Project Number 780251

D5.6 Text Processing Pipelines (Final Version)

Version 1.0
8 July 2020
Final

Public Distribution

Edge Hill University

Project Partners: **Alpha Bank**, **ATB**, **Centrum Wiskunde & Informatica**, **CLMS**, **Edge Hill University**, **GMV**, **OTE**, **SWAT.Engineering**, **The Open Group**, **University of L'Aquila**, **University of Namur**, **University of York**, **Volkswagen**

Project Partner Contact Information

Alpha Bank Vasilis Kapordelis 40 Stadiou Street 102 52 Athens Greece Tel: +30 210 517 5974 E-mail: vasileios.kapordelis@alpha.gr	**ATB** Sebastian Scholze Wiener Strasse 1 28359 Bremen Germany Tel: +49 421 22092 0 E-mail: scholze@atb-bremen.de
Centrum Wiskunde & Informatica Tijs van der Storm Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 9333 E-mail: storm@cwi.nl	**CLMS** Antonis Mygiakis Mavrommataion 39 104 34 Athens Greece Tel: +30 210 619 9058 E-mail: a.mygiakis@clmsuk.com
Edge Hill University Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk	**GMV Aerospace and Defence** Almudena Sánchez González Calle Isaac Newton 11 28760 Tres Cantos Spain Tel: +34 91 807 2100 E-mail: asanchez@gmv.com
OTE Theodoros E. Mavroeidakos 99 Kifissias Avenue 151 24 Athens Greece Tel: +30 697 814 7618 E-mail: tmavroeid@ote.gr	**SWAT.Engineering** Davy Landman Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 633754110 E-mail: davy.landman@swat.engineering
The Open Group Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org	**University of L′Aquila** Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L'Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it
University of Namur Anthony Cleve Rue de Bruxelles 61 5000 Namur Belgium Tel: +32 8 172 4963 E-mail: anthony.cleve@unamur.be	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Volkswagen Behrang Monajemi Berliner Ring 2 38440 Wolfsburg Germany Tel: +49 5361 9-994313 E-mail: behrang.monajemi@volkswagen.de	

Document Control

Version	Status	Date
0.1	Document outline	21 April 2020
0.2	First draft	10 June 2020
0.3	First full draft	15 June 2020
0.4	Further editing draft	1 July 2020
0.5	Internal review	3 July 2020
0.6	Changes based on internal review	8 July 2020
1.0	Final	8 July 2020

Table of Contents

1 Introduction **1**

1.1 Overview . . . 1

1.2 Intentions . . . 2

1.3 Outcomes . . . 2

2 Background and Literature Review **3**

2.1 Text Mining & Text Processing . . . 3

2.2 Natural Language Processing Tasks . . . 3

2.2.1 Natural Language Processing Toolkits . . . 6

2.3 Processing Pipelines . . . 6

2.4 Parallel and Distributed Processing Frameworks . . . 7

2.4.1 Apache UIMA DUCC . . . 7

2.4.2 Apache Spark . . . 7

2.4.3 Apache Flink . . . 8

2.5 Natural Language Processing in Typhon . . . 9

3 Methodology **11**

3.1 Datasets . . . 11

3.1.1 Amazon Product Reviews Dataset . . . 11

3.1.2 CoNLL-2003 - Named Entity Recognition Dataset . . . 12

3.1.3 Custom Weather Named Entity Recognition Dataset . . . 12

3.2 Use case Scenarios . . . 12

3.2.1 Experimental Use Cases . . . 13

3.2.2 Partners Use Cases . . . 14

3.3 Experimental Settings . . . 15

3.3.1 Experimental Setup . . . 15

3.3.1.1 Cluster Configuration . . . 15

3.3.1.2 Data Pre-processing . . . 16

3.3.1.3 NLP Toolkit . . . 16

3.3.2 Evaluation Settings . . . 16

3.3.3 Controlled Permutations . . . 17

4 Results **18**
4.1 Experiment Evaluation . . . 18
4.1.1 Speed . . . 18
4.1.2 Data Size . . . 18
4.1.3 Throughput . . . 18
4.2 Distributed Text Processing Framework Design Evaluation . . . 19

5 Evaluation **21**
5.1 Discussion . . . 21
5.1.1 Importance of Serialisability in Distributed Frameworks . . . 21
5.1.2 Data and Task Parallelism . . . 21

6 Natural Language Analysis Engine **22**
6.1 Overview . . . 22
6.2 Development Technologies . . . 24
6.2.1 Libraries . . . 24
6.2.2 Technologies . . . 25
6.3 Description of Components . . . 25
6.3.1 External Facing NLAE REST API . . . 25
6.3.1.1 **processText** . . . 26
6.3.1.2 **queryTextAnalytics** . . . 26
6.3.1.3 **deleteDocument** . . . 27
6.3.2 Internal Components . . . 27
6.3.2.1 **Job Manager** . . . 27
6.3.2.2 **Staging Manager** . . . 28
6.3.2.3 Job Executor . . . 28
6.3.2.4 Flink Monitoring API . . . 28
6.4 Deployment and Scalability . . . 28

7 Risks and Limitations **30**

8 Typhon Requirements **31**

9 Conclusion **33**

Executive Summary

In this deliverable we present the results of experimental testing of Natural Language Processing (NLP) pipelines within parallel and distributed frameworks. These pipelines previously discussed in deliverable D5.4, were evaluated in this report from three performance perspectives; namely speed, data size and throughput. This report firstly begins by reviewing prior work in text processing, describes what constitutes a Natural Language Processing (NLP) pipeline and discusses three NLP tasks adopted for scrutiny in this report, i.e., Named Entity Recognition, Sentiment analysis and Co-reference resolution. We then discuss distributed frameworks which can be used to scale out the processing of each of the pipelines, i.e., Apache Spark, Apache Flink, UIMA DUCC, and therefore provide practical performance gains due to parallelism. Following on from the literature review of distributed technologies, we present the methodology used for experimental testing by detailing the datasets used, along with the technical setting used to perform NLP tasks. In our results, we report the performance benchmarks of NLP tasks produced in distributed environments and discuss the important findings discovered while using distributed frameworks for this purpose. We then outline how the pipelines are integrated within a Natural Language Analysis Engine (NLAE), which exposes NLP functionality within the TYPHON ecosystem through a RESTful API. Lastly, we discuss risks and limitations to testing of NLP pipelines within distributed environments, review project requirements related to text processing and conclude with remarks on future work.

1 Introduction

Natural Language Processing (NLP) pipelines are fundamental in building generative knowledge from unstructured textual data [1]. Recently, research and development in NLP has turned towards distributed computing [2, 3] in order to mitigate the computational cost of extracting knowledge by orchestrating resources within a cluster of computation nodes. In particular, technologies such as Spark and Flink have become de-facto standards for scaling out computations in both academia and industry [4], whereas other frameworks such as UIMA DUCC [5] have been developed as alternative frameworks to provide built-in support for distributed text processing.

With respect to TYPHON, this deliverable follows on from work completed in D2.2 and D5.4. In this deliverable, we present the findings of computational benchmarks for NLP pipelines within a controlled environment. We used a small cluster of standard specification computers to investigate the ability of NLP pipelines to run in a distributed processing environment, and evaluated the success of this attempt. The configuration of the cluster we employed consisted of a master node which managed the execution of jobs, a dedicated Elasticsearch back-end node which served as our data source/sink and two slave nodes which were dedicated for the computing tasks within each pipeline. We implemented the NLP pipelines in all cases using the Stanford CoreNLP toolkit as it is very popular within the literature. Lastly, we sourced samples for experimental testing using Amazon product reviews [6], the CoNLL-2003 dataset [7] and a custom-built dataset to conduct tests for all pipelines under each distributed framework.

Our investigation into Distributed Text Processing Frameworks (DTPF) identified significant performance gains when comparing distributed frameworks to a linear Java baseline. We also discovered architectural limitations to the Stanford CoreNLP toolkit when used in distributed environments and also identified drawbacks of DUCC, when processing pipelines within a cluster of small size. These results indicate that scaling out NLP tasks is an effective strategy to reduce computational cost. However, the results also reveal dependencies on toolkits used in such pipelines and limitations associated to frameworks that require a large number of cluster nodes. Our results imply that researchers pursuing performance gains in parallel processing of NLP pipelines should consider carefully the cluster configuration, the toolkit compatibility and granularity in distributed environments, and the dependencies of a framework within a cluster. Consequently, the experimentation and evaluation performed led to the design and development of the Natural Language Analysis Engine (NLAE). The NLAE, integrated into TYPHON, provides natural language analysis services in the context of unstructured textual data.

1.1 Overview

The deliverable consists of eight sections. Section 2 focuses on presenting the NLP pipelines that were developed to test the text processing capabilities of different Distributed Text Processing Frameworks (DTPF). It also provides related background and reviews existing frameworks that offer parallel processing. Section 3 outlines the methodological procedure used to systematically test NLP tasks within a cluster configuration. In section 4, we present the findings of running NLP pipelines using different DTPFs and discuss the levels of performance achieved in each context. Section 5 examines the significant findings from experimental testing and discusses the importance of the results for future development in distributed NLP research. Section 6 discussed how the outcome of the pipeline development was incorporated into the NLAE, which integrates NLP functionality into TYPHON. In section 7, we elaborate on the risks and limitations encountered in testing, which could reinforce further research in the field to alleviate the challenges faced in our objectives. Lastly, section 8 looks at the progress made towards the TYPHON project requirements and section 9 concludes this deliverable and outlines avenues for future work.

1.2 Intentions

The objective of this deliverable is to provide details about the design and development of the Natural Language Analysis Engine (NLAE), which is integrated into TYPHON to provide dedicated NLP functionality. To this effect, we present the findings of our experimentation with distributed NLP pipelines using existing DTPFs and highlight the similarities among them in terms of performance on common NLP tasks. This report aims to discuss the difficulty in configuring frameworks for distributed use and examine underlying dependencies which could make one alternative more viable than another.

1.3 Outcomes

The primary outcome of this deliverable is the Natural Language Analysis Engine (NLAE) that has been implemented to be integrated into TYPHON to provide natural language analysis capabilities. The design decisions of the NLAE were guided by a set of evaluation experiments that we conducted to generate benchmarks for NLP pipelines under different Distributed Text Processing Frameworks (DTPF). These pipelines were developed for experimental testing of Named Entity Recognition (NER), Sentiment Analysis and Co-reference Resolution, which are some of the most common NLP tasks in practice.

2 Background and Literature Review

In this section, we review literature related to Natural Language Processing pipelines, namely the three tasks adopted for the purposes of this deliverable: Named Entity Recognition, Sentiment Analysis and Co-reference Resolution. We also discuss the Distributed Text Processing Frameworks used in the experimental evaluation and provide a comparison of their features for distributed processing.

