Unlocking the Heterogeneous Landscape of Big Data NLP with DUUI

Alexander Leonhardt **Giuseppe Abrami** **Daniel Baumartz** **Alexander Mehler**
Goethe University Frankfurt
Robert-Mayer-Strasse 10
60325 Frankfurt am Main
s0637113@stud.uni-frankfurt.de • {abrami,baumartz,mehler}@em.uni-frankfurt.de

Abstract

Automatic analysis of large corpora is a complex task, especially in terms of time efficiency. This complexity is increased by the fact that flexible, extensible text analysis requires the continuous integration of ever new tools. Since there are no adequate frameworks for these purposes in the field of NLP, and especially in the context of UIMA, that are not outdated or unusable for security reasons, we present a new approach to address the latter task: DOCKER UNIFIED UIMA INTERFACE (DUUI), a scalable, flexible, lightweight, and feature-rich framework for automatic distributed analysis of text corpora that leverages Big Data experience and virtualization with Docker. We evaluate DUUI's communication approach against a state-of-the-art approach and demonstrate its outstanding behavior in terms of time efficiency, enabling the analysis of big text data.

1 Introduction

Automatic analysis of text corpora is an important task in many research areas that use Natural Language Processing (NLP). This includes fields as diverse as digital humanities (e.g. Brooke et al., 2015), economics (e.g. Qureshi et al., 2022), or biodiversity research (Driller et al., 2020). All these areas are supported by the availability of increasingly large text repositories such as newspaper corpora (e.g. New York Times, 2019), parliamentary corpora (e.g. Barbaresi, 2018; Rauh and Schwalbach, 2020; Abrami et al., 2022) or Wikipedia (Pasternack and Roth, 2008). However, text processing of large corpora is time-consuming, resource-intensive, and thus costly. In addition, NLP routines are required as pre-processing steps to perform even more time-consuming tasks as, e.g., textual entailment (Paramasivam and Nirmala, 2021), rhetorical analysis (Joty et al., 2015) or argument mining (Ding et al., 2022). In any event, the application of NLP methods is gaining acceptance in almost all text-based disciplines. This increasingly leads to scenarios in which ever larger corpora have to be processed regarding ever more complex tasks while ensuring a mixed methods approach (Johnson et al., 2007). In contrast, project participants, especially those outside the NLP field, often have limited computer skills. Thus, a suitable framework for flexible, transparent, and time-optimized pre-processing of large corpora is essential, allowing the integration of ever new analysis methods. For this purpose, software is required, which serves the following functions:

(A) **Horizontal and vertical scaling:** Since time is a significant factor, pre-processing must be horizontally and vertically scalable. That is, pre-processing tools should be deployable on different systems, regardless of whether they are on servers or workstations. This is what we call *horizontal scaling*. In addition, resource-intensive processes should be automatically delegated to resource-rich machines to enable their parallelization. This is what we call *vertical scaling*. Increasingly large corpora demand similar scaling properties of NLP (Divita et al., 2015; Kim et al., 2021).

(B) **Capturing heterogeneous annotation landscapes:** For the various pre-processing tasks (e.g., tokenization, lemmatization, PoS tagging, parsing, coreference analysis, NER etc.), there are varieties of tools (e.g. Stanford NLP (Manning et al., 2014) and spaCy (Honnibal et al., 2020)), that use proprietary annotation formats, especially when there are no annotation standards for the task. Therefore, in order to use the results of such tools and establish their comparability, it is necessary to do so in a common environment that abstracts from the underlying output formats.

(C) **Capturing heterogeneous implementation landscapes:** Competing tools for the same task may be implemented in different programming languages that impose different requirements on the runtime behavior of these tools (Bhatnagar et al., 2022). Thus, such tools should be reusable simultaneously and independently of the platform and programming language. At the same time, tools may not be available locally, but only, e.g., via an API (e.g., RESTful, WebSocket). This in turn leads to the requirement that a suitable NLP platform encapsulates the heterogeneity of different implementation landscapes.

(D) **Reproducible & reusable annotations:** Evaluating automatically generated annotations requires tracking the engines or pipelines that created them. This is important, e.g., if the data is to be shared among scientists. Thus, any application of an NLP pipeline should be reproducible, just as these pipelines should be reusable and modifiable (Trisovic et al., 2020; Jupyter et al., 2018; Cacho and Taghva, 2020). It should be possible to track pipelines at the level of individual documents or sequences thereof. As shown by (Leonhardt, 2022), this approach can induce a considerable overhead in terms of memory usage and performance to be managed by the NLP platform.

(E) **Monitoring and error-reporting:** Running NLP routines may fail. The reasons for such failures range from errors in the input data, missing information, platform peculiarities to programming errors. Thus, a powerful monitoring and error reporting system is needed to troubleshoot and monitor processing statuses.

(F) **Lightweight usability:** To utilize the wide range of NLP methods, knowledge of computer science and programming is required. In particular, the maintenance and further development of such tools requires a high level of expertise. This results in two requirements: in terms of ease of use and maintenance of the entire infrastructure, and the ease with which tools can be integrated or updated.

Currently, there is no framework in NLP that meets criteria A–F. Moreover, the number of usable frameworks has recently decreased significantly due to Log4j vulnerability (Hiesgen et al., 2022), which occurred in 2021 (see, e.g., the Apache framework *DUCC*). Text corpora are therefore increasingly processed with proprietary systems, which increase development effort and are ultimately not reusable. We present an approach that meets requirements A–F to overcome this trend: DOCKER UNIFIED UIMA INTERFACE (DUUI), an efficient, horizontally and vertically scalable framework that handles big text data in a simple, modular, and reusable manner, while mitigating annotation- and implementation-related heterogeneities. Similar to TEXTIMAGER (Hemati et al., 2016), DUUI is based on UIMA (Unstructured Information Management Applications) (Ferrucci et al., 2009). UIMA is used in numerous projects and remains an active area of research, notwithstanding the shortcomings noted by Götz et al. (2014) – see Figure 11 in the appendix. DUUI is available on GitHub[1] and provides a lightweight framework to utilize different annotators in a distributed setup based on several implementations embedded into Docker (`www.docker.com`) containers used as microservices.

There are many proprietary NLP frameworks (see Section 2). DUUI accounts for this heterogeneity: it integrates these frameworks, supports interfaces, and stores all generated annotations and analysis results in UIMA. UIMA allows for defining software modules for the analysis of unstructured data (e.g., texts). This is done with so-called *Analysis Engine*s (AE). The results of an analysis are stored in a *Common Analysis Structure* (CAS) (Götz and Suhre, 2004) with reference to a defined schema description (*Type System Descriptors*). CAS is the basic data structure in which data is analyzed and stored with metadata. To process data, each AE receives a CAS object, performs its analyses, and passes the revised CAS object to the next AE. Thereby, each pre-processing step comprises one to several pipelines, which are encapsulated and managed by DUUI.

In this paper, we describe the principles and implementation of DUUI. We evaluate its efficiency and show that DUUI meets the criteria A–F. In Section 2, we present related work and introduce DUUI in Section 3. In Section 4, we present an evaluation based on benchmark studies and address issues regarding annotation frameworks. We conclude with an outlook on future work (Section 5) and a conclusion (Section 6).

[1]`https://github.com/texttechnologylab/DockerUnifiedUIMAInterface`

2 Related Work

DUUI's use cases are related to various research areas, including high-performance computing (HPC) and reproducible pipelines.

Several approaches exist to improve application scalability based on UIMA. One of the most common approaches to scale AEs, the *Apache Hadoop cluster* (Apache, 2006), is used by Exner and Nugues (2014); Zorn et al. (2013); Nesi et al. (2015); Aydin and Hallac (2018). It works well for simple pipelines that are decomposable into map reduce jobs, but has problems with more complex scenarios. Scenarios regarding routines that do not entirely fit into main memory are particularly difficult to model with an Apache Hadoop cluster. In addition, detecting and fixing *Out Of Memory* (OOM) issues appears to be a recurring problem in Apache Hadoop (Xu et al., 2015). Further, complex AEs that combine multiple aggregations after initial information extraction are difficult to accomplish (Götz et al., 2014). An alternative approach is proposed by Hodapp et al. (2016), which follows a similar microservice-centric approach as DUUI.

Despite this similarity, DUUI offers a considerably simpler setup that does not require the laborious setup of an *Apache Message Broker*.

An alternative approach based on microservices is the *Computation Flow Orchestrator* (CFO) (Chakravarti et al., 2019). It uses technologies, such as Docker and Kubernetes[2], and Google's *Buffer Interface Definition Language protocol*. CFO organizes analyses as flows and distributes them to individual nodes addressed by an orchestrator, which generates a set of startup scripts and REST interfaces using Docker in preparation.

In this way one gets access to the individual endpoints and to debug information. This approach is similar to DUUI, but requires time-consuming learning of the scripting language, while native UIMA integration is not directly possible – a problem for many frameworks. Further, its setup is not straightforward, as profound knowledge of Docker and the underlying network bridge is required; alternatively, the makefile would have to be changed to allow the system to be compiled without Docker. Another approach comes from Agerri et al. (2015) and Arshi Saloot and Nghia Pham (2021). It is based on *Apache Storm*, which also enables distributed processing of texts via microservices. GATE Cloud (Tablan et al., 2013) allows for distributing processes with UIMA components. But since GATE is a large and complex system, its use requires a significant amount of learning.

Framework	A	B	C	D	E	F
Apache DUCC	+	+	○	−	○	−
Apache Storm	+	+	○	−	○	○
Hadoop Cluster	+	○	+	−	○	−
CFO	+	+	+	−	○	○
TextImager	+	+	+	−	−	○
GATE Cloud	+	+	○	−	○	○
Data-Centric	−	+	+	−	○	+
Flyte	+	+	+	−	+	−
argo	+	+	+	−	+	−
DUUI	+	+	+	+	+	+

Table 1: Comparison of frameworks according to the criteria A–F (columns) of Section 1: + (fulfilled), ○ (partially fulfilled), − (not fulfilled).

The *Data-Centric Framework* (Liu et al., 2020) offers a more complete workflow, which allows for pre-processing, visualization and post-annotation. It generalizes the UIMA schema to map NLP data in a Python environment based on Forte[3]. This solution lacks the ability to split pre-processing across servers and is neither platform nor programming language independent. In addition, its proprietary use of UIMA makes reuse difficult.

A similar tool is *Flyte*[4], implemented in Golang and available for Python, Java, and Scala, which distributes pre-processing to Kubernetes containers. This rather complex framework shares some features with DUUI, but getting started and setting up are challenging. While it is positive that data formats are not predefined, it makes subsequent use in existing frameworks more time-consuming. This also holds for *argo*[5], which also uses Kubernetes to realize the mapping and running of workflows.

Using criteria A–F from Section 1, these frameworks can be compared with each other, as shown in Table 1. It should be noted that is difficult to compare the use and effort required to maintain or reuse the components of these frameworks (Criterion F). However, in addition to providing powerful NLP methods, a desired framework should be easy to use, especially for non-experts. Since there was no framework that meets all the criteria from A to F, we developed DUUI to fill this gap. DUUI enables

[2]https://kubernetes.io

[3]https://github.com/asyml/forte

[4]https://flyte.org/

[5]https://argoproj.github.io/

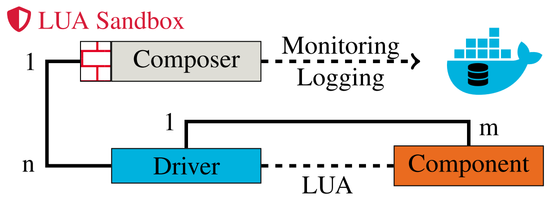

Figure 1: The modular architecture of DUUI

the implementation of these criteria by using UIMA as annotation standard. Thus, documents annotated in UIMA can be reused without DUUI. However, feature D concerning reproducibility, which is fulfilled by DUUI, goes further: it refers to the ability to reconstruct and reuse all annotations performed, including analysis engines, models, and settings.

3 Docker Unified UIMA Interface

The problems described so far cause that the implementation of all approaches from Section 2 is time-consuming and ultimately makes their maintenance impossible. Most of these frameworks meets more than half of the requirements A–F from Section 1 (cf. Table 1) for NLP based on Big Data.

To overcome this shortcoming, DUUI integrates functions in a modular way, making its operation reproducible and usable by non-experts. It enables source-text reduced programming and is mainly based on Docker containers, while allowing for integrating other systems (see Figure 1). DUUI uses so-called COMPOSERs, possibly connected to a database to enable dynamic monitoring, which consist of n DRIVERs, each containing up to m COMPONENTs, each of which encapsulates an *Analysis Engine* (AE). Since this cross-programming-language system bridges functions given their heterogeneous implementations, it is reasonable to create a unified interface for Docker containers to enable cross-implementation operations. This includes two aspects, as Docker allows individual containers to use different programming languages (Section 1, Feature C) and taggers (Feature B). Beyond that, processes can also be executed in the local *Java Runtime Environment* (JRE) as shown in Figure 2.

3.1 Composer

The role of the COMPOSER is similar to the role of the master in the *Map Reduce Framework* (Dean and Ghemawat, 2004). It initializes all DRIVERs with their respective COMPONENTs, controls the document flow through the pipeline and reports metrics to the database. The COMPOSER also manages the global Lua (Ierusalimschy et al., 2007) state, providing a shared set of global libraries and security policies to the underlying DRIVERs. Lua is used to realize programming language independent communication between the individual AEs (see Section 3.4). By taking control over the error handling and the advanced control flow mechanisms, every pipeline can have a policy fit to its task. The COMPOSER starts pipeline processing by reading a set of documents and passing them through the pipeline. Upon completion of the pipeline, the COMPOSER terminates and cleans up instantiated DRIVERs and COMPONENTs. COMPOSERs allow for monitoring and logging of NLP routines (Ta-

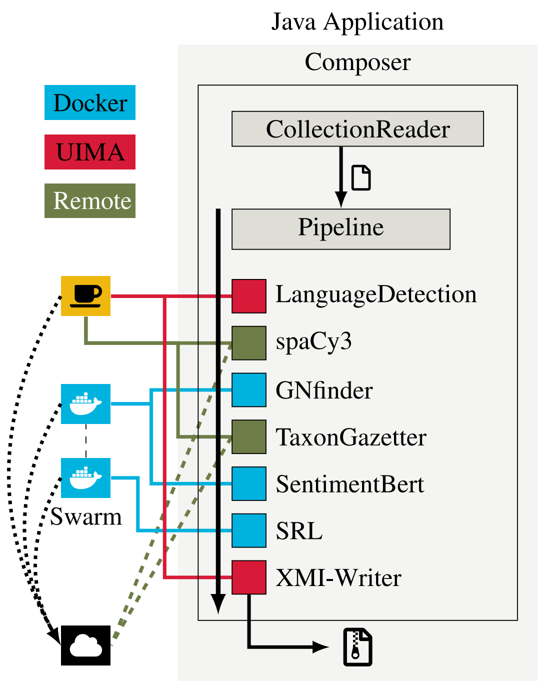

Figure 2: An example DUUI pipeline which, like all DUUI pipelines, is part of a Java application: a composer is instantiated containing a set of drivers and associated components. Several pipelines are registered as components in corresponding drivers. Each driver defines the communication between the implementations of the components. *UIMA* and *Remote* are executed in *Java Runtime Environment* (☕). Components can also be run as Docker containers, that is, as DOCKER DRIVER or SWARM DRIVER. In the first driver, services are used as Docker images running on the local system as Docker containers to provide vertical scaling. The second driver also uses Docker images, but is distributed in a Swarm network of attached Docker nodes to provide horizontal scaling. For security reasons, using Docker Swarm is only possible if the pipeline is initially executed on the Docker leader node.

ble 1, Feature E). For this purpose, an SQLite database and a Docker connection via ArangoDB[6] or InfluxDB[7] are available. Due to our interface design, other database management systems can be added. This eases troubleshooting and performance analyses. Further, a Docker image instantiated in Swarm mode can be used to overview the workload on the Swarm network.

3.2 Driver

A DRIVER instantiates, orchestrates and manages a set of COMPONENTs. Each DRIVER serves as a bridge to a specific functionality, thereby giving almost native access to the underlying software. This enables advanced use cases, e.g., scheduling containers on specific nodes or restricting system resources for specific parts of a pipeline. Because DRIVERs are designed as interfaces, further application environments can be included extending the given ones. DUUI currently includes four predefined DRIVERs to scale NLP processes horizontally and vertically while running COMPONENTs on the local machine as well within Docker swarm:

1. The **UIMA DRIVER** runs a UIMA AE on the local machine (using local memory and processor) in the same process within the JRE and allows scaling on that machine by replicating the underlying AE (Feature A in Section 1). This enables the use of all previous analysis methods based on UIMA AE without further adjustments.

2. **DOCKER DRIVER:** The DUUI core DRIVER runs COMPONENTs on the local Docker daemon and enables machine-specific resource management (vertical scaling, Feature A). This requires that the AEs are available as Docker images according to DUUI to run as Docker containers. It is not relevant whether the Docker image is stored locally or in a remote registry, since the Docker container is built on startup. This makes it very easy to test new AEs (as local containers) before being released. The distinction between local and remote Docker images is achieved by the URI of the Docker image used.

3. The **SWARM DRIVER** complements the DOCKER DRIVER; it uses the same functionalities, but its AEs are used as Docker images distributed within the Docker Swarm network (horizontal scaling, Feature A). A swarm consists of n nodes and is controlled by a leader node within the Docker framework. However, if an application using DUUI is executed on a Docker leader node, the individual AEs can be executed on multiple swarm nodes.

4. **REMOTE DRIVER:** AEs that are not available as containers and whose models can or should not be shared can still be used if they are available via REST. Since DUUI communicates via Restful, remote endpoints can be used for pre-processing. In general, AEs implemented based on DUUI can be accessed and used via REST, but the scaling is limited regarding request and processing capabilities of the hosting system. In addition, COMPONENTs addressed via the REMOTE DRIVER can be used as services. This has advantages for AEs that need to hold large models in memory and thus require a long startup time. To avoid continuous reloading, it may be necessary to start a service once or twice in a dedicated mode and then use a REMOTE DRIVER to access it. To use services, their URL must be specified to enable horizontal scaling.

DRIVERs can be added to integrate processes in other runtime environments (see Section 5). The encapsulation of the implementation of the AEs is done by means of their COMPONENTs.

3.3 Component

A COMPONENT represents an AE according to UIMA and is defined by the instantiating DRIVER which imposes requirements on its COMPONENT. The implementations of the DRIVERs differ, with UIMA DRIVER being closest to the AEs of UIMA. The latter instantiates and uses its COMPONENT based solely on the associated AE, so no modification to an existing AE is required. Therefore, existing AEs implemented in Java can be used directly in DUUI, although scaling is limited to the vertical level due to the limitation of execution in the JRE. Unlike UIMA DRIVER, COMPONENTs of all other DRIVERs (see Section 3.2 – Drivers) must follow the RESTful scheme of DUUI.

3.4 Communication

Given heterogeneous programming languages (Section 1), there is a need for a method for the communication of COMPONENTs that is fast, flexible, and

[6] https://www.arangodb.com/
[7] https://www.influxdata.com

error tolerant. The communication within DUUI is designed to integrate a variety of programming languages, even without direct access to the UIMA framework. This requires a transformer that converts UIMA documents so that they can be processed by the programming language of the respective COMPONENT. In addition, partial serialization of documents is required because the size of UIMA documents increases rapidly during preprocessing. To meet these requirements, we use Lua as an embedded scripting language for communication. Through fine-grained access and interoperability within a *Java Virtual Machine* (JVM), Lua allows annotators to precisely define elements of the documents to be processed, and to use all communication formats that Lua or Java support. This flexibility is exemplified by implementations of annotators in Java, Python, and Rust using various formats such as MessagePack, JSON, UIMA XMI and UIMA Binary. Figure 12 and 13 (appendix), compare this in detail. By using Lua (see Figure 15 (appendix)), DUUI meets Feature B and C.

3.5 Lua Sandbox

Although the potential of Lua is outstanding, one aspect deserves special attention because it can harm the host system. A Lua script allows the client COMPONENT to execute invoked methods without verification by the host system, allowing unrestricted access to the host. To solve this problem, we implemented a "sandbox" that limits the number of statements executed by the Lua interpreter before it aborts execution. The sandbox works like a class loader, restricting Lua calls and allowing only a certain number of named classes. This means that the COMPOSER (Figure 1) can be equipped with a firewall (the so-called sandbox) that restricts LUA scripts to use only authorized classes like a *white list*. Therefore all other classes on the host system (COMPOSER) are not accessible (as shown in Figure 14 in lines 07-10).

3.6 Reproducibility

DUUI fulfills Feature D: reproducibility at document level. Each pipeline component is fully serializable and annotates each processed document. The reproducibility level of a component depends on its type: components of UIMA DRIVER and REMOTE DRIVER are less reproducible than the components of other drivers. For REMOTE DRIVER, this is partly because the endpoint providing the annotation service cannot be controlled. Because of this design, users can provide services that are copyrighted or otherwise legally restricted without having to redistribute the software. Despite the specification of the full COMPONENT in the processed document, different package managers and the inability of Java to provide source code along with the package version mean that the respective pipelines cannot be reproduced without the help of the user. DOCKER DRIVER and REMOTE DRIVER overcome these limitations by enabling pipeline replication across multiple hosts and environments.

3.7 DUUI-integration

The goal of DUUI is the straightforward integration of NLP routines on three levels: (1) Operation of existing and new AEs without the need to integrate a new library (e.g. *Flyte*, *argo*) or build a more complex infrastructure (e.g. *Flyte*, *GATE*, *CFO*) and integrate AEs with different programming languages. (2) Integrate simple to complex AEs by connecting self-contained tools with DUUI, which provides a set of existing components in Docker images[8] each routine runs as usual, with its results returned to the COMPOSER only via REST to keep the overall integration manageable. (3) Using DUUI within existing infrastructures requires that they use XMI as an exchange format; otherwise, existing readers or writers (e.g. CoNLL, JSON, etc.) must be used. The use is directly possible via Java or a terminal call by passing documents to be processed. This is certainly the most complex solution compared to the previous points, but it can be encapsulated by a web API. In any event, DUUI can be implemented and used with manageable to low effort, as required by Feature F. All DUUI libraries can be included via Maven (see Figure 14 in the appendix for an example setup).

4 Evaluation

We conducted several tests to show the feasibility of DUUI. Besides the projects listed in Section 2, there is only one project that is not implemented in Java and enables UIMA processing directly in Python: dkpro-cassis (Klie and de Castilho, 2020). As tools are increasingly developed exclusively in Python, the use of a native Python library is important for UIMA, and its performance characteristics are a key basis for comparison with implementations in non-native (e.g., non-Java) programming languages. We computed benchmarks for the Ger-

[8]Several AE's are available via GitHub

man parliamentary corpus (Barbaresi, 2018) and the largest corpus of German parliamentary minutes, GerParCor (Abrami et al., 2022), which contains 37 881 with spaCy annotated minutes from three centuries. At first sight, the subset relation of both corpora seems odd, but their individual processing is an important indicator, as will be shown.

4.1 Serialization

For evaluating the serialization speed between Java and Python (using dkpro-cassis as a reference) all texts in the test corpus are processed with spaCy and BreakIteratorSegmenter. To this end, we use the Barbaresi (2018) corpus, since GerParCor (due to its coverage period) contains characters that are not XML 1.0 conform and thus cannot be processed by dkpro-cassis. Figure 3 and 4 show that the serialization speed is linear for the Java CAS implementation and linear with a tendency to be polynomial for dkpro-cassis. In any event, the former outperforms the latter by a large margin (see Table 2 and Figure 5). dkpro-cassis' serialization performance quickly degrades when handling larger documents (see Figure 6). Moreover, its mandatory use of XML 1.0 limits the processable character set in contrast to the Lua communication of DUUI.

Using UIMA documents directly in Python via dkpro-cassis is a step ahead. But since more and more NLP tools are implemented in Python, a (de-)serialization in this language leads to a significant increase in runtime depending on document size.

Framework	Mean speed	Max exec. time
Java XMI	154,42 ms	1 034,31 ms
dkpro-cassis	1 619,58 ms	14 464,83 ms

Table 2: (De-)serialization performance of the Java XMI and the Python dkpro-cassis implementation.

4.2 Partial serialization

Often, annotators need only a subset of all available annotations. Therefore, it is important to allow partial serializations of a document to save bandwidth and CPU cycles. Since we introduced the Lua bridge for communication between annotators of different programming languages, we next compared document (de-)serialization between Java, Python and Rust annotators: To this end, we developed the first annotator in Rust that is usable with a UIMA pipeline. The task was to serialize all tokens from GerParCor. The reason for this choice was

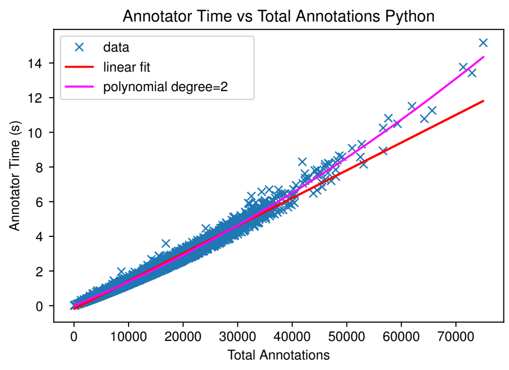

Figure 3: Python, using *dkpro-cassis*, (de-)serialization speed (seconds) against the total number of document annotations.

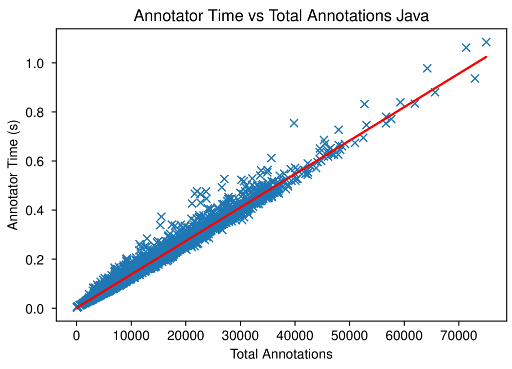

Figure 4: Java (de-)serialization speed in seconds plotted against the total number of document annotations.

that token annotations account for about one-third of all annotations in documents. Figure 7 shows a clear disadvantage of the Python implementation compared to Java and Rust, whose times are linear to the number of annotations.

4.3 Bottleneck

COMPOSERS execute Lua code to transform CAS documents into a domain processable by the respective annotator and transform the annotator's response back into the CAS domain. This creates a bottleneck, as this conversion is currently performed on the executing system. We use *Amdahl's law* (Rodgers, 1985) to investigate this bottleneck:

$$S_{latency}(s) = s/((1-p) \cdot p)$$

It describes a program's speedup with two parameters: the fraction p of the program that is parallelizable and the number s of concurrent executors working on the parallelizable fraction of the program. We process GerParCor with the spaCy-COMPONENT and deduce the parallelizable frac-

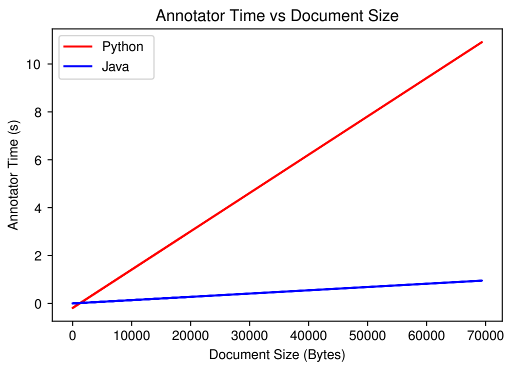

Figure 5: Java and Python (de-)serialization speed in seconds plotted against the total document annotations.

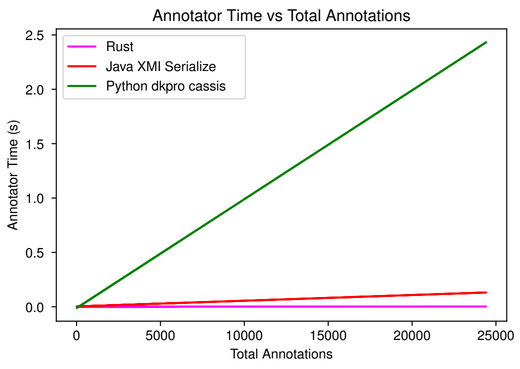

Figure 7: Token (de-)serialization in Java, Rust and Python.

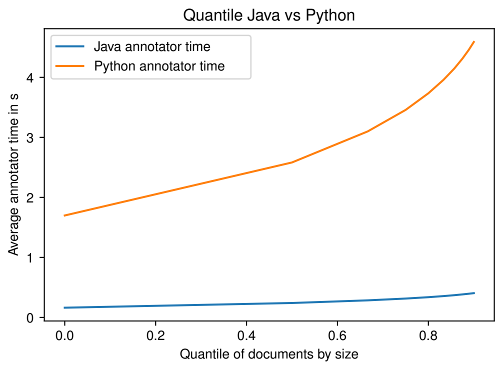

Figure 6: Document selection by quantile number: e.g., 0.8 refers to all documents that make up the top 20% of the largest documents. The frameworks are compared by their mean (de-)serialization time on each quantile.

tion of it from the mean time spent in the COMPONENT according the total time for the COMPONENT to run. Figure 16 in the appendix shows the parameter space and the balance between the computational power of the computer running the COMPOSER and the number of instances that can be distributed over any available network.

4.4 DUUI benchmarks

The major bottleneck is (de-)serialization of documents. Since this factor increases linearly and is greatly reduced by the use of Lua, there remains the processing time within each AE. To evaluate the usability of DUUI, we performed a runtime measurement in different scenarios. Processing with DUUI for spaCy was performed for a ramdomized sample of GerParCor with 100, 200, 500, and 1,000 documents using 1, 2, 4, and 8 Docker instances. Figure 8 shows the results: the processing time improves considerably by increasing the number of processes. This is evident in both variants, local Docker and Swarm, although increasing the number of local Docker instances to more than 8 would not be practical due to hardware limitations in the evaluation setup. Examples 100 and 500 show that a reduction in processing time can be achieved by increasing the number of instances. However, this is not always the case, as depending on the system load, the total runtime can also be higher for multiple processes (e.g. Sample 200 / 1000, Docker8). Although similar behavior occurs in Swarm mode, the overall processing time is higher compared to local Docker because the network-based I/O of the components consumes additional time. In the setup used, no prioritization of the available swarm nodes was done, so the usage of each node is random. Since the individual nodes have different hardware performance and are located in different network segments, this results in lower performance for large documents. The result in Figure 17 (see appendix), using up to 32 instances in swarm mode, shows similar behavior, although here sample size was not randomized, but the n smallest documents were chosen in each case to neglect network latencies. This shows that the number of COMPONENTS in a swarm network can be dynamically increased using DUUI. Next, we compared DUUI with *Flyte* and *CFO* (for the results, see Figure 9 and Figure 10) for the comparison with *Flyte*, documents from the Barbaresi (2018) corpus were preprocessed with *spaCy*, using XMI as I/O. For both tools a Python script with *dkpro-cassis* was implemented to (de)serialize the XMI documents. When comparing with *CFO*, the same corpus but a different library was used to enrich the input documents; this concerns tokenization using *NLTK* (Bird et al., 2009). The results show the clear advantages of DUUI over *Flyte* in processing UIMA documents, while it is on par with *CFO* but increases the versatility of the data flow by embedding a Turing-complete language.

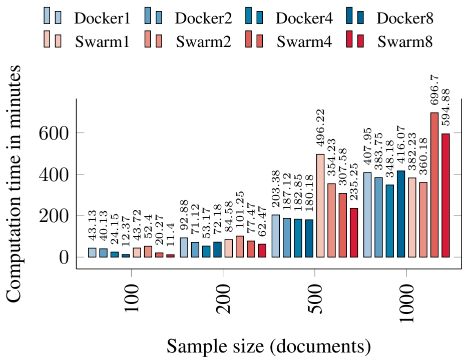

Figure 8: To illustrate the performance of DOCKER DRIVER and SWARM DRIVER, we processed a GerParCor sample using a spaCy-COMPONENT.

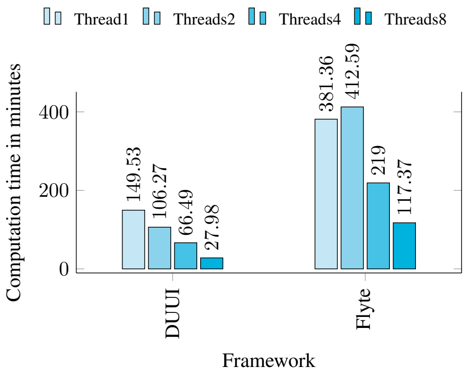

Figure 9: Runtime comparison between DUUI and *Flyte*, for processing *German political speeches* (Barbaresi, 2018) with 1, 2, 4 and 8 threads respectively. The single-core variant for *Flyte* was run outside; the variants with 2, 4 and 8 threads were run using the Docker in Docker demo provided by the *Flyte* sandbox. In DUUI, all processing performed ran in a spaCy-COMPONENT within the DOCKER DRIVER.

5 Future Work

Since there are container solutions besides Docker (e.g., OpenShift, Kubernetes and Podman), they are candidates for DUUI drivers. This would solve a problem with Docker, as GPU usage is limited: an issue that has been pending for over six years. The bottleneck of single-machine multi-threaded (de-)serialization may be solved with ActiveMQ (Snyder et al., 2011). Since we implemented a new form of Big Data NLP, optimization becomes urgent. This concerns prioritization in clusters and parallelization of individual COMPONENTs within the pipeline, as long as these components do not

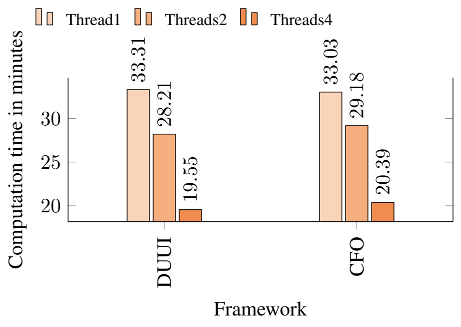

Figure 10: Runtime comparison between DUUI and *CFO*, for processing *German political speeches* (Barbaresi, 2018) with 1, 2, 4 threads respectively. Both runtimes were set up to run in a dockerized environment. As can be seen here the performance of DUUI and *CFO* is almost on par even though *CFO* makes already use of high performance optimizations like the usage of HTTP/2 and dedicated docker networks.

depend on the results of previous analyses. Finally, the future of DUUI requires a web interface for managing and running of COMPONENTs.

6 Conclusion

We introduced DOCKER UNIFIED UIMA INTERFACE, a framework for Big Data NLP. Its design and implementation leverage experience with TEXTIMAGER and Apache DUCC to create a platform-independent, scalable, and lightweight environment that combines the functionality of a number of frameworks. It enables horizontal scaling via a native Docker Swarm implementation. Regarding platform and programming language independence based on UIMA, problems arise with XMI serialization with Python. To address this issue, DUUI implements Lua-based communication between COMPONENTs and COMPOSERs, resulting in a massive improvement in performance (see our evaluation). DUUI is currently the most powerful system meeting the requirement of Big Data NLP. It implements a Lua sandbox for security reasons to ensure that only permitted methods are executed on the host system. Due to the sandbox and the self-sufficiency of all annotators, DUUI can easily be hosted in public registries without causing conflicts between annotators. Its use of Docker containers as standalone analysis components is a new approach to implementing UIMA annotators and will be published under the AGPLv3 license. It will support Big Data analysis in a variety of disciplines that use NLP for their scientific purposes.

Limitations

In the current implementation of DUUI there are a few limitations that are explicitly listed here. The strength of DUUI becomes apparent when running all analyses in Docker containers, especially in swarm mode. Unfortunately, this also brings the following limitations:

1. In Docker Swarm, there is currently no useful way to specify load balancing, prioritization, and service management for the individual analysis processes.

2. At the same time, but this is independent of the Docker swarm, no GPU container can currently be started in a native way in order for the Docker image to access the GPU.

3. Considering that Lua invocations are executed directly on the host system, it is important for security reasons to define within a Lua sandbox which methods and classes are available.

4. In addition, one more, but small, limitation is that there are currently not many tools implemented according to DUUI, which would make it easier to use them broadly.

These limitations can already be bypassed at present. For instance, limit 1 can be handled by manually configuring the Docker Swarm network if the knowledge is available. In addition, all Docker containers that should use a GPU (limit 2) can be started and invoked manually with the required launch as a service (REMOTE DRIVER). However, existing tools are migrated in the context of projects, and the fallback provided by the UIMA DRIVER means that all existing approaches can already be reused (limit 4).

Ethical aspects

This work has been developed with the ACL Code of Ethics in consideration. With our contribution, we would like to provide an innovation in terms of systematic data processing with reference to text corpora. Therefore, due to the subject matter, our contribution does not entail any ethical issues. Regardless of this, in the long run we cannot prevent texts that are hurtful, disturbing or even legally prohibited from being processed with our framework. The authors are aware of this situation, but we also respect free research.

References

Giuseppe Abrami, Mevlüt Bagci, Leon Hammerla, and Alexander Mehler. 2022. German Parliamentary Corpus (GerParCor). In *Proceedings of the Language Resources and Evaluation Conference*, pages 1900–1906, Marseille, France. European Language Resources Association.

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

Apache. 2006. Apache Hadoop. `https://hadoop.apache.org/`. Last accessed: 04/28/2022.

Mohammad Arshi Saloot and Duc Nghia Pham. 2021. Real-time text stream processing: A dynamic and distributed nlp pipeline. In *2021 International Symposium on Electrical, Electronics and Information Engineering*, ISEEIE 2021, page 575–584, New York, NY, USA. Association for Computing Machinery.

Galip Aydin and Ibrahim Riza Hallac. 2018. Distributed nlp.

Adrien Barbaresi. 2018. A corpus of German political speeches from the 21st century. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC 2018)*, Miyazaki, Japan. European Language Resources Association (ELRA).

Roopal Bhatnagar, Sakshi Sardar, Maedeh Beheshti, and Jagdeep T Podichetty. 2022. How can natural language processing help model informed drug development?: a review. *JAMIA Open*, 5(2). Ooac043.

Steven Bird, Ewan Klein, and Edward Loper. 2009. *Natural language processing with Python: analyzing text with the natural language toolkit*. O'Reilly Media, Inc.

Julian Brooke, Adam Hammond, and Graeme Hirst. 2015. Gutentag: an nlp-driven tool for digital humanities research in the project gutenberg corpus. In *Proceedings of the Fourth Workshop on Computational Linguistics for Literature*, pages 42–47.

Jorge Ramón Fonseca Cacho and Kazem Taghva. 2020. The state of reproducible research in computer science. In *17th International Conference on Information Technology–New Generations (ITNG 2020)*, pages 519–524, Cham. Springer International Publishing.

Rishav Chakravarti, Cezar Pendus, Andrzej Sakrajda, Anthony Ferritto, Lin Pan, Michael Glass, Vittorio Castelli, J. William Murdock, Radu Florian, Salim Roukos, and Avi Sil. 2019. CFO: A framework for building production NLP systems. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): System Demonstrations*, pages

31–36, Hong Kong, China. Association for Computational Linguistics.

Jeffrey Dean and Sanjay Ghemawat. 2004. Mapreduce: Simplified data processing on large clusters.

Dimension.AI. uima in publications - dimensions.

Yuning Ding, Marie Bexte, and Andrea Horbach. 2022. Don't drop the topic - the role of the prompt in argument identification in student writing. In *Proceedings of the 17th Workshop on Innovative Use of NLP for Building Educational Applications (BEA 2022)*, pages 124–133, Seattle, Washington. Association for Computational Linguistics.

G. Divita, M. Carter, A. Redd, Q. Zeng, K. Gupta, B. Trautner, M. Samore, and A. Gundlapalli. 2015. Scaling-up nlp pipelines to process large corpora of clinical notes. *Methods of information in medicine*, 54(06):548–552.

Christine Driller, Markus Koch, Giuseppe Abrami, Wahed Hemati, Andy Lücking, Alexander Mehler, Adrian Pachzelt, and Gerwin Kasperek. 2020. Fast and easy access to central european biodiversity data with biofid. *Biodiversity Information Science and Standards*, 4:e59157.

Peter Exner and Pierre Nugues. 2014. KOSHIK-A large-scale distributed computing framework for NLP. *ICPRAM 2014 - Proceedings of the 3rd International Conference on Pattern Recognition Applications and Methods*, pages 463–470.

David Ferrucci, Adam Lally, Karin Verspoor, and Eric Nyberg. 2009. Unstructured Information Management Architecture (UIMA) Version 1.0. OASIS Standard.

T. Götz and O. Suhre. 2004. Design and implementation of the UIMA Common Analysis System. *IBM Systems Journal*, 43(3):476–489.

Thilo Götz, Jorn Kottmann, and Alexander Lang. 2014. Quo Vadis UIMA? *Proceedings of the Workshop on Open Infrastructures and Analysis Frameworks for HLT, OIAF4HLT 2014 - Held at the 25th International Conference on Computational Linguistics, COLING 2014*, pages 77–82.

Wahed Hemati, Tolga Uslu, and Alexander Mehler. 2016. Textimager: a distributed uima-based system for nlp. In *Proceedings of the COLING 2016 System Demonstrations*. Federated Conference on Computer Science and Information Systems.

Raphael Hiesgen, Marcin Nawrocki, Thomas C. Schmidt, and Matthias Wählisch. 2022. The race to the vulnerable: Measuring the log4j shell incident.

Sven Hodapp, Sumit Madan, Juliane Fluck, and Marc Zimmermann. 2016. Integration of UIMA Text Mining Components into an Event-based Asynchronous Microservice Architecture. *Proceedings of the Workshop on Cross-Platform Text Mining and Natural Language Processing Interoperability (INTEROP 2016) at LREC 2016*, pages 19–23.

Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. 2007. The evolution of lua.

R. Burke Johnson, Anthony J. Onwuegbuzie, and Lisa A. Turner. 2007. Toward a definition of mixed methods research. *Journal of mixed methods research*, 1(2):112–133.

Shafiq Joty, Giuseppe Carenini, and Raymond T. Ng. 2015. CODRA: A Novel Discriminative Framework for Rhetorical Analysis. *Computational Linguistics*, 41(3):385–435.

Project Jupyter, Matthias Bussonnier, Jessica Forde, Jeremy Freeman, Brian Granger, Tim Head, Chris Holdgraf, Kyle Kelley, Gladys Nalvarte, Andrew Osheroff, M Pacer, Yuvi Panda, Fernando Perez, Benjamin Ragan-Kelley, and Carol Willing. 2018. Binder 2.0 - reproducible, interactive, sharable environments for science at scale. pages 113–120.

Soojeong Kim, Sunho Choi, and Junhee Seok. 2021. Keyword extraction in economics literatures using natural language processing. In *2021 Twelfth International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 75–77.

Jan-Christoph Klie and Richard Eckart de Castilho. 2020. Dkpro cassis - reading and writing uima cas files in python.

Alexander Leonhardt. 2022. Reproducible annotations.

Zhengzhong Liu, Guanxiong Ding, Avinash Bukkittu, Mansi Gupta, Pengzhi Gao, Atif Ahmed, Shikun Zhang, Xin Gao, Swapnil Singhavi, Linwei Li, Wei Wei, Zecong Hu, Haoran Shi, Xiaodan Liang, Teruko Mitamura, Eric Xing, and Zhiting Hu. 2020. A data-centric framework for composable NLP workflows. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 197–204, Online. Association for Computational Linguistics.

Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations*, pages 55–60.

Paolo Nesi, Gianni Pantaleo, and Gianmarco Sanesi. 2015. A hadoop based platform for natural language processing of web pages and documents. *J. Vis. Lang. Comput.*, 31:130–138.

New York Times. 2019. New York Times. `https://developer.nytimes.com/apis`. Accessed: 2019; Data provided by The New York Times.

Aarthi Paramasivam and S. Jaya Nirmala. 2021. A survey on textual entailment based question answering. *Journal of King Saud University - Computer and Information Sciences*.

Jeff Pasternack and Dan Roth. 2008. The wikipedia corpus. Technical report.

Shafiullah Qureshi, Ba Chu, Fanny S Demers, Michel Demers, et al. 2022. Using natural language processing to measure covid19-induced economic policy uncertainty for canada and the us. Technical report, Carleton University, Department of Economics.

Christian Rauh and Jan Schwalbach. 2020. The ParlSpeech V2 data set: Full-text corpora of 6.3 million parliamentary speeches in the key legislative chambers of nine representative democracies.

David P. Rodgers. 1985. Improvements in multiprocessor system design. *SIGARCH Comput. Archit. News*, 13(3):225–231.

Bruce Snyder, Dejan Bosnanac, and Rob Davies. 2011. *ActiveMQ in action*, volume 47. Manning Greenwich Conn.

Valentin Tablan, Ian Roberts, Hamish Cunningham, and Kalina Bontcheva. 2013. Gatecloud. net: a platform for large-scale, open-source text processing on the cloud. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 371(1983):20120071.

Ana Trisovic, Philip Durbin, Tania Schlatter, Gustavo Durand, Sonia Barbosa, Danny Brooke, and Mercè Crosas. 2020. Advancing computational reproducibility in the dataverse data repository platform. In *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems*, P-RECS '20, page 15–20, New York, NY, USA. Association for Computing Machinery.

Lijie Xu, Wensheng Dou, Feng Zhu, Chushu Gao, Jie Liu, Hua Zhong, and Jun Wei. 2015. Experience report: A characteristic study on out of memory errors in distributed data-parallel applications. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 518–529.

Hans-Peter Zorn, Johannes Simon, Martin Riedl, Richard Eckart de Castilho, and Steffen Remus. 2013. DKPro BigData. `https://dkpro.github.io/dkpro-bigdata/`. Last accessed: 04/28/2022.

A Appendix

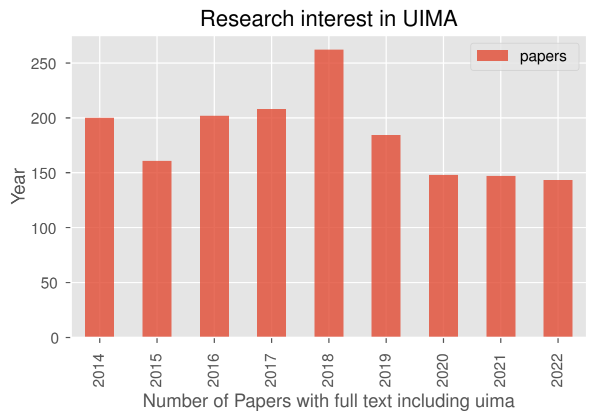

Figure 11: Research interest in UIMA in the previous years based on the number of papers including UIMA in the full text of their paper (Dimension.AI).

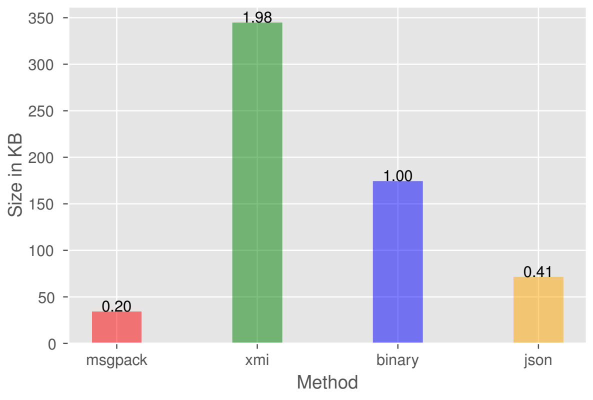

Figure 12: Serialized document payload size by method.

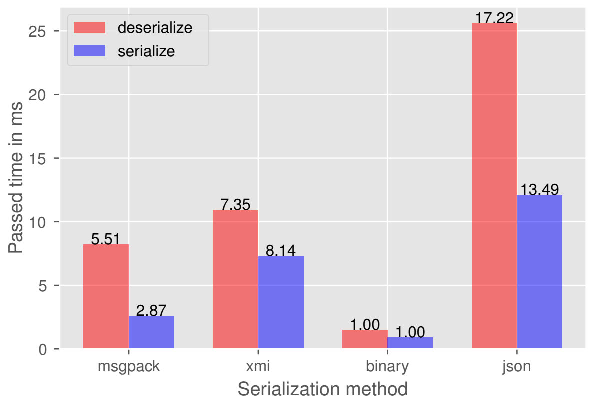

Figure 13: Serialize and deserialize performance by method, the binary serialization is set as baseline to determine the factors for the other methods.

```java
int iWorkers = 2; // define the number of workers
JCas jc = JCasFactory.createJCas(); // A empty CAS document is defined.
// load content into jc ...

DUUILuaContext ctx = LuaConsts.getJSON(); // Defining LUA-Context for communication
// Definition of an LUA sandbox that restricts LUA to using only the listed classes
ctx.withSandbox(new DUUILuaSandbox().withAllowedJavaClass("java.lang.String")
    .withAllowedJavaClass("java.nio.charset.StandardCharsets")
    .withAllowedJavaClass("org.apache.uima.fit.util.JCasUtil")
    .withAllowedJavaClass("Taxon"));

// Defining a storage backend based on SQlite.
DUUISqliteStorageBackend sqlite = new DUUISqliteStorageBackend("loggingSQlite.db")
.withConnectionPoolSize(iWorkers);

// The composer is defined and initialized with a standard Lua context as well with a storage backend.
DUUIComposer composer = new DUUIComposer().withLuaContext(ctx)
                        .withScale(iWorkers).withStorageBackend(sqlite);

// Instantiate drivers with options (example)
DUUIDockerDriver docker_driver = new DUUIDockerDriver().withTimeout(10000);
DUUIRemoteDriver remote_driver = new DUUIRemoteDriver(10000);
DUUIUIMADriver uima_driver = new DUUIUIMADriver().withDebug(true);
DUUISwarmDriver swarm_driver = new DUUISwarmDriver();

// A driver must be added before components can be added for it in the composer. After that the composer is able to use the individual drivers.
composer.addDriver(docker_driver, remote_driver, uima_driver, swarm_driver);

// A new component for the composer is added
composer.add(new DUUIDockerDriver.
    Component("gnfinder:latest")
    .withScale(iWorkers)
    // The image is reloaