that checks whether a message contains the information expected (e.g., a Snippet Extraction component expects documents, an Exact Answer Extraction component expects snippets, etc).
- Runs an execution routine that unpacks any necessary data, calls the remote API and extends the initial message with the information obtained.
- Runs (if necessary) an error handling routine that updates the message with appropriate warnings in order to notify the next nodes in execution.
- Controls caching and logs all steps of execution appropriately.
- Creates and stores the Transaction object describing the execution of a single message.
- Returns the updated message to the current Node so it can be forwarded to the next Node in line.

The consumption process of the Worker depends on the “testing” parameter of the Message’s header. If set to True, the Workers will use as input the gold field of the Message. If not the Workers will use as input the body field.

Figure 3.2
Nodes practically do not operate on the messages they receive. Messages are stored within a Queue waiting for the first available Worker to consume them. Recall that the workers are the entities responsible for most of the operations on data (execution, storing, logging and so on).
Notice that basically the consumption of a Message within a Worker is handled by 4 basic routines:

- **Compatibility check**: Checks if the message contains the expected input.
- **Execution**: Unpacks the data from the message, calls the method that executes the remote call and extends the message’s body with the appropriate output.
- **Error handling**: Appends an error warning in the message’s header and extends the body with the appropriate default output.
- **Transaction storing**: Creates a transaction object containing all useful information and stores it in the database.

**Transaction (abstract class)**

Each message consumption is followed by generating transaction information, important especially for post experiment analysis. Such information is:

- Execution input (body and gold).
- Execution output (body and gold).
- When the execution started and when it ended.
- Whether the execution was successful (or errors were raised).
- Which execution option was used.

This information is encapsulated within a Transaction object and stored in a database for future use. Transactions are mostly used to encapsulate data rather than execute operations on these data. However, transactions provide one critical operation, that of filtering out unnecessary information from being stored in the database.

**Topology**

The Node objects are a representation of the nodes of the execution graph. By contrast, the Topology object is the representation of the execution graph itself. Topology, being the top component class in the class model hierarchy (Figure 3.4), is responsible for the allocation of the resources needed to set up the system Nodes, the management of the Nodes, and the communication of the framework and the application layer (assuming the role of the “middle man” the topology delivers new questions to the nodes and returns the responses generated to the application level).

So the basic entities of the system are divided mainly in two categories: a) Entities that are used for encapsulating information and are basically blocks of data, and b) Entities that are responsible for the system’s organisation and execution. The second group, consisting of the classes Topology, Node, Worker follows a strict hierarchy presented in Figure 3.4.
3.4 System components

Thus far, we have presented the framework’s entities and basic functionality. In this section we will describe how to build a system using the framework, which core classes should be extended, how to create the components of the system, and the chain of operations within a component.

Components

A component is an autonomous part of the system that a) uses/extends the core functionality of the framework to connect with other components, b) obtains responses from a remote NLP/IR component, c) stores and d) processes that information. While the framework provides most of this functionality, there are some parts that depend on each component’s use case and remote counterpart. Hence, when creating a new component we must provide extensions for the Node, Worker and Transaction abstract classes that provide the missing functionality.

Extending Node

The framework comes with a selection of evaluation metrics, but which ones to use depends on the use case. For an information retrieval component, we will use different evaluation metrics than for a summarizer. Therefore, we need to extend the Node class and add an evaluator method which is:

- A method whose name ends with “evaluator”.
- Will receive as input the body and gold data.
- Will return a Python dictionary containing the metrics and their values.

Extending Transaction

We cannot pre-determine what filtering a user will apply to the data stored in the database, since it depends on the user needs and the component’s use case. Therefore we have to extend the Transaction class and override the filtering functionality.

Extending Worker

We cannot provide a generic implementation for the call to the remote counterpart. Hence, we need to extend the Worker class and add an executor method which is

- A method whose name ends with “executor”.
- Will receive as input a Python dictionary containing the expected input fields and their values.
- Will return a Python dictionary containing the expected output fields and their values.

Configuration file
Each component’s behaviour depends on a set of configuration parameters that define several execution options and environment variables. Such parameters are the expected input and output fields (and their expected type), which executor method is the default execution option, the connection URI to the database, the number of workers, caching parameters, etc. Therefore, we also need to provide a configuration file containing our options for these parameters. For more information on the configuration files and their format refer to the corresponding appendix section.

So practically, to create a new component we need to create a Python package within the topology folder, add 3 files that contain the extensions of the core classes Node, Worker, Transaction with implementations of the missing functionality and a configuration file.

**Code auto-generation**

Of course the aforementioned process would be against our design target for simplicity, therefore we provide a management tool that auto-generates the folders and files leaving only a few lines of missing code for the user to implement. For example, if we choose to create the component “information retrieval” the tool auto-generates the following:

- A Python package within the topology folder named information_retrieval.
- The package will contain:
  - A file “information_retrieval_node” containing the extension to the Node class.
  - A file “information_retrieval_worker” containing the extension to the Worker class.
  - A file “information_retrieval_transaction” containing the extension to the Transaction class.
  - A .json configuration file with all the parameters the user can provide/tweak.

**Examples**
Figure 3.5

Auto-generated Worker class extension. Notice that the user will just have to implement in line 20 a simple API call (approximately 5~10 lines of code) to create a functional component. Everything else is handled by the framework. More specifically the execution routine at the abstract class Node will create the args input dictionary, call dynamically the executor specified in the configuration file and update the message’s body with the result of the execution.

To demonstrate the simplicity of the framework we attach in the following figures the implementation of one executor and two evaluators for the information retrieval component of our baseline QA system (chapter 4).

Figure 3.6

Example of an executor performing a call to a remote ElasticSearch engine. Notice that given an appropriate API, only a few lines of simple code are needed to link a new component to the system.
Figure 3.7
Example of two basic evaluators for the Information Retrieval node. Notice the simplicity of the code the user has to provide: just unpacking the appropriate field (documents) and calling the evaluation metrics provided by the system.

Figure 3.8
Percentage of auto-generated code (in lines) per component. The auto-generated code includes the classes, the equivalent constructors and some additional methods. Even though it is simple, this code follows strict name conventions, so by providing it we do not only save time for the user, we also prevent possible errors.
4. Almosino QA

We have demonstrated so far the basic structure and logic of the Almosino Skeleton framework. In this chapter we will review how we used the framework to create Almosino QA, a biomedical question answering system, using baseline NLP/IR components.

4.1 Target

Out of the many applications of Question Answering systems we are particularly interested in the biomedical domain. More specifically, we would like our framework to be used in order to construct systems that can participate in the BioASQ\(^3\) data challenge.

BioASQ

BioASQ is a large-scale biomedical semantic indexing and question answering competition. As stated in [1] “BioASQ assesses the ability of systems to semantically index very large numbers of biomedical scientific articles, and to return concise and user-understandable answers to given natural language questions by combining information from biomedical articles and ontologies”.

We are interested in the second part of the BioASQ competitions (Task B), which is the question answering challenge. This challenge is split in two phases: Phase A provides the participants with a set of natural language questions and a large collection of PubMed documents, and requires them to extract relevant documents and snippets. (The participants are also required to return relevant RDF triples and concepts from ontologies, but we are only concerned with texts in this thesis.) Phase B provides the same set of questions as Phase A, along with the gold results of Phase A, and requires the participants to extract exact answers and generate ‘ideal’ answers.

4.2 Baseline components

We will first discuss the baseline components we use in order to generate the system’s responses. Note that these baselines were not intended to perform as a state of the art QA system, but to demonstrate that a) our framework actually works, and b) the system can be used to construct QA applications.

- **Document Retrieval**: For the document retrieval component we have used an ElasticSearch\(^4\) installation as an Information Retrieval engine.
- **Snippet Extraction**: For the snippet extraction process we use a greedy approach. First we split the text in sentences and then we select the snippets by assigning a relevance score to each sentence depending a) on the number of common (non stopword) words

---

\(^3\) The [BioASQ](https://bioasq.org) challenge

\(^4\) The [ElasticSearch](https://www.elastic.co) open source search and analytics software
between the query and the sentence and b) the relevance score of its neighbour sentences (first and second next and previous sentences).

- **Summary extraction:** In order to generate ideal answers (summaries) we use a greedy approach. Starting with an initially empty text, we extend the summary with snippets that have the highest relevance score (obtained during snippet extraction) and contribute the most with new information. The level of contribution of each snippet is measured as the number of (non stopword) words it contains that do not already appear in the summary.

- **Exact answer extraction:** For the exact answer extraction we consider 3 components one for each question type the system supports.
  - **YesNo:** Return Yes if all words of the processed query exist in the snippets.
  - **Factoid:** Return the most common entity in the snippets that does not appear in the query.
  - **List:** Return a list of the most common entities in the snippets that do not appear in the query.

### 4.3 System architecture

The structure of the QA system is more elaborate than the structure introduced in Figure 1.1. The main difference is that we split the Exact Answer Extraction in 3 different components. Messages are routed to each component depending on the type of the question. We also introduce a new component named Message Cache. This component has no NLP/IR related functionality. Since Exact Answer Extraction and Ideal Answer Extraction run in parallel, the role of this component is to store temporarily in a data structure the Messages received so that the results of the 2 processes can be combined before returned to the Entry Node of the topology. The structure of the system is depicted in Figure 4.1.
Figure 4.1

*The Almosino QA component structure. Notice that one component is marked as exit/entry point. That means that this component initially receives the Message from the Topology and returns this Message to the topology once the process is completed.*

4.4 API

The implementation of Almosino QA comes with a very basic, Json based, web API that can be used to a) submit questions, and b) view statistics concerning the performance of a component on a collection of questions. In this section we provide a basic description of the API calls (the parameters of each call and the system responses).

Submit question

In order to submit a question, the user has to post to our web endpoint a Json message containing the following data:

- **user_name(optional)**: The user’s name, set to “anonymous” if not provided.
- **collection_name(optional)**: The collection’s name, set to “test” if not provided.
- **gold(optional)**: A dictionary with the gold data, set to empty if not provided.
- **params**(optional): A dictionary with execution parameters (e.g., number of documents to be retrieved), set to empty if not provided.
- **question**(required): The submitted question.
- **type**(required): The type of the question submitted.
- **testing**(optional): Recall the testing parameter explanation. Set to False if not provided.

The response of the system contains the system's results (output of every system component) in Json format:

- documents
- snippets
- exact-answer
- ideal-answer
- processed_question

**Statistics**

In order to extract statistics concerning the performance of a component in a collection of questions, the user has to post to our web endpoint a json message containing the following data:

- **user_name**(required): The name of the user that submitted the collection.
- **collection_name**(required): The name of the collection of questions.
- **node**(required): The node name of the component. For example for Information Retrieval the equivalent node name is information_retrieval_node (lowercase, separated with underscore).

The response of the system contains two categories of statistics a) health statistics addressing a Node’s performance in general, and b) evaluation metrics, the results of the evaluator methods implemented in the component’s Node.

**Health statistics:**
- Number of successful queries
- Number of failed queries
- Executors used
- Number of testing queries
- Average completion time

**Evaluation Metrics** (recalling the example of Figure 3.7):
- ordered_evaluator
  - Mean Average Precision at k
  - Mean nDCG at k
- unordered_evaluator
  - Mean Precision
4.5 Experiments

In this section, we will list a set of experiments we executed, in order to verify that the system performs as expected.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Links</th>
<th>Aim</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Submit one question in simple mode and one question in testing mode</td>
<td>Experiment 1</td>
<td>Test functionality</td>
<td>Normal response returned</td>
</tr>
<tr>
<td>Submit in testing mode a set of 1303 questions from the BioASQ 4b task.</td>
<td>Experiment 2</td>
<td>Performance, statistics functionality</td>
<td>Queries served, statistics obtained.</td>
</tr>
<tr>
<td>Inject errors in our code to test our system’s robustness</td>
<td>Experiment 3</td>
<td>Robustness</td>
<td>Error caught, not propagated to the system.</td>
</tr>
</tbody>
</table>

Table 4.1

The list of experiments executed to evaluate the system’s performance and robustness.

Experiment 1

Initially we tested the API by submitting a question from the BioASQ 4b competition in non testing and testing modes. The reader is reminded that when executing in testing mode, we use as input the gold fields provided, while when executing in simple (or non testing) mode, we use the output of the previous components as input. The response of our API is shown in Figure 4.2.

At this point we will demonstrate the difference between the two types of responses. Given:

- **Query**: “What are the symptoms of Muenke syndrome?”
- **Type**: “Summary”
The ideal answer returned by our system in testing mode is:

"Muenke syndrome is characterized by coronal craniosynostosis (bilateral more often than unilateral), hearing loss, developmental delay, and carpal and/or tarsal bone coalition. Tarsal coalition is a distinct feature of Muenke syndrome and has been reported since the initial description of the disorder in the 1990s. The facial features of children with FGFR3Pro250Arg mutation (Muenke syndrome) differ from those with the other eponymous craniosynostotic disorders. We documented midfacial growth and position of the forehead after fronto-orbital advancement (FOA) in patients with the FGFR3 mutation. Increased digital markings were more severe posteriorly in Muenke patients than in non-Muenke patients. The Muenke patients with unilateral coronal synostosis showed a somewhat more severe asymmetry in the anterior part of the skull than the non-Muenke patients. Muenke syndrome is a genetically determined craniosynostosis that involves one or both coronal sutures. In some patients it is associated with skeletal abnormalities such as thimble-like middle phalanges, coned epiphysis, and/or neurological impairment, namely sensorineural hearing loss or mental retardation. Muenke syndrome caused by the FGFR3(P250R) mutation is an autosomal dominant disorder mostly identified with coronal suture synostosis, but it also presents with other craniofacial phenotypes that include mild to moderate midface hypoplasia."

The ideal answer returned by our system in simple mode is:

“We sought to determine the demand for CAM by those with neuropsychiatric symptoms compared to those without neuropsychiatric symptoms as measured by out-of-pocket expenditure. METHOD: We compared CAM expenditure between US adults with and without neuropsychiatric symptoms (n = 23,393) using the 2007 National Health Interview Survey. The foods shown to be tolerable were generally bland, sweet, salty, and starchy. CONCLUSIONS: This study identified specific foods that worsen as well as foods that may help alleviate symptoms of gastroparesis. Multiple logistic regression analyses were employed to assess the associated factors of depressive symptoms in people with low eGFR. Therefore, we compared the prevalence of symptoms and medical histories (symptoms or patient-reported diseases) between Yusho patients and healthy controls to demonstrate the effects of Yusho on health conditions. These symptoms are often refractory to standard therapies, and patients may consequently opt for complementary and alternative medicine therapies (CAM). Among 25,324 study participants, 2.9% (n = 723) of all participants had low eGFR, and 16.7% (n = 121) of these participants were self-reported to have depressive symptoms in the low eGFR group. Foods that provoked symptoms differed in quality from foods that alleviated symptoms or were tolerable.”

The administrator of the system can monitor the successful execution of a question by monitoring the log:

matrix_node [Thread-1] [INFO] Message received
matrix_node [Thread-1] [INFO] Message input is compatible.
matrix_node [Thread-1] [INFO] Message handled successfully by execution routine.
information_retrieval_node [Thread-2] [INFO] Message received
information_retrieval_node [Thread-2] [INFO] Message input is compatible.
information_retrieval_node [Thread-2] [INFO] Message handled successfully by execution routine.
snippet_extraction_node [Thread-3] [INFO] Message received
matrix_node [Thread-1] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [INFO] Message input is compatible.
information_retrieval_node [Thread-2] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [INFO] Message handled successfully by execution routine.
summary_extraction_node [Thread-4] [INFO] Message received
exact_yesno_node [Thread-5] [INFO] Message received
summary_extraction_node [Thread-4] [INFO] Message input is compatible.
exact_yesno_node [Thread-5] [INFO] Message input is compatible.
summary_extraction_node [Thread-4] [INFO] Message handled successfully by execution routine.
exact_yesno_node [Thread-5] [INFO] Message handled successfully by execution routine.
snippet_extraction_node [Thread-3] [INFO] Transaction stored successfully.
exact_yesno_node [Thread-5] [INFO] Transaction stored successfully.
summary_extraction_node [Thread-4] [INFO] Transaction stored successfully.
Figure 4.2

The API's response to a query. Notice that the exact_answer field returns no actual value since the type of the question is “summary”.
Experiment 2

During this thesis, a lot of effort was spent on creating a framework that is lightweight and scalable. Ideally, we would like to test our system with batches of queries that are submitted in burst mode. (By burst mode we mean that there is no time gap between one request and the next one.) For that purpose we used a set of 1303 queries from the BioASQ 4b task. We submitted the questions using one, two and four processes to send data increasing the burstiness of the data transmission. We tuned the system to use one, four and eight workers in each Node. We repeated the experiments 3 times each, after disabling any caching functionality used by our system or the external components. All the experiments terminated with the system handling the workload successfully. Figure 4.3 shows the results of one experiment for the information retrieval node.

```
[{
  "Average completion time": 0.0005418265541059094,
  "Executors": [
    "elastic_search_5_executor"
  ],
  "Failed queries": 0,
  "Node name": "InformationRetrievalNode",
  "Number of queries": 1303,
  "Success rate": 1.0,
  "Successful queries": 1303,
  "Testing queries": 1303
},
{
  "ordered_evaluator": {
    "Mean NDCG @ k": 0.29867483735720657,
    "Mean average precision @ k": 0.15209789525274414
  },
  "unordered_evaluator": {
    "Mean F-score": 0.18996907513479586,
    "Mean Precision": 0.21673062164236381,
    "Mean Recall": 0.23167509379261436
  }
}
]
```

Figure 4.3

Statistics concerning the information retrieval node execution in a single experiment as returned by the system’s API.
Figures with statistics concerning other nodes are provided in the appendix section of Experiment 2.

One of the most interesting statistics is the time spent per request on the operations of the framework compared to the time spent per request on the NLP/IR components. On average our system with the given baseline components needs 0.049 - 0.06 seconds to complete a single request. On average 0.037 - 0.053 seconds are spent on NLP/IR remote counterparts meaning that our framework’s operations on average require 0.007 - 0.012 seconds.

![Points scored](image)

**Figure 4.4**

The percentage of time per request consumed by the NLP/IR functionality and the framework. Notice that the percentage consumed by the framework is increased by the fact that we use baseline, and therefore faster, components. In a system with actual NLP/IR components the framework would probably consume less than 5%.

We have mentioned that the system needs between 0.049 and 0.06 seconds to complete a single request. However, when executing batches, due to the graph structure of our topology, as soon as a component finishes the processing of a Message, it begins the processing of the next one. This means that before our system completely processes a Message it has already began processing the next 3-5 Messages even when executing with a single worker in each component. We have estimated the worst cases of average per request execution by dividing the time required to complete all requests to the number of queries. Results are demonstrated in Table 4.2 and Figure 4.5
<table>
<thead>
<tr>
<th>Number of workers per node</th>
<th>Number of spamming processes</th>
<th>Worst total time required</th>
<th>Average time per request</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8</td>
<td>64 (seconds)</td>
<td>0.0491 (seconds)</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>62 (seconds)</td>
<td>0.0476 (seconds)</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>59 (seconds)</td>
<td>0.0453 (seconds)</td>
</tr>
</tbody>
</table>

Table 4.2

Average time of completion per request with number of workers per node tuning.

Figure 4.5

Worst and best average time of execution in single request execution and batch execution with one, four and eight workers. More threads per node provide better performance in general, however the difference is only a few milliseconds. This is explained by a) the fact that the graph-structure of the topology makes the process a-priori multi-threaded and b) the performance of our baseline components. In a system with actual NLP/IR components, the number of workers per component would have greater impact on the system’s performance.
Experiment 3

Recall our need to provide a robust system that will not be affected by unexpected behaviour in NLP/IR remote components. In order to test the robustness of our system we executed 2 experiments. First we injected a line of code that executes division by zero in the snippet extraction executor. Then we changed the snippet extraction remote component to return a text message instead of a list of snippets. The first case was handled within the consumption routine of the snippet extraction Worker, while the second was diagnosed and handled by the compatibility check routine of the next component. The results returned are presented in Figure 4.6 and Figure 4.7.

```json
{
  "body": "what are the symptoms of Muenke syndrome?",
  "exact_answer": null,
  "ideal_answer": [
    "",
  ],
  "processed_query": "symptoms Muenke syndrome?",
  "snippets": [],
  "type": "summary"
}
```

Figure 4.6

The API response. Notice that no errors are propagated to the user, just an empty response.

```json
[
  {
    "Average completion time": 0.011285166730548102,
    "Executors": [
      "greedy_baseline_executor"
    ],
    "Failed queries": 0,
    "Node name": "SnippetExtractionNode",
    "Number of queries": 2609,
    "Success rate": 0.999616713836719,
    "Successful queries": 2608,
    "Testing queries": 2609
  },
  {
    "ordered_evaluator": {
      "Mean NDCG @ k": 0.034986893755869941,
      "Mean average precision @ k": 0.000571749955936876
    },
    "unordered_evaluator": {
      "Mean F-score": 0.023947327326540855,
      "Mean Precision": 0.026789366853169738,
      "Mean Recall": 0.02553398092953094
    }
  }
]
```

Figure 4.7

Statistics for the snippet extraction node. Notice that the success rate is no longer 100%.
The logs below show at which point the error appeared and at which point it was handled.

**Exception injection**

matrix_node [Thread-1] [INFO] Message received
matrix_node [Thread-1] [INFO] Message input is compatible.
matrix_node [Thread-1] [INFO] Message handled successfully by execution routine.
information_retrieval_node [Thread-2] [INFO] Message received
information_retrieval_node [Thread-2] [INFO] Message input is compatible.
information_retrieval_node [Thread-2] [INFO] Message handled successfully by execution routine.
snippet_extraction_node [Thread-3] [INFO] Message received
snippet_extraction_node [Thread-3] [INFO] Message input is compatible.
matrix_node [Thread-1] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [ERROR] integer division or modulo by zero
information_retrieval_node [Thread-2] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [ERROR] Execution handling routine failed to handle message: <src.skeleton.core.message.Message instance at 0x7fa6cc74bf38>
snippet_extraction_node [Thread-3] [WARNING] Message handled successfully by error handler.
summary_extraction_node [Thread-4] [INFO] Message received
summary_extraction_node [Thread-4] [WARNING] Message input is incompatible.
summary_extraction_node [Thread-4] [WARNING] Message handled successfully by error handler.
snippet_extraction_node [Thread-3] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [INFO] Message handled successfully by execution routine.
information_retrieval_node [Thread-2] [INFO] Message received
information_retrieval_node [Thread-2] [INFO] Message input is compatible.
information_retrieval_node [Thread-2] [INFO] Message handled successfully by execution routine.
matrix_node [Thread-1] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [INFO] Message received
snippet_extraction_node [Thread-3] [INFO] Message input is compatible.
information_retrieval_node [Thread-2] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [INFO] Message handled successfully by execution routine.
summary_extraction_node [Thread-4] [INFO] Message received
summary_extraction_node [Thread-4] [WARNING] Message input is incompatible.
snippet_extraction_node [Thread-3] [INFO] Transaction stored successfully.
summary_extraction_node [Thread-4] [WARNING] Message handled successfully by error handler.
summary_extraction_node [Thread-4] [INFO] Transaction stored successfully.

**Incompatible output**

matrix_node [Thread-1] [INFO] Message received
matrix_node [Thread-1] [INFO] Message input is compatible.
matrix_node [Thread-1] [INFO] Message handled successfully by execution routine.
information_retrieval_node [Thread-2] [INFO] Message received
information_retrieval_node [Thread-2] [INFO] Message input is compatible.
information_retrieval_node [Thread-2] [INFO] Message handled successfully by execution routine.
matrix_node [Thread-1] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [INFO] Message received
snippet_extraction_node [Thread-3] [INFO] Message input is compatible.
information_retrieval_node [Thread-2] [INFO] Transaction stored successfully.
snippet_extraction_node [Thread-3] [INFO] Message handled successfully by execution routine.
summary_extraction_node [Thread-4] [INFO] Message received
summary_extraction_node [Thread-4] [WARNING] Message input is incompatible.
snippet_extraction_node [Thread-3] [INFO] Transaction stored successfully.
summary_extraction_node [Thread-4] [WARNING] Message handled successfully by error handler.
summary_extraction_node [Thread-4] [INFO] Transaction stored successfully.
5. Conclusions

We have addressed the problem of combining different NLP/IR components in a robust manner by implementing a framework that can connect multiple components in a robust and agile way. Besides question answering, the Almosino Skeleton framework can be used by any Data Science related system that aims to join together multiple components. In this chapter we return to the challenges and design targets that were presented in the second chapter to discuss how they were addressed. We will also list some tasks for future work on both the framework and the Almosino application.

5.1 Challenges

Let us begin with the challenges presented and why we think that systems created with the Almosino framework will not be affected by the problems these cases cause.

Programming Language heterogeneity

We deal with this problem by handling the heterogeneous components as remote counterparts. The source code developed with this strategy is much easier to debug and less error-prone than using wrappers, sockets or pipelines. Communication with REST services is widely used in the software industry and such services are available (and well documented) in most - if not all - programming languages.

System cost and efficiency

This challenge was addressed by following a decentralized logic for the system. Each NLP/IR component can be set in a different machine, specifically optimised for that component. Thus, we can create systems that when deployed on the cloud will be much less costly than a centralized installation. Moreover, our multi-threaded and message oriented approach ensures that in a continuous flow of messages no component remains idle. For example, when a message is under the information retrieval stage another would be in the summarization and so on.

System agility

In this thesis a lot of effort was spent in designing the framework to be as agile as possible. We provided a solution that addresses the problem of dynamic components and execution options since adding new components and execution options involves no re-engineering in the framework and only a few lines of code in general.
Robustness

As mentioned in section 4.5, the system’s robustness was tested by injecting errors in the source code. However, as we will mention in the future work section, we would like this process to become a little bit smarter and less strict in error-handling decisions.

5.2 Design Targets

Configurability

This design target was the easier to address. Each component contains a configuration file (autogenerated like the python files) with any execution parameters needed. No execution parameter at any point is hard-coded inside the code.

Simplicity

Simplicity is hard to estimate since developers in general tend to be biased concerning the simplicity and readability of their source code. However, there are some indicators that the system we provide does not have a large learning curve. Such indicators are the relatively simple class model, the easy to understand graph structure of the system, the message oriented communication protocol, the auto-generation of code and the simple code snippets the user has to provide in order to create executors/evaluators as presented in section 3.4.

5.3 Future work

More execution options

Using baseline systems does not demonstrate the full potential of our framework. Being simple, the baselines are less error-prone and certainly much faster than actual systems. By adding actual systems, we can demonstrate the usefulness of the statistics provided by the system and the robustness of our error-handling routines.

Smart error-handling

Right now, the error handling is very strict and not too smart. We would like first of all to make configurable the decision for each component to either continue or skip the execution if an error happens in the previous components. We would also like to handle smartly the error-cases. More specifically, we would like our workers to dynamically start using another execution option (if available) if they start getting too frequently errors from their current execution option.
Web exposure

Almosino should have a web application where users can upload their queries (or collections of queries) and view the system's responses. The web application should also contain an administrator plugin that the system's administrator can use to set-up the system, change configuration options, view the system's statistics in a more user-friendly fashion.

Queue execution

While the system, due to its threaded character, successfully serves requests arriving on burst mode, it can not be guaranteed that with slower NLP/IR components and real-life application demands the system will not face performance issues. A common practice to resolve this problem is to execute asynchronously incoming requests utilising task queues.
6. Appendix

Configuration files

Example of a configuration file (information retrieval component's configuration). Some fields values are hidden for security/privacy reasons.

```json
{
  "cache": {
    "active": false,
    "active_all": false,
    "compress": false,
    "mmap_mode": null,
    "verbose": 1,
    "cache_dir": "..."
  },
  "cls_name": "information_retrieval_node",
  "db_args": {
    "connection_uri": "...",
    "db": "alosino",
    "manager_type": "mongo_manager"
  },
  "description": "Document retrieval component: Retrieves documents relative to the query provided.",
  "executor": "elastic_search_5_executor",
  "expected_fields_mapping": {
    "input": {"processed_query": {"expected_type": "unicode"}},
    "output": {"documents": {"expected_type": "list", "default": []}}
  },
  "logging": {
    "handler": [\n      "information_retrieval_node_file_handler",
      {\n        "class": "logging.handlers.RotatingFileHandler",
        "filename": "...",
        "formatter": "standard",
        "level": "INFO"
      }\n    ],
    "logger": [\n      "information_retrieval_node",
      {\n        "handlers": [\n          "console",
          "information_retrieval_node_file_handler"
        ],
        "level": "INFO",
        "propagate": true
      }\n    ],
    "name": "InformationRetrievalNode",
    "num_of_workers": 1
  }
}
```

Figure 6.1

Configuration file for the information retrieval component. The file is auto-generated, in this case the fields that have been altered are the executor, expected_fields_mapping, and database connection uri.
Experiment 2

Testing the system on batch question submissions.

```
[
  {
    "Average completion time": 0.010278587874136606,
    "Executors": [
      "greedy_baseline_executor"
    ],
    "Failed queries": 0,
    "Node name": "SnippetExtractionNode",
    "Number of queries": 1303,
    "Success rate": 1.0,
    "Successful queries": 1303,
    "Testing queries": 1303
  },
  {
    "ordered_evaluator": {
      "Mean NDCG @ k": 0.035013744787148429,
      "Mean average precision @ k": 0.0086638181279003572
    },
    "unordered_evaluator": {
      "Mean F-score": 0.023965705929382405,
      "Mean Precision": 0.026809925812228193,
      "Mean Recall": 0.025553577231088521
    }
  }
]
```

Figure 6.2

_Statistics for the snippet extraction node_
"Average completion time": 0.0692916346891788182,
"Executors": [ "greedy_baseline_executor"
],
"Failed queries": 0,
"Node name": "SummaryExtractionNode",
"Number of queries": 1303,
"Success rate": 1.0,
"Successful queries": 1303,
"Testing queries": 1303
},
"bleu_metrics_evaluator": {
  "bleu_score": 0.09501401116431227
},
"rouge_metrics_evaluator": {
  "rouge_1_f_score": 0.28235,
  "rouge_1_f_score_cb": 0.27862,
  "rouge_1_f_score_ce": 0.2802,
  "rouge_1_precision": 0.20396,
  "rouge_1_precision_cb": 0.20648,
  "rouge_1_precision_ce": 0.20756,
  "rouge_1_recall": 0.72274,
  "rouge_1_recall_cb": 0.7177,
  "rouge_1_recall_ce": 0.72806,
  "rouge_2_f_score": 0.19981,
  "rouge_2_f_score_cb": 0.18557,
  "rouge_2_f_score_ce": 0.18733,
  "rouge_2_precision": 0.11229,
  "rouge_2_precision_cb": 0.11904,
  "rouge_2_precision_ce": 0.12553,
  "rouge_2_recall": 0.43314,
  "rouge_2_recall_cb": 0.42624,
  "rouge_2_recall_ce": 0.44099,
  "rouge_3_f_score": 0.13116,
  "rouge_3_f_score_cb": 0.12709,
  "rouge_3_f_score_ce": 0.13501,
  "rouge_3_precision": 0.09528,
  "rouge_3_precision_cb": 0.09191,
  "rouge_3_precision_ce": 0.09847,
  "rouge_3_recall": 0.32534,
  "rouge_3_recall_cb": 0.31836,
  "rouge_3_recall_ce": 0.33251,
  "rouge_4_f_score": 0.11182,
  "rouge_4_f_score_cb": 0.10784,
  "rouge_4_f_score_ce": 0.1157,
  "rouge_4_precision": 0.08163,
  "rouge_4_precision_cb": 0.07639,
  "rouge_4_precision_ce": 0.0847,
  "rouge_4_recall": 0.26944,
  "rouge_4_recall_cb": 0.26229,
  "rouge_4_recall_ce": 0.27662,
  "rouge_5_f_score": 0.21022,
  "rouge_5_f_score_cb": 0.20684,
  "rouge_5_f_score_ce": 0.21369,
  "rouge_5_precision": 0.15048,
  "rouge_5_precision_cb": 0.1476,
Figure 6.3

Statistics for the ideal answer extraction node. For the extraction of rouge scores we used the Rouge [2] toolkit and for the extraction of the bleu score we used Natural Language Toolkit [3]
Figure 6.4
Statistics for the exact answer extraction component generating factoid answers.

Figure 6.5
Statistics for the exact answer extraction component generating list answers.

[{
  "Average completion time": 0.0066840490797546023,
  "Executors": [
    "baseline_factoid_executor"
  ],
  "Failed queries": 0,
  "Node name": "ExactFactoidNode",
  "Number of queries": 326,
  "Success rate": 1.0,
  "Successful queries": 326,
  "Testing queries": 326
},
{
  "mrr_factoid_evaluator": {
    "mean_reciprocal_rank": 0.058282208588957052
  }
}]

[{
  "Average completion time": 0.0076738461538461544,
  "Executors": [
    "baseline_list_executor"
  ],
  "Failed queries": 0,
  "Node name": "ExactListNode",
  "Number of queries": 325,
  "Success rate": 1.0,
  "Successful queries": 325,
  "Testing queries": 325
},
{
  "unordered_list_evaluator": {
    "mean_f_score": 0.027788949567654642,
    "mean_precision": 0.02,
    "mean_recall": 0.055884850192542503
  }
}]

43/44
Figure 6.6

Statistics for the exact answer extraction component generating yes/no answers

```json
[
  {
    "Average completion time": 0.005572207084686656,
    "Executors": [
      "baseline_yesno_executor"
    ],
    "Failed queries": 0,
    "Node name": "ExactYesnoNode",
    "Number of queries": 367,
    "Success rate": 1.0,
    "Successful queries": 367,
    "Testing queries": 367
  },
  {
    "unordered_yesno_evaluator": {
      "accuracy": 0.13623978201634879
    }
  }
]```
References