2.1 Text Mining & Text Processing

Text mining refers to methods and approaches used to facilitate "the discovery by computer of new, previously unknown information, by automatically extracting information from different written resources" [8]. Its theoretical basis is closely related to data mining, however it principally differs, as natural language is the unit of analysis denoting both implicit (i.e., semantic) or explicit (i.e., syntactic) representations within text.

Text processing is a specialisation of text mining which aims to automatically produce outputs (i.e., annotations) from a source, in order to leverage deeper knowledge embedded within such unstructured text. Text processing is typically offered in NLP pipelines, i.e. chains of components that analyse and manipulate textual information to generate helpful outputs as annotated text. In the following subsections, we discuss common NLP tasks and how they are chained together to build text processing pipelines.

2.2 Natural Language Processing Tasks

In deliverable D5.4, we developed a type system consisting of tasks associated with NLP functions. In this report, we present experimental testing of three core NLP features distributed across a physical cluster which are a subset of the TYPHON type system. The following are some of the most common components that provide NLP functionality.

Tokenisation: Tokenisation is the process of dividing the input text into subunits called *tokens* [9]. A Tokeniser is a component that performs the tokenisation of input text. It takes a stream of characters as an input and splits these to generate a list of tokens based on a pre-defined model or rule-base. Figure 1 shows a simple example of a tokeniser that splits the sentence into individual tokens.

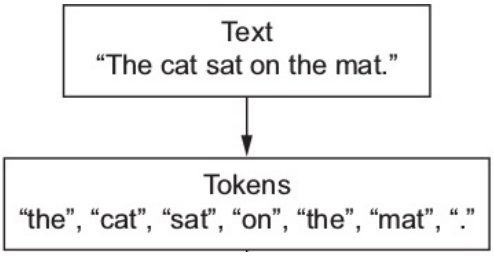

Figure 1: Text Tokenisation Example

Sentence Splitting: Sometimes, NLP applications require large complex input text to be split into sentences, to mine more meaningful information out. Sentence splitters are components that provide the functionality to split large pieces of text into smaller manageable and intelligible sentences [10]. Figure 2 shows a basic example of a sentence splitter.

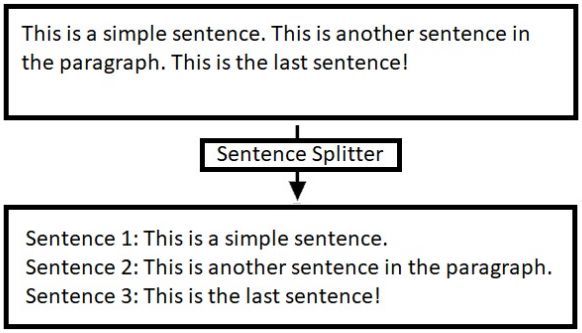

Figure 2: Sentence Splitting Example

N-Grams: N-Grams are sets of co-occurring *n* words/tokens within a given context window [11], where *n* is a real number. N-Gram extraction, as a statistical NLP technique, is very useful in calculating the probabilities of occurrence and co-occurrence of terms, which helps define statistical dependence relations between them in text. For example, for the sentence *"The cat sat on the mat"* figure 3 shows the n-grams where n=2.

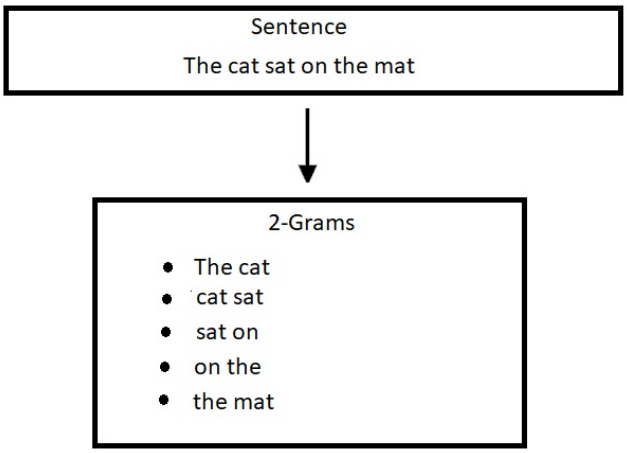

Figure 3: N-Grams Example

Part-Of-Speech (POS) Tagging: The process of labeling words according to the lexical category they belong to is called Part-of-Speech (POS) tagging. A token in text can be a "noun", "verb", "pronoun" etc. according to the role it plays in the sentence. POS tagging is used to highlight these roles played by the tokens in the context of the given input to build syntax trees. Figure 4 shows the corresponding POS tags for the sentence *"I like to play football"*.

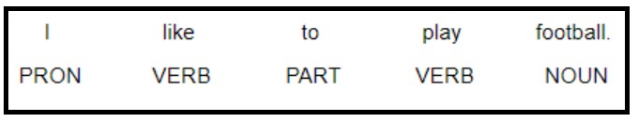

Figure 4: Part-Of-Speech Tagging Example. PRON stands for pronoun and PART for infinitive particle.

Named Entity Recognition: Named Entity Recognition (NER) is the task of extracting named entities from unstructured text, which correspond to pre-defined categories, such as names, organisations, locations and time. Figure 5 shows an example of NER annotation on text. NER is important mainly because it is a foundation for further high-level NLP tasks, which depend on entity references, such as Co-reference resolution. For this reason, NER was considered in the benchmark experiments across distributed frameworks.

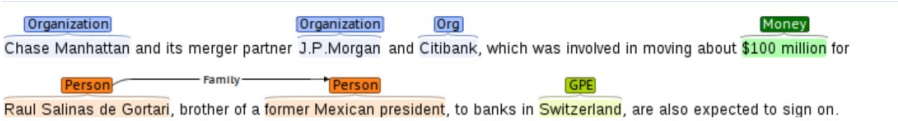

Figure 5: Named Entity Recognition Example

Sentiment Analysis: Sentiment Analysis (SA) is the semantic procedure of extracting opinions from text. Its output is usually represented in the form of a polarity, positive and negative. Sentiment analysis is critical in contemporary NLP because a majority of unstructured text is sourced from consumers or actors, whose opinions may be subjectively skewed based on their individual beliefs and consequently by their utterances. Due to the prominence and need of identifying sentiment in text, we have chosen to investigate this NLP task in a distributed manner. Figure 6 shows results of a sentiment analysis example.

Loves the German bakeries in Sydney. Together with my imported honey it feels like home	Positive
@VivaLaLauren Mine is broken too! I miss my sidekick	Negative
Finished fixing my twitter...I had to unfollow and follow everyone again	Negative
@DinahLady I too, liked the movie! I want to buy the DVD when it comes out	Positive
@frugaldougal So sad to hear about @OscarTheCat	Negative
@Mofette briliant! May the fourth be with you #starwarsday #starwars	Positive

Figure 6: Sentiment Analysis Example

Co-reference Resolution: Co-reference resolution is the procedure of identifying the syntactic relationship between entities and expressions that refer to them within a document as shown in figure 7. Co-reference resolution has been studied extensively, as its application in discourse analysis uncovers background knowledge, which can be important in text summarisation. For this purpose, we have chosen to include Co-reference resolution in our experiments using distributed frameworks.

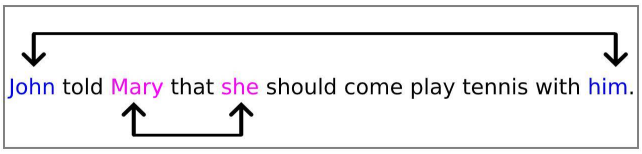

Figure 7: Co-reference Resolution Example

2.2.1 Natural Language Processing Toolkits

A toolkit can be defined as a *"a single utility program, a set of software routines or a complete integrated set of software utilities that are used to develop and maintain applications and databases"*. Toolkits provide researchers and developers with functions and routines that work out-of-the-box to prototype and build software applications quickly. There is a wide variety of open source NLP tools available in many programming languages.

Stanford CoreNLP[1] is a well-known and one of the most popular NLP toolkits, widely used in academic research. Standford CoreNLP provides statistical NLP, deep learning NLP, and rule-based NLP functionality for multiple programming languages including Java. CoreNLP provides turn-key functionality for most of the NLP tasks with minimum code customisation. Moreover, it provides pre-trained models for different NLP tasks available for multiple languages. The CoreNLP library is often referred to as the state-of-the-art for NLP and it is for these reasons that we have used Stanford CoreNLP to build our pipelines for the Natural Language Analysis Engine (NLAE).

OpenNLP[2] is an Apache Foundation project that also provides pre-built components for common NLP tasks. OpenNLP is extensible and is written for easy integration with other libraries and has a good API for communicating with existing code-base. However OpenNLP is computationally more demanding than CoreNLP and therefore takes more time to perform common tasks [12].

2.3 Processing Pipelines

A processing pipeline can be defined as a set of pre-defined procedures to incrementally build and transform data towards a desired output. For example, in Figure 8, it can be observed that to generate NER labels for a given text, the text needs to be tokenised, then enriched with Part-Of-Speech (POS) tags and finally processed through shallow parsing. These single steps constitute a chain of computational processes which form an NLP pipeline. In our experiments, we employed data parallelism through respective distributed frameworks to distribute and execute these pipelines across resources within a physical cluster. In this approach, we are able to improve performance by distributing the processing tasks across resources on remotely located machines.

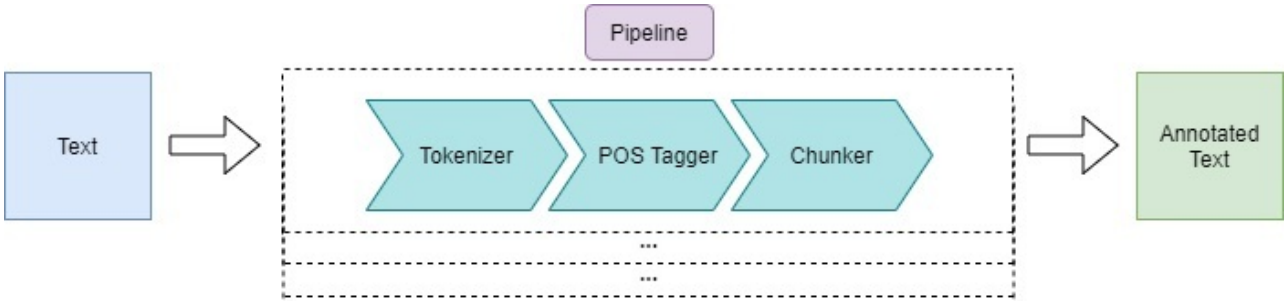

Figure 8: NER Text Processing Pipeline

In the following subsection, we discuss each of the distributed frameworks employed in this deliverable and contrast their unique features and capabilities.

[1] https://nlp.stanford.edu/software/
[2] https://opennlp.apache.org/

2.4 Parallel and Distributed Processing Frameworks

Parallel processing frameworks in recent years have provided a method for avoiding the bottleneck of scaling up resources, by allowing the scaling out of processes via simultaneous execution in parallel. This is primarily done using computer clusters, i.e. groups of computers working in tandem to breakdown computations in parallel. Distributed frameworks are the software technology which facilitates this strategy, and although abstractions and techniques may differ between solutions, the frameworks tend to align in their approach of using parallelisation of computer code and nodes to drive down computational costs. In our previous deliverable, D5.4 we discussed several frameworks which provided such functionality and in the following subsections we discuss each in detail before providing a comparison of their shared features.

2.4.1 Apache UIMA DUCC

Distributed UIMA Cluster Computing (DUCC) [3] is a Linux cluster controller designed to scale out UIMA pipelines for high throughput collection processing jobs, as well as for low latency real-time applications [5]. DUCC is built on top of UIMA Asynchronous Scaleout (UIMA-AS), a scalability replacement for the UIMA Collection Processing Manager (CPM) that allows UIMA pipelines to be run across a collection of processing nodes. DUCC is designed to deal with complex UIMA pipelines with large memory requirements by distributed execution on multiple threads and nodes. The major components of DUCC include:

- **Orchestrator (OR)**: provides the essential operational functionalities of the system.
- **Resource Manager (RM)**: allocates the constrained resources amongst user requests on a *fair-share* policy.
- **Services Manager (SM)**: facilitates the control and monitoring of services.
- **Agents**: one-per-node, responsible for deploying, monitoring and controlling processing on each node respectively.
- **JobDriver (JD)**: Manage workitem from the CollectionReader to the analytical pipeline and report performance statistics.
- **Process Manager (PM)**: monitor and control processes throughout the cluster.

DUCC's Job Driver identifies pipelines that can run in parallel and executes them on different processing nodes, as shown in Figure 10. Similarly to Flink, DUCC manages errors in the pipeline, hence, if part of the process fails to execute, the whole process does not have to be repeated.

2.4.2 Apache Spark

Spark[5] is a general-purpose framework which specialises in large-scale batch processing. Although it was recently extended in terms of stream processing functionality, it was not strategically designed to handle large volumes of live data, environments in which memory is sparse or situations in which there is low tolerance for latency requirements [13]. Spark's unique offering is that it's core abstraction Resilient Distributed Datasets (RDD) is fault tolerant and this abstraction conducts in-memory computing which avoids traditional I/O bottlenecks of relational systems. Spark has become hugely popular among academic and business circles [14] primarily because it offers a simple, efficient and mature solution to big data processing.

[3] https://uima.apache.org/doc-uimaducc-whatitam.html
[4] https://uima.apache.org/d/uima-ducc-1.0.0/duccbook.html#x1-24800015.2
[5] https://spark.apache.org/

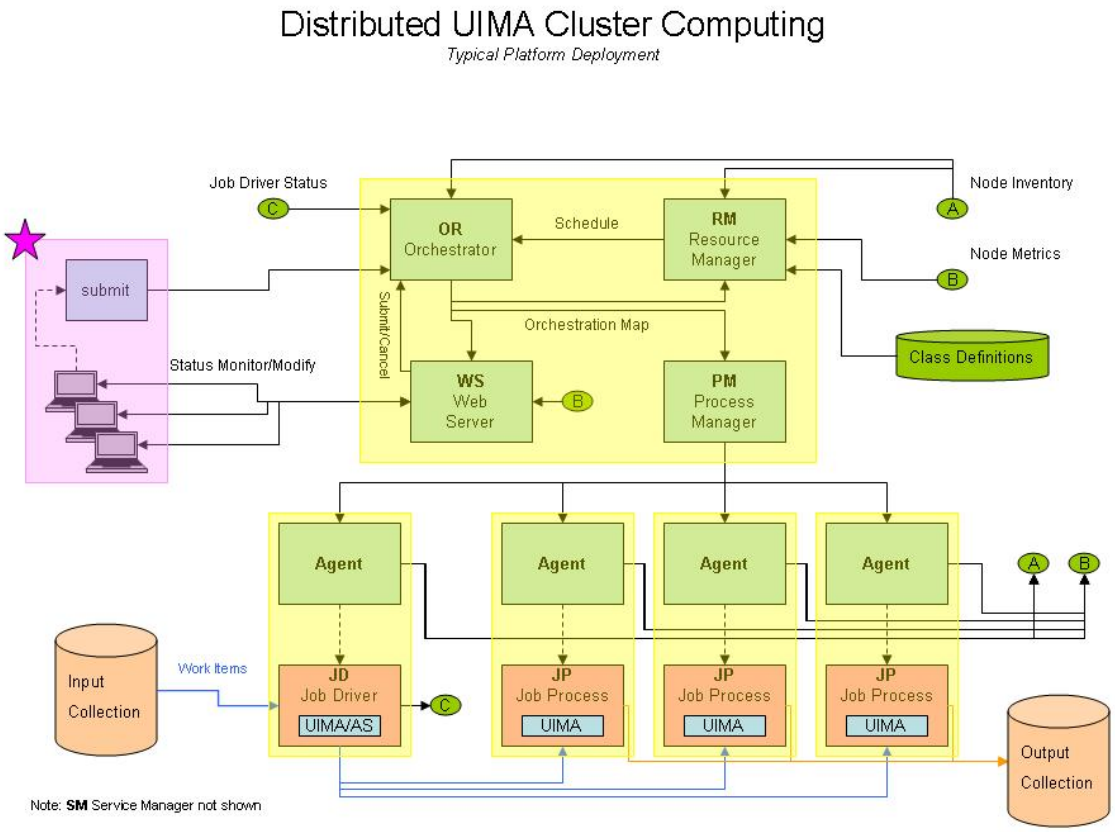

Figure 9: Apache UIMA DUCC Platform Deployment[4]

Spark follows a master-slave architecture, shown in Figure 11, which involves a driver node that orchestrates jobs through an application-level Spark context. Each context is handled by multiple worker nodes, who have the single purpose of being job executors. When a Spark application is submitted to the driver, it is internally converted into a Directed Acyclic Graph (DAG), which is then converted into a physical execution plan consisting of staged tasks for individual workers to execute. On completion of tasks, the driver is notified and a Spark application is successfully processed.

2.4.3 Apache Flink

Flink[6] is a leading open-source solution for parallel processing. It excels at processing both bound, i.e., fixed, and unbound data sets within contexts which require low data latency and high fault tolerance [15]. Flink's unique offering is it's in-memory performance combined with an internal program optimiser which improves processing performance. Flink uses a DataSet and DataStream API to implement applications for processing and these are represented as bound and unbound datasets, respectively. Flink also follows a master-slave architecture shown in Figure 12, with JobManagers controlling jobs and TaskManagers processing them. Applications submitted to Flink are represented as streaming dataflows and consist of streams and transformation

[6]https://flink.apache.org/

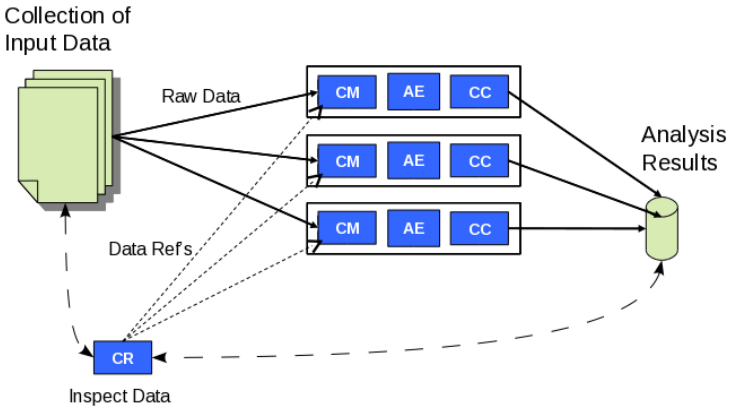

Figure 10: Apache UIMA DUCC Collection Processing Job Model

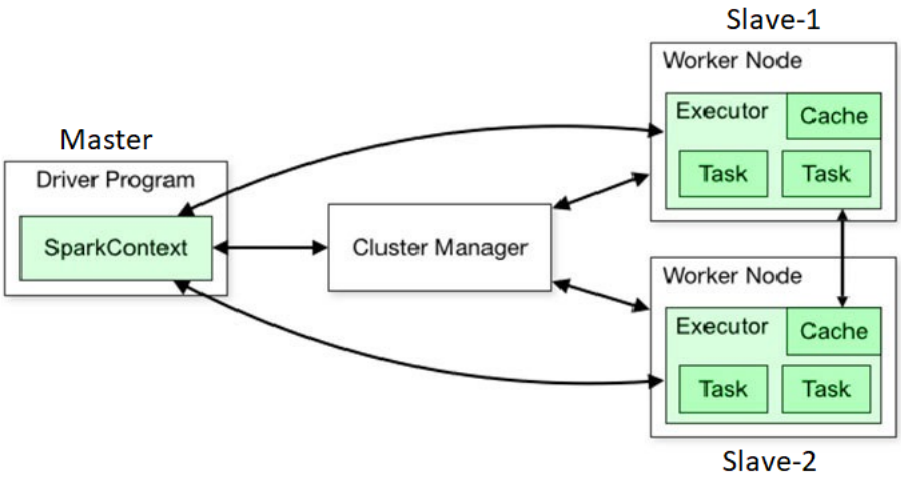

Figure 11: Apache Spark Master-Slave Architecture

operations. The streams represent an intermediate result, while transformations take a stream and apply operations, which change one DataStream into another. Each dataflow may have one to many sources, a number of desired transformations and one to many sinks. Dataflows at runtime are represented similarly to DAGs within Spark, however Flink automatically reconfigures streaming dataflows to best utilise the resources available from TaskManagers within the cluster, reducing management overhead on the part of the designer [16].

2.5 Natural Language Processing in Typhon

With ubiquitous and growing storage of polystores within the TYPHON ecosystem, it is necessary to enrich such data with annotations based on the structure and meaning of documents stored, as this augments developer actions and expands empirical inference. Data is a source of learning from the past, and as TYPHON provides scalable storage in heterogeneous environments, it makes sense to offer a dedicated NLP facility that enhances its analytical capabilities. Users of TYPHON can then request or act upon extracted knowledge to be better informed about document characteristics, enabling greater strategic intuition [17]. Moreover, the abil-

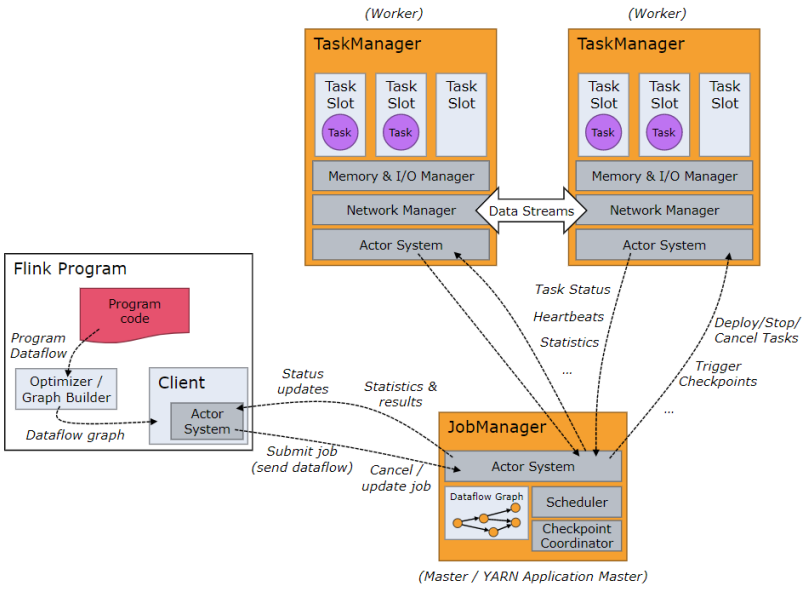

Figure 12: Apache Flink Master-Slave Architecture

ity to scale up NLP text analysis to meet expanding volumes of data is a significant research area and field of experimental challenge [2]. A traditional linear NLP structure would be inadequate for what TYPHON has envisioned, as this would result in bottlenecks and poor performance throughput. Hence, we have focused on DTPFs as architectures which can consistently mitigate computational costs and increase throughput.

The NLAE outlined within this deliverable is a tailored solution for this purpose. It integrates into TYPHON as an endpoint to TYPHONQL as shown in Figure 13. The NLAE receives queries and requests, processing these internally in parallel as NLP tasks and outputs the result to an ElasticSearch sink. Therefore, the NLAE functions as an on-demand NLP gateway within TYPHON without exposing NLP abstraction to unnecessary sources. Furthermore, the NLAE integrates into the TYPHON type system by performing a lookup of the text modelling tasks currently supported. In this way, the NLAE is able to transform a requested query into an annotated result, with the concealed benefit of parallelism. In the following section, we explain how we compared different distribution frameworks and the structure we used to test the performance of DTPFs for suitability within the NLAE.

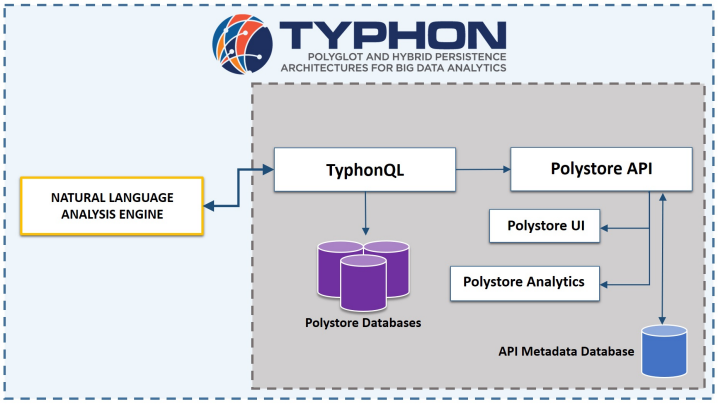

Figure 13: Natural Language Analysis Engine in Typhon

3 Methodology

In this section we outline the methodology that was adopted to evaluate the different Distributed Text Processing Frameworks (DTPF). We provide information about the datasets that were used for our experimentation and how these were mapped in our Elasticsearch server. Next we give details of the physical cluster that was used to run these experiments and outline the different pipelines that were developed for the experimental and partner use case scenarios. Finally we provide insights to the evaluation criteria that were used for the comparative analysis of the different DTPFs.

3.1 Datasets

For our evaluations, we have run experiments for the different pipelines that we designed on publicly available datasets. The datasets that have been used for evaluating the DTPFs are:

- Amazon Product Reviews Dataset
- CoNLL-2003 - (English) Named Entity Recognition Dataset
- Custom Weather Named Entity Recognition Dataset

3.1.1 Amazon Product Reviews Dataset

The Amazon Product Reviews dataset consists of 142.8 million user reviews from May 1996 until July 2014 [6]. The full dataset contains some duplicate reviews with a total size of 20GB. Smaller subsets of the dataset are available for experimentation purposes and have been divided into reviews of products "per-category". For example, the *reviews_Books_5* dataset containing user reviews on books bought from Amazon consists of 8,898,041 reviews with a compressed size of 3GB and uncompressed size of 8.9GB. The uncompressed json file consists of one-review-per-line with the following format:

```
{
   "reviewerID": "A2SUAM1J3GNN3B",
   "asin": "0000013714",
   "reviewerName": "J. McDonald",
   "helpful": [2, 3],
   "reviewText": "I bought this for my husband who plays the piano.  He is
      having a wonderful time playing these old hymns.  The music  is at times
      hard to read because we think the book was published for singing from
      more than playing from.  Great purchase though!",
   "overall": 5.0,
   "summary": "Heavenly Highway Hymns",
   "unixReviewTime": 1252800000,
   "reviewTime": "09 13, 2009"
}
```

where:

- **reviewerID** is the ID of the reviewer, e.g. A2SUAM1J3GNN3B
- **asin** is the ID of the product, e.g. 0000013714
- **reviewerName** is the name of the reviewer
- **helpful** is a helpfulness rating of the review, e.g. 2/3

- **reviewText** is the text of the review
- **overall** is the rating of the product
- **summary** is a summary of the review
- **unixReviewTime** is the time of the review (unix time)
- **reviewTime** is the time of the review (raw)

3.1.2 CoNLL-2003 - Named Entity Recognition Dataset

The CoNLL-2003 Named Entity Recognition dataset for English language consists of a total of 1393 news articles with 219,553 sentences and 301,418 tokens [7]. The dataset uses the Inside–outside–beginning (IOB) tagging scheme with one word per line along with its corresponding Part-of-Speech (PoS) tag and empty lines representing sentence boundaries. The dataset has been used for the training, development and testing of NER taggers. Figure 14 shows an example sentence from the dataset:

U.N.	NNP	I-NP	I-ORG
official	NN	I-NP	O
Ekeus	NNP	I-NP	I-PER
heads	VBZ	I-VP	O
for	IN	I-PP	O
Baghdad	NNP	I-NP	I-LOC
.	.	O	O

Figure 14: CoNLL-2003 NER Dataset

A custom script was developed to read data from the original dataset and format it in the schema required by the Elasticsearch mapping. Since the original dataset had a small number of articles, entries were duplicated to increase the size of the resulting dataset. As the goal of our experiments was not to test the accuracy of the NLP components, but the effect of data size on the distributed processing pipelines, data duplication technique was acceptable. Initially sentence boundaries were used as markers for individual data objects, i.e. one sentence per entry in the Elasticsearch index. During the duplication process, N-number of sentences, where $N = 3, \ldots, 10$ were concatenated to create longer text entries.

3.1.3 Custom Weather Named Entity Recognition Dataset

The third dataset used in our experiments was generated in-house by collecting weather emails from the Meteosafe website[7]. Meteosafe provides periodic emails about the current weather conditions in an area of interest. Data was collected by registering to the service and receiving weather updates every 15-minutes for two locations. Text from a total number of 8,640 emails was used to generate the dataset. As with the CoNLL-2003 dataset, the emails dataset was augmented with data duplication to increase the size of the dataset for our experiments. Figure 15 shows a sample email from the weather dataset.

3.2 Use case Scenarios

To evaluate the performance of the Distributed Text Processing Frameworks (DTPFs), our evaluation framework consisted of the following use case scenarios.

[7] https://meteosafe.com/

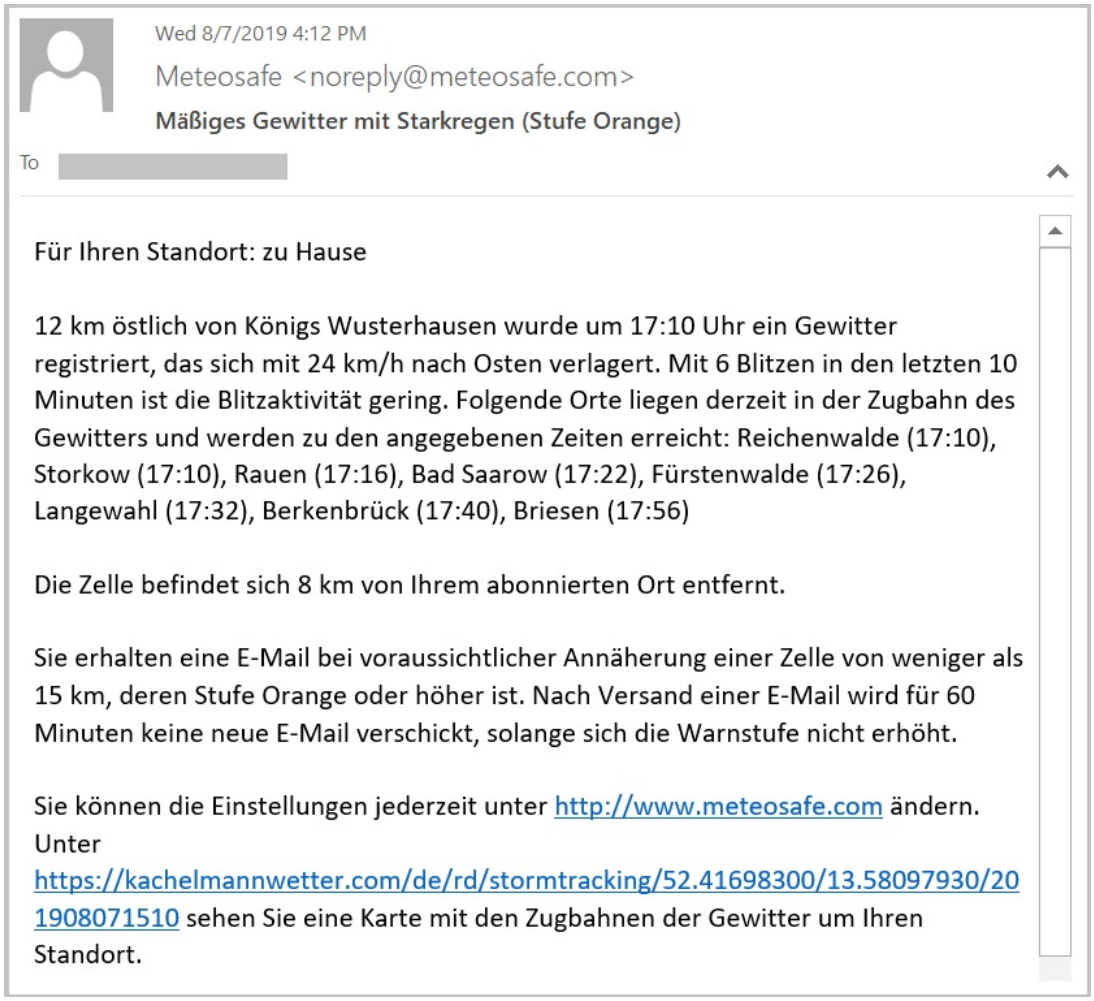

Figure 15: Meteosafe Weather Email

- Experimental Use Cases
- Partner Use Cases

3.2.1 Experimental Use Cases

To evaluate the performance of the DTPFs, separate NLP pipelines were designed for each of the frameworks and compared to a Java baseline. This baseline replicated the NLP source code of other tests, however implemented no parallelism within its environment, therefore was our control measurement in experiments. We utilised the following use cases for testing NLP pipelines within each of the frameworks:

Sentiment Analysis for Amazon Product Reviews Dataset Sentiment Analysis is an NLP task that aims to identify affective states, polarity or subjective opinion of actors from the context of natural language. We used the Stanford CoreNLP Sentiment Analysis classifier to classify reviews in the dataset. The pipelines for sentiment analysis for all the DTPFs consisted of a tokeniser, that tokenises the input into individual tokens, the sentence splitter, that splits inputs based on the sentence boundaries, the parser, that parses the tokens per sentence to generate the sentiment of the sentence, as shown in Figure 16. The CoreNLP sentiment analyser

only generates the sentiment for individual sentences, hence heuristics are applied to get the overall sentiment of a paragraph.

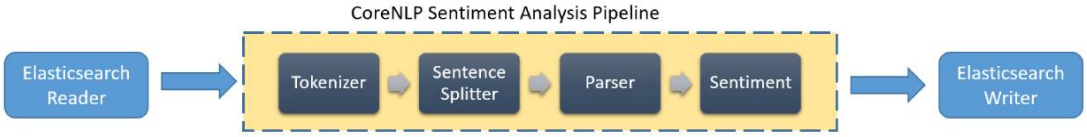

Figure 16: Sentiment Analysis Pipeline for Amazon Dataset

Named Entity Recognition for the CoNLL-2003 NER Dataset Named Entity Recognition is the NLP task of tagging annotations of named entities on a given document. In the Stanford CoreNLP toolkit, the pipeline for the NER Tagger tokenises the input into individual tokens, then splits the input text by sentence boundaries and then assigns a Part-Of-Speech (POS) tag to each token. These tokens are then lemmatised and finally the corresponding NER tag is allocated to each token as shown in Figure 17.

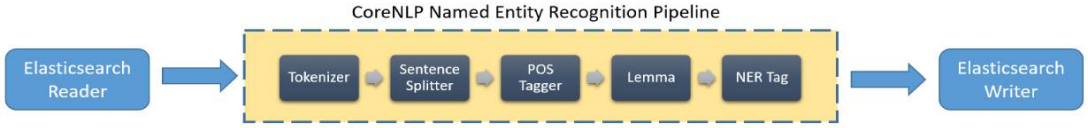

Figure 17: NER Pipeline for CoNLL-2003 NER Dataset

Co-reference Resolution for the CoNLL-2003 NER Dataset Co-reference resolution aims to identify all expressions that refer to shared entities within a document. To perform Co-reference resolution, the CoreNLP library tokenises the input, splits the input into sentences and adds corresponding POS tags to each token. It then performs lemmatisation and generates a NER tag for each token. The tokens are then parsed to generate their dependencies, which are then used to calculate co-reference. The pipeline for co-reference resolution is shown in Figure 18

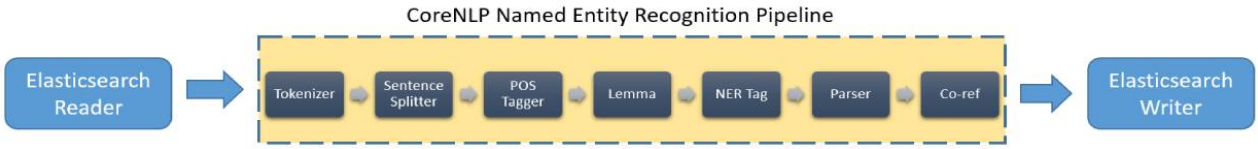

Figure 18: Co-Reference Resolution Pipeline for CoNLL-2003 NER Dataset

3.2.2 Partners Use Cases

A custom Weather Entity Tagger was designed for the project partners Volkswagen and ATB during the course of the project. The pipeline to evaluate the performance of the tagger with respect to the DTPFs included a CoreNLP standalone Tokeniser, to tokenise the input, a CoreNLP sentence splitter, to split input sentences, and a CoreNLP Custom-Trained NER Tagger to tag weather entities in the data. The pipeline is shown in Figure 19

Figure 19: NER Pipeline for Custom Weather Emails Dataset

3.3 Experimental Settings

The following subsections present the experimental and evaluation settings that were used for experimentation in our evaluation framework.

3.3.1 Experimental Setup

In this subsection we discuss the hardware configuration of the cluster used for experimentation, the preparation steps that allowed to experiment with data of various sizes and the effect of the cluster size.

3.3.1.1 Cluster Configuration Each of the use case pipelines have been executed on a standalone cluster which consists of one Master-Node and three Slave-Nodes. as shown in Figure 20. The specification of each system in the cluster is shown in Table 1. Slave-Node1 and Slave-Node2 were configured to run as slaves/worker nodes for all the DTPFs, while Slave-Node3 was configured to run as the Elasticsearch server.

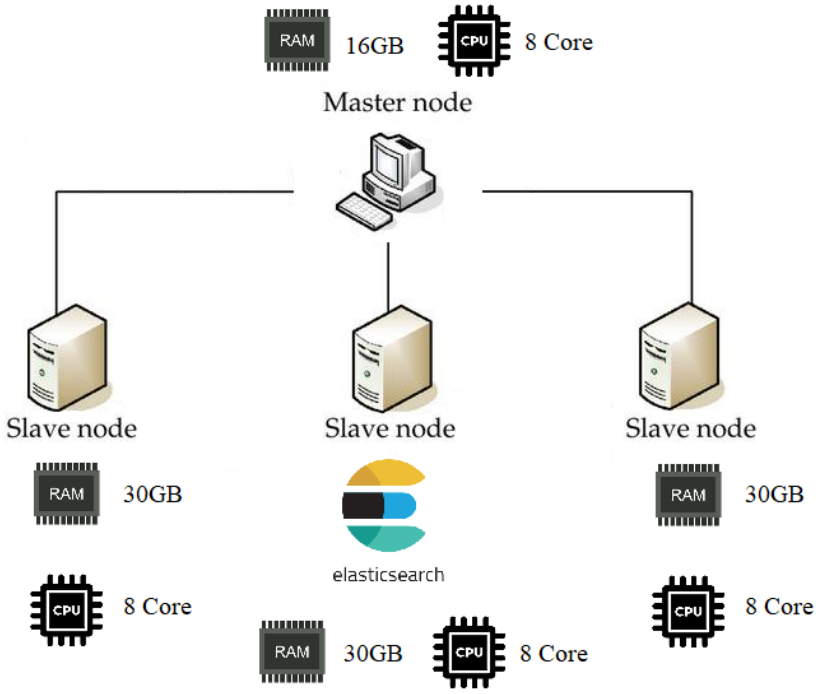

Figure 20: Experiment Cluster Setup

This was done to ensure that the I/O operations to the Elasticsearch have minimum processing overhead. All the DTPFs were installed on the cluster with default configuration parameters specified in their documentation.

Table 1: Machine Specifications for the Cluster

Node	CPU	Cores	RAM (GB)	OS
Master	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz	8	16	Ubuntu 16.04 Xenial
Slave-Node1	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz	8	32	Ubuntu 16.04 Xenial
Slave-Node2	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz	8	32	Ubuntu 16.04 Xenial
Slave-Node3	Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz	8	32	Ubuntu 16.04 Xenial

3.3.1.2 Data Pre-processing

As discussed in section 3.1, three different datasets were used in the evaluation of the DTPFs. To simulate 'big-data' datasets we duplicated instances within each dataset, augmenting their size. We did this as our goal is not to measure performance metrics associated with the individual components, but the performance of the DTPFs on distributed pipelines. To create a more realistic scenario we indexed documents using the Elasticsearch cluster.

3.3.1.3 NLP Toolkit

All pipelines designed for evaluation used the pre-trained models available for the Stanford CoreNLP library. Stanford CoreNLP is one of the leading Open Source projects that provides a number of language analysis tools written in Java under the GNU General Public License [8]. Apart from providing tools that work out-of-the-box, Stanford CoreNLP also provides pre-trained models for common NLP tasks for seven different languages. The following models were used for the NLP tasks in our experimental pipelines:

Table 2: NLP Toolkit Specifications

Task	Classifier	Model Specification
Sentiment Analysis	Recursive Neural Tensor Network	pre-packaged-model
Named-Entity Recognition	Conditional Random Field	english.conll.4class.distsim.crf.ser.gz
Co-reference Resolution	Deterministic	pre-packaged-model
Custom Named-Entity Recognition	Conditional Random Field	ATB-ner-model-de.ser-V5.gz

3.3.2 Evaluation Settings

As described in the previous sections, the focus of our evaluation was the performance of the Distributed Text Processing Frameworks on similar NLP tasks. To ensure constant times in the I/O operations the standard Elasticsearch Java API [9] was used to read and write data from Elasticsearch. All pipelines used the same Stanford CoreNLP classifiers and models to process the text. We evaluated the performance of the DTPFs on three different metrics, i.e. speed, data size and throughput. The evaluation settings used to benchmark the DTPFs for our experiments are explained in Table 3.

[8] http://www.gnu.org/licenses/gpl-2.0.html
[9] https://www.elastic.co/guide/en/elasticsearch/reference/6.8/api-java.html

Table 3: Evaluation Settings

Unit	**Measure**
Speed	Time in *minutes* required to complete tasks on different dataset sizes
Data size	N is the number of documents, where $N = 1000, \ldots, 100000$
Throughput	Documents *successfully* processed per *minute*

3.3.3 Controlled Permutations

We tested the pipelines with different sample sizes and cluster sizes (1-slave node and 2-slave node settings) to quantify the effect of these changes. Each pipeline was run five times and the average throughput was used as our marker against the baseline. The adaptation of cluster size, given our small cluster, suggested that throughput grew proportional on the number of slaves within the cluster for both Spark and Flink. Interestingly, this was not observed for UIMA DUCC for light-weight pipelines, such as NER. This was because the time required to annotate each item was very short, hence a bottleneck was created by the JobDriver that passed too many items causing a deadlock. For Sentiment Analysis, which is a more complex task, UIMA DUCC showed similar traits as Spark and Flink, and the job performance for a 2-node cluster was better than that of 1-node cluster.

4 Results

In this section, we discuss the evaluation benchmarks resulting from NLP pipelines processed through the Distributed Text Processing Frameworks (DTPF) on our experimental datasets. We additionally describe and identify the factors on which these frameworks differ in practice.

4.1 Experiment Evaluation

4.1.1 Speed

As specified in section 3.3.2, we evaluated the DTPFs from three perspectives; speed, size of data and throughput. Each pipeline was designed to run an NLP pipeline on the chosen dataset with the underlying DTPF managing the distribution of the pipeline across the cluster. To create a baseline benchmark, we ran the pipelines as a traditional linear Java Application on a single machine. For a sample of 100,000 documents, it can be observed in Table 4 that the baseline benchmark for speed was considerably high across all three NLP tasks. The distributed frameworks increased throughput by several orders of magnitude, most notably in the case of the computationally expensive Co-reference and Sentiment Analysis tasks.

Table 4: Speed: Time taken to complete 100k documents

NLP Task	Java	Spark	Flink	UIMA DUCC
Named Entity Recognition	380 mins	**88 mins**	**88 mins**	270 mins
Co-reference	4340 mins	672 mins	**624 mins**	723 mins
Sentiment Analysis	4320 mins	**624 mins**	680 mins	666 mins

4.1.2 Data Size

The pipelines designed to perform the three NLP tasks were run on different data sizes to assess the effect of the size of input data on the performance of the DTPF. Different workloads ranging from 1000 documents to 100k documents were experimented with to find if the performance of the underlying DTPF was correlated with the size of input data. In almost all the cases no significant correlation was seen between processing rate and data size for any of the DTPFs except for UIMA DUCC. In case of UIMA DUCC, the size of data appeared to have a considerable effect on the initialisation time of the application.

4.1.3 Throughput

The last evaluation metric of our framework was throughput. Throughput is the rate at which each DTPF processed the data. To this effect, throughput was directly influenced by the type of NLP task being performed. For computationally expensive tasks, such as Sentiment analysis and Co-reference resolution, the throughput of the frameworks was adversely affected. This showed a direct correlation between the NLP component and the throughput of the framework.

Table 5: Throughput: Documents processed per minute

	Throughput: Docs/Minute		
NLP Task	***Spark***	***Flink***	***UIMA DUCC***
Named Entity Recognition	**1136**	**1136**	370
Co-reference	148	**160**	143
Sentiment Analysis	**152**	148	149

4.2 Distributed Text Processing Framework Design Evaluation

In this section, we examine the structural differences among distributed frameworks and discuss how they affect development using each respective technology. Table 6 presents some of the core differences between the frameworks examined in this deliverable. The main abstraction in Spark is Resilient Distributed Datasets (RDDs) which for all purposes are distributed collections of data which are partitioned across nodes within the cluster and actioned in parallel. For Flink, the primary abstractions are DataSet and DataStream which are used for distributed batch and streaming applications, respectively. For UIMA DUCC, the Common Analysis Structure (CAS) is the central data structure. UIMA provides a native Java interface to CAS, called JCas, that can represent features as Java objects. Generating a CAS object is a computationally expensive task which adversely affects the performance of DUCC in small-sized clusters.

Table 6: Comparison of Distributed Frameworks

	Spark	**Flink**	**DUCC**
Data Model	RDD	DataStream	Common Analysis Structure (CAS)
Computation Model	Micro-batch	Streams	Collection
Memory Optimisation	Manual	Automatic	Manual

Although from an implementation point-of-view Spark and Flink frameworks are actioned similarly (through transformers/operators such as map and reduce), their computational models differ considerably. Spark utilises a micro-batch processing model which combines the advantages of executing on large volumes of data at rest, with a continuous flow operator which segments data to a series of micro-batches. Flink conversely adopts a operator-based continuous flow model which executes data as it arrives. This fundamental difference means that the context in which data is supplied for processing matters and also indicates from our experiments that both frameworks perform sufficiently in the context of batch processes.

The last difference between the frameworks we identified was how internal memory was optimised. In Spark, memory is configured, meaning that the application needed to be manually optimised given the context of resources within the cluster. Flink goes a step further by employing built-in program optimisations to streamline the execution of jobs and smooth spikes that can potentially occur. UIMA DUCC on the other hand uses the job configuration settings to ensure that each application receives the exact amount of memory it needs.

It is important to mention here that we were limited in our capability to perform task parallelism as the inherent design structure of the Stanford CoreNLP toolkit meant that each node needed to construct an annotation pipeline rather than having this serialised and shared (via master) through the network. Moreover the procedure of loading pipelines with properties were fundamentally serial processes, which compromised the ability to break the pipeline into more granular processes. This correspondingly decreased throughput on each slave node by duplicating and serialising computational efforts. Furthermore, the Stanford CoreNLP toolkit faces

compatibility issues within serialised environments (i.e., edu.stanford.nlp.pipeline.StanfordCoreNLP not implementing a serialisable interface), which directly impacts its performance in distributed environments. This, however, identifies a significant limitation to an existing software tool which can reinforce the design and direction of distributed NLP systems through selective integration of NLP toolkits which are compatible within distributed environments. In our utility of distributed frameworks for NLP processing, it was Flink which was the most reliable, transparent and compatible with the processing we required. It is our recommendation that Flink be used in circumstances similar in nature to this deliverable, due to its stability, ease of customisation and functionality that it delivers.

5 Evaluation

In this section, we examine the critical discussion points uncovered from conducting distributed research using NLP pipelines. We firstly examine the importance of serialisable classes in distributed environments (i.e., master-slave architectures), then discuss the potential performance gains by combining data and task parallelism, where possible, and lastly we discuss the comparability issues that NLP toolkits face when granularity is low.

5.1 Discussion

The following subsections highlight the research implications of our experimentation. We first describe the serialisability issues faced with Stanford CoreNLP and then discuss the perspectives of data and task parallelisation.

5.1.1 Importance of Serialisability in Distributed Frameworks

A major finding in our experiments was the bottleneck faced by one of the most widely used NLP toolkits in the literature. This was primarily due to the toolkit's Annotator object (edu.stanford.nlp.pipeline.StanfordCoreNLP) not being Serialisable, in effect not anticipating an NLP object being shared between JVM's within a distributed cluster. The implication of this design structure means that without the process of serialisation, the master portion of an application program has no ability to share an NLP object with its many workers, duplicating the efforts of many slaves to reconstruct objects which should in principle be shared. This design bottleneck directly impacts parallelism by duplicating CPU usage, as the time required for building an object is longer than the time required for deserialisation (i.e., initialising from cache). Moreover, as cluster computing is highly dependent on effective communication, core Serialisation tuning techniques, such as Kryo Serialisation, in both Spark and Flink are unavailable, reducing the ability for distributed frameworks to manage computational complexity. This underlying design choice of serialisation inherently influences research outcomes and therefore should be given strict consideration when designing NLP pipelines in distributed environments.

5.1.2 Data and Task Parallelism

While data parallelism is often achieved by scaling out a problem to simultaneous workers, task parallelism is often more difficult and contextual in highly coupled pipelines. Task parallelism seeks to identify the granularity of a job and parallelise segments of asynchronous pipeline code which can be concurrently executed in memory on the same data in order to mitigate the costs of running linear chains of code. In our experiments, we found that the Stanford CoreNLP Annotator object was not designed to be granular as the single Annotator object is made to load pipeline flags (i.e., NLP components) from a Properties object on initialisation. This means that segmenting a pipeline would mean creating n number of Annotator objects, hence incrementing computational costs. Our research indicates that NLP toolkits and pipelines more susceptible to task parallelism would offer the best of both worlds for research outcomes when combined with data parallelism. As this deliverable has shown, existing frameworks deliver significant proportional performance gains based on input size via data parallelism. The best scenario for DTPFs would then be to augment this performance gain by leveraging toolkits compatible with task parallelism, to gain proportional performance improvements on the number of independent, i.e., non-serial, tasks performed. Following this recommendation would lead research outcomes to produce maximum performance gains in distributed environments.

6 Natural Language Analysis Engine

In this section, we discuss the containerised product of our NLP workflows within the Natural Language Analysis Engine (NLAE), a standalone lookup for TYPHON which provides on-request NLP functionality. NLAE has been developed to support the internal TYPHON type-system and service requests through a REST API.

6.1 Overview

The Natural Language Analysis Engine (NLAE) is a self-contained standalone analysis engine, that integrates into the TYPHON ecosystem to provide on-demand Natural Language Processing capabilities. The NLAE interfaces with TYPHON through a REST API, which allows the end-user to send queries to *process*, *query* and *delete* data as shown in Figure 21.

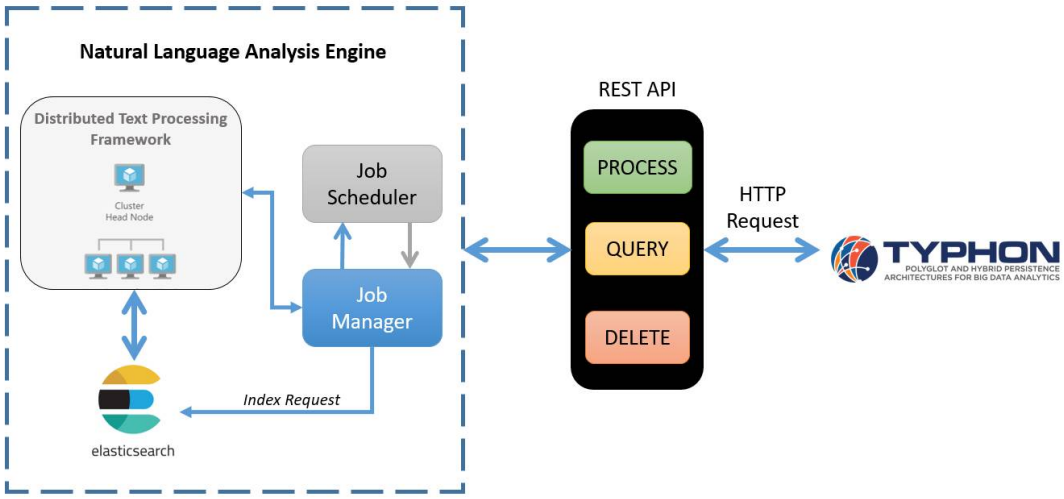

Figure 21: Natural Language Analysis Engine

Data to be processed by the NLAE is transferred from TYPHON through API requests. When a request to process data is received, NLAE maps the data to an ElasticSearch index mapping. The ElasticSearch mapping describes the schema of JSON documents which includes information about the data types of fields as well as how to index them in Lucene indexes. The index mapping was defined in consultation with the project partners to ensure a consistent schema is used for all data saved in the ElasticSearch indexes. Below we provide the mapping description that is used to create new indexes in ElasticSearch:

```
{
  "mappings" : {
    "_doc" : {
     "properties" : {
       "entityType" : {
         "type" : "text",
         "fields" : {
           "keyword" : {
             "type" : "keyword",
            }
```

```
          }
        },
        "fieldName" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
             }
          }
        },
        "id" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
             }
          }
        },
        "nlpFeatures" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
            }
          }
        },
        "text" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
            }
          }
        },
        "workflowName" : {
          "type" : "text",
          "fields" : {
            "keyword" : {
              "type" : "keyword",
            }
          }
        }
      }
    }
  }
}
```

Once data has been stored in the ElasticSearch index, the Job Scheduler schedules a job with the *workflowName* and an internal API call is sent to the Job Manager to start the job for the given ElasticSearch index. The user can send an API call to *query* the results for a job. If the job is completed, NLAE starts sending the results, otherwise a *"job in progress"* message is returned. Below is the list of NlpTaskTypes currently supported by the NLAE:

- ParagraphSegmentation
- SentenceSegmentation
- Tokenisation
- PhraseExtraction
- nGramExtraction
- POSTagging
- Lemmatisation
- Stemming
- DependencyParsing
- Chunking
- SentimentAnalysis
- TextClassification
- TopicModelling
- TermExtraction
- NamedEntityRecognition
- RelationExtraction
- CoreferenceResolution

6.2 Development Technologies

We integrated a wide range of Java-based open-source technologies in the development of NLAE. The deployment of NLAE is achieved through the orchestration of containerized microservices. We employ libraries and technologies that facilitate NLAE as a micro service, which will be discussed in the following subsections.

6.2.1 Libraries

- **Springboot (2.2.4)**: Springboot helps in resolving two main elements in the deployment of NLAE, dependencies and configurations. Springboot delivers a production-ready deployment of NLAE which saves developers from having to have prior technical knowledge of development in order to use service deployment.
- **ElasticSearch (6.8.1)**: ElasticSearch is a distributed full-text search engine based on the Lucene library. ElasticSearch supports multi-tenancy (i.e., is a software which is able to serve many users) and provides real-time analytics search, which is important in the context of NLAE that needs to scale to large volumes of requests. Its integration into NLAE has been realised through a Java REST API used by NLP workflows to perform datastore operations and persistently store annotated entities.
- **JacksonXML (2.10.3)**: JacksonXML is a serialisation technology based on the Jackson library which assists in the deployment of NLAE. JacksonXML integrates with Springboot and supports NLAE by providing marshalling/unmarshalling functionality in scenarios where XML processing is required.
- **Swagger (2.9.2)**: Swagger is an open-source framework which supports the construction of APIs within the deployment of NLAE. Swagger automatically identifies annotations within developer code and builds RESTful web services with documentation, in order to support the deployment and access of API requests for developers.
- **Spring-Fox (2.9.2)**: Spring-Fox is an open-source framework in the Spring ecosystem which automates JSON API documentation. We utilized Spring-Fox as it naively integrated with other libraries in NLAE and reduced the efforts on the part of the developer.

6.2.2 Technologies

- **REST API** RESTful web services are a key technology within NLAE. REST APIs allow for complex requests to be made simple using a uniform middleware, internet protocols and web resources. We use REST APIs to expose the primary functions of NLAE via endpoints which then allow for easy developer access across a network. REST APIs have become considerably standardised and as such we developed a REST API to interact with NLAE for NLP functionality.
- **Docker** Docker is a deployment technology which separates an application from the hardware setup on which it operates. We choose Docker as it was important for NLAE to function in production environments that may not be familiar or a priori known. Docker packages NLAE within a container which then grants autonomy from system-specific issues that may arise under production or development. Additionally, containerization means that multiple instances of services can be run at the same time, even on virtual machines.
- **Kubernetes** Kubernetes is an open-source deployment platform for managing containerized services and helps in orchestrating applications automatically. Kubernetes is particularly helpful in managing a cluster, grouping and scaling containerized services. Apache Flink provides official Docker images on Docker Hub [10] which can be used to deploy a Session or Job cluster in a containerized environment such as Kubernetes. To deploy a Flink cluster on Kubernetes, Flink provides a set of (common resource definitions) [11] to launch the common cluster components using the *kubectl* command.
- **Open API** Open API is an API specification format for REST APIs. It allows NLAE to specify its endpoints, define input/output operation parameters, specify authentication rules and outline licensing information.
- **Asynchronous Communication** NLAE utilises asynchronous network interactions to avoid the drawback from synchronous blocking. All *indexing* and *search* requests to Elasticsearch are made using the Asynchronous REST client. In this way, NLAE can expose resources to allow more non-blocking requests to be made concurrently, decreasing the overhead on individual services.

6.3 Description of Components

This section describes the components of the Natural Language Analytics Engine (NLAE) and explains how these components interact with each other to store data, assign processing jobs and return results to TYPHON. As seen in Figure 21, the NLAE consists of three main components with an external REST API that services TYPHON requests. To this effect the NLAE is seen as a black box for the remaining of the TYPHON ecosystem, with the REST API exposing endpoints that are consumed by TYPHON.

6.3.1 External Facing NLAE REST API

The externally facing NLAE REST API provides endpoints that are consumed by TYPHON to send data, request job processing and retrieve data from the NLAE. The NLAE REST API exposes the following three endpoints:

- **processText**
- **queryTextAnalytics**
- **deleteDocument**

[10] https://hub.docker.com/_/flink

[11] https://ci.apache.org/projects/flink/flink-docs-stable/ops/deployment/kubernetes.html#common-cluster-resource-definitions

6.3.1.1 processText The processText endpoint receives POST requests containing individual entities from TYPHON to prepare them for text processing. An entity is a TyphonQL object that consists of:

- **entityType**: can be any object such as a *person*, a *product* or a *review*
- **fieldName**: the column name from the Typhon polystore where the object is saved
- **id**: the unique identifier allocated to the data object in the polystore
- **text**: the textual data that needs to be analyzed
- **nlpFeatures**: the NLP task(s), such as *Sentiment Analysis* or *Name Entity Recognition*, that need to be performed on the textual data.
- **workflowName**: the name of the specific workflow that will be run to generate the processed result

The endpoint receives the request as a JSON object with the following model specification:

```
{
   "entityType": "type : String",
   "fieldName": "type : String",
   "id": "type : String",
   "text": "type : String",
   "nlpFeatures": "type : [String]",
   "workflowName": "type : [String]"
}
```

When a request is received, the *process* method performs validation checks on the JSON object to ensure that the required properties are provided and supported by the TYPHON TypeSystem. Once the request passes all the checks, it is forwarded to the *Messaging Queue (MQ) Producer* to be staged in the *Indexing Queue*.

6.3.1.2 queryTextAnalytics The queryTextAnalytics endpoint receives POST requests containing an *entityType*, a *fieldName* and an *nlpExpression* to retrieve *processed* entities from the ElasticSearch engine. The *entityType* and *fieldName* are the object attributes specified in the TyphonOL model, whereas the *nlpExpression* contains an SQL *WHERE* clause in a string format such as *"WHERE sentiment = 'positive'"*. The entityType, the fieldName and the nlpExpression are used to build an ElasticSearch *SearchRequest Query* that returns *SearchHits*, if corresponding matches are found in the ElasticSearch index. If there are not any results found, a message *"No results for the requested resource have been processed"* is returned. The *query.getResult()* method breaks down the nlpExpression to get *fields* of interest in the Elasticsearch index. To retrieve the documents an Elasticsearch Java Search API query is constructed as follows:

```
BoolQueryBuilder boolQuery = new BoolQueryBuilder();
boolQuery.must(new MatchQueryBuilder("entityType", query.getEntityType()));
boolQuery.must(new MatchQueryBuilder("fieldName", query.getFieldName()));
boolQuery.must(new MatchQueryBuilder("result", query.getResult()));

// Search Source
SearchSourceBuilder searchSourceBuilder = new SearchSourceBuilder();
searchSourceBuilder.query(boolQuery);

// Search Request
SearchRequest searchRequest = new SearchRequest("typhon_jobs");
searchRequest.source(searchSourceBuilder);
SearchResponse response = null;

try {
```

```
        response = client.search(searchRequest);
} catch (IOException e) {
        e.printStackTrace();
}

SearchHits hits = response.getHits();
for (SearchHit x : hits) {
    \\ Handle hits
}
```

In the case that the search does not return any results, it is assumed that the entityType in question has *not yet been processed*, and a corresponding message is returned. A request to the queryTextAnalytics endpoint contains a JSON object with the following model specifications:

```
{
   "entityType": "type : String",
   "fieldName": "type : String",
   "nlpExpression": "type : NlpExpression"
}
```

6.3.1.3 deleteDocument The deleteDocument endpoint receives POST requests containing a specific *id* for an individual document. This id is used to delete the document from the ElasticSearch engine. The Delete method uses the *prepareDelete* method of the *Delete API* provided by ElasticSearch Java API. The prepareDelete method deletes a JSON document from an Elasticsearch index with the specific *id* value provided. For example, the following Java code deletes the JSON document from an index called typhon_jobs, under a type called _job, with id valued 1:

```
DeleteResponse response =
                client.prepareDelete("typhon_jobs", "_job", "1").get();
```

The JSON object for the delete request has the following model specifications:

```
{
  "id": "type : String"
}
```

6.3.2 Internal Components

Internally, the Natural Language Analysis Engine (NLAE) consists of a number of components that interact with each other to schedule/run jobs and provide processing results. Figure 22 shows the major internal components of the NLAE which are further discussed in the following sub-sections.

6.3.2.1 Job Manager The job manager is implemented using a *Messaging Queue*, with queues for *Staging* data to be indexed and *Scheduling* jobs to be run once data ingestion has completed. The Job Manager creates an *Exchange*, which is a message routing agent. The exchange is responsible for *routing the messages to different queues with the help of bindings and routing keys*. Exchanges ensure integrity by ensuring messages are delivered to only the queues they are bound to. Once the exchange and queues are set up, the Job Manager initiates the *Consumers* for each queue. The REST API acts as a *Producer* and sends messages to

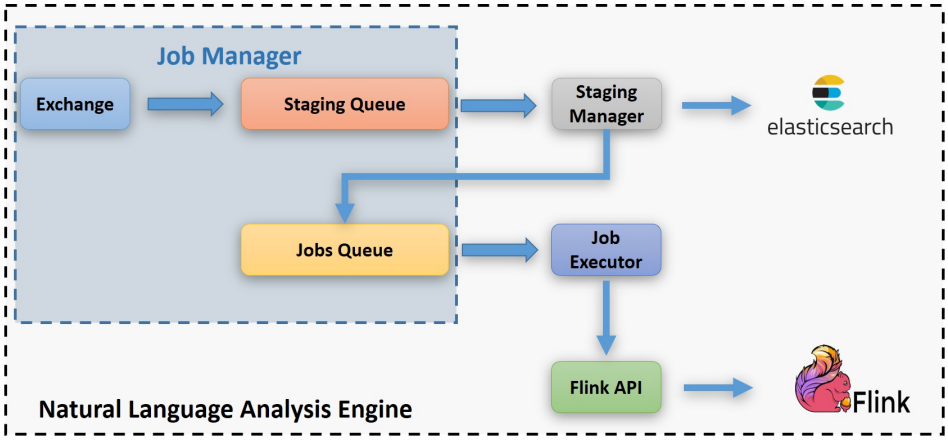

Figure 22: NLAE Internal Components

the Exchange, which routes them to the corresponding queues. Consumers act as background services which listen for messages on their respective queues and handle the payload when a message is received.

6.3.2.2 Staging Manager The Staging Manager is a *Consumer* bound to the *Staging Queue*. When this Consumer is initiated, it establishes a connection to the ElasticSearch engine using a *High Level REST Client* (HLRC). As messages (documents) arrive in the Staging Queue, the Consumer sends asynchronous requests to the HLRC to index the documents in the ElasticSearch index. Once all messages in the Staging Queue have been consumed, the Consumer initiates a *Job Scheduling* request and publishes a message to the *Jobs Queue*. In doing so, the Consumer also acts as a *Producer*.

6.3.2.3 Job Executor The Job Executor is a *Consumer* bound to the *Jobs Queue*. As soon as a message arrives in the queue, the Job Executor consumes it, locates the corresponding resources and sends an HTTP request to the *Flink Monitoring API* to submit a job to the Flink framework.

6.3.2.4 Flink Monitoring API Apache Flink provides a *Monitoring REST API* which can be used to upload jars, submit jobs, query the status of jobs and collect statistics about jobs that are running or have been completed. The Flink Monitoring API is a RESTful API that accepts HTTP requests and returns JSON responses.

6.4 Deployment and Scalability

To simplify and automate the deployment of the Natural Language Analysis Engine (NLAE), it is envisioned to be deployed as a Dockerised container. Docker [12] containers are an effective way of packaging software in a way that is predictable and consistent. Containers allow code to be packaged along with its dependencies so that applications can be deployed consistently and effectively on any physical or virtual host capable of running Docker. Since the internal components of the NLAE are loosely coupled, they can be deployed as separate containers which allows for separation of concerns between components, such as the ElasticSearch

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

engine and the Distributed Text Processing Framework. Docker Containers allow applications to scale, however managing such operations at scale becomes complicated. Operating at scale requires inter-container as well as intra-container coordination. To manage these operations, Kubernetes[13] is used as the orchestration engine. Kubernetes provides tools and functionalities for automating deployment, scaling and management of containerised applications.

[13] kubernetes.io

7 Risks and Limitations

The experiments and results were impacted by the disruptions caused by COVID-19 lockdown. The lockdown posed multiple communication and development challenges as it disrupted the normal flow of the project timelines. Due to the closure of the Department of Computer Science and the university in general, it was not possible to access the physical cluster and hence the horizontal scaling was not possible beyond the initial 2-node cluster. It's our understanding that experimentation with a bigger pool of slave nodes would definitely yield more insights into the pros and cons of the different Distributed Text Processing Frameworks we evaluated.

Another limitation associated with our evaluation is that the relatively small size of each individual *data object*, since each data object consisted of a review or a short text span. This played an important role specially in the low performance of UIMA DUCC which is built to breakdown large computations into smaller units to be distributed across the cluster.

We used ElasticSearch version 6.8.1 for the evaluation of DTPFs and the development of the NLAE. Some of the functionality in the corresponding Java API has since been deprecated and might be completely removed in the future versions.

Finally, with respect to UIMA DUCC, we feel that one of the biggest limitations was the lack of supporting material available online and a somewhat cryptic documentation. This caused a steep learning curve and may have affected the evaluations adversely.

8 Typhon Requirements

In this section, we review the technology requirements and use case requirements of deliverable D1.1 that are related to text processing. Tables 7 and 8 present our progress towards each technology and use case requirement, respectively. Requirements that are to be completed are marked with the task number and the time frame in which they will be addressed. We have edited them from the previous deliverable D5.4 according to our progress.

Table 7: Consolidated Technology Requirements

Number	Requirement	Priority	Status
WORKPACKAGE 2: HYBRID POLYSTORE DESIGN			
D1	TyphonML shall enable the specification of data entities and relationships that will be stored in TyphonML	Shall	Done
D4	Definition of custom data types to be used in TyphonML data models shall be supported.	Shall	Done
D5	Specification of data types that are needed for applying text-specific analysis (e.g. text, video, recordings) shall be supported.	Shall	Done
D6	The definition of structured data types (e.g. sentences, facts, entities, events) that can be extracted from text and represented in TyphonML shall be supported.	Shall	Done
WORKPACKAGE 4: HYBRID POLYSTORE QUERYING			
D33	The TyphonQL engine shall support normalization of natural language fragments to enable "querying modulo spelling".	Shall	Done
D36	TyphonQL shall support querying textual data.	Shall	Done
WORKPACKAGE 5: HYBRID POLYSTORE ANALYTICS AND MONITORING			
D50	The development of text mining pipelines for data events shall be simplified.	Shall	Done
WORK PACKAGE 7: PLATFORM INTEGRATION AND EVALUATION			
D76	Each of the Typhon components shall adhere to the specified TYPHON architectural guidelines	Shall	Done
D77	Each of the Typhon components shall use Git for source code control	Shall	Done

Table 8: Use-case requirements

Number	Requirement	Priority	Status
TEXT DATA MODELLING			
27	The text data storage shall be able to parse its data to a relational database	Shall	Done
28	The text data storage shall be able to parse its data to an array database	Shall	Done
29	Text data modelling for XML files shall be provided	Shall	Done
POLYSTORE QUERY LANGUAGE			
47	The polystore query language should expose the same semantic data types and operations of the underlying database technologies as defined in their schemas or metamodels	Should	Done
52	The query language shall be able to interpret and execute text search queries	Shall	Done
QUERIES ON STRUCTURED DATA			
58	The system shall be able to process queries on relational databases	Shall	Done
61	The system shall be able to process queries on text stores	Shall	Done
QUERIES ON TEXTUAL DATA			
63	The polystore query language should expose a relevant subset of the data types and operations of at least one of the following: Solr, Lucene	Should	Partially done
64	In the text data queries it shall be possible to search for one or more different keywords in one query	Shall	Done
65	Text data queries using patterns or full text search shall be supported	Shall	Done
66	The text data queries may be able to autocorrect words	May	To be done
67	The text data queries may be able to recognise incorrect spellings and mark them	May	To be done
68	The text data queries may be able to recognise different spellings for one word (AE/BE)	May	To be done
69	The text data queries shall be able to recognise the Greek language	Shall	Done

9 Conclusion

This deliverable presented our progress with task 5.6, where we tested the performance of Natural Language Processing (NLP) pipelines within distributed environments to highlight design decisions for the development of the Natural Language Analysis Engine for TYPHON.
We designed and developed a number of text processing pipelines, and measured their execution using three frameworks. We investigated parallel and distributed data processing frameworks that can be used to scale out the performance of NLP functionality within distributed settings. We outlined our experimental findings and discussed factors such as compatibility and granularity of NLP toolkits, which impact performance when used in a distributed fashion. Our results also examined the potential for data and task parallelism and argues for a combination of both to maximise performance gains for NLP pipelines. Finally, based on our findings, we provided a detailed description of the design and development of the Natural Language Analysis Engine (NLAE) that integrates into the TYPHON ecosystem to provide text processing capabilities. We provided details about the external and internal components of the NLAE and how they interact with each other to perform NLP tasks supported by TYPHON.
Future work involves the extension of experiments to a wider range of NLP pipelines.We also aim to expand the size of our cluster and also integrate different data sources within our experiments for testing. Additional experiments can help uncover further NLP pipeline benchmarks and integrate these pipelines within NLAE.

References

[1] Elizabeth D Liddy. Natural language processing. 2001.

[2] Rodrigo Agerri, Xabier Artola, Zuhaitz Beloki, German Rigau, and Aitor Soroa. Big data for natural language processing: A streaming approach. *Knowledge-Based Systems*, 79:36–42, 2015.

[3] Ovidiu-Cristian Marcu, Alexandru Costan, Gabriel Antoniu, and María S Pérez-Hernández. Spark versus flink: Understanding performance in big data analytics frameworks. In *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 433–442. IEEE, 2016.

[4] Jeyhun Karimov, Tilmann Rabl, Asterios Katsifodimos, Roman Samarev, Henri Heiskanen, and Volker Markl. Benchmarking distributed stream data processing systems. *arXiv preprint arXiv:1802.08496*, 2018.

[5] James R Challenger, Jaroslaw Cwiklik, Louis R Degenaro, Edward A Epstein, and Burn L Lewis. Distributed uima cluster computing (ducc) facility, July 19 2016. US Patent 9,396,031.

[6] Ruining He and Julian McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. In *proceedings of the 25th international conference on world wide web*, pages 507–517, 2016.

[7] Erik F. Tjong Kim Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. In Walter Daelemans and Miles Osborne, editors, *Proceedings of CoNLL-2003*, pages 142–147. Edmonton, Canada, 2003.

[8] Marti Hearst. What is text mining. *SIMS, UC Berkeley*, 5, 2003.

[9] Gregory Grefenstette. Tokenization. In *Syntactic Wordclass Tagging*, pages 117–133. Springer, 1999.

[10] Katrin Tomanek, Joachim Wermter, and Udo Hahn. Sentence and token splitting based on conditional random fields. In *Proceedings of the 10th Conference of the Pacific Association for Computational Linguistics*, volume 49, page 57, 2007.

[11] Christian Buck, Kenneth Heafield, and Bas Van Ooyen. N-gram counts and language models from the common crawl. In *LREC*, volume 2, page 4. Citeseer, 2014.

[12] Jay Nanavati and Yogesh Ghodasara. A comparative study of stanford nlp and apache open nlp in the view of pos tagging. *Int. J Soft Comput. Eng.*, 5(5):57–60, 2015.

[13] Abdul Ghaffar Shoro and Tariq Rahim Soomro. Big data analysis: Apache spark perspective. *Global Journal of Computer Science and Technology*, 2015.

[14] Salman Salloum, Ruslan Dautov, Xiaojun Chen, Patrick Xiaogang Peng, and Joshua Zhexue Huang. Big data analytics on apache spark. *International Journal of Data Science and Analytics*, 1(3-4):145–164, 2016.

[15] Diego García-Gil, Sergio Ramírez-Gallego, Salvador García, and Francisco Herrera. A comparison on scalability for batch big data processing on apache spark and apache flink. *Big Data Analytics*, 2(1):1, 2017.

[16] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. State management in apache flink®: consistent stateful distributed stream processing. *Proceedings of the VLDB Endowment*, 10(12):1718–1729, 2017.

[17] Mark Van Rijmenam, Tatiana Erekhinskaya, Jochen Schweitzer, and Mary-Anne Williams. Avoid being the turkey: How big data analytics changes the game of strategy in times of ambiguity and uncertainty. *Long Range Planning*, 52(5):101841, 2019.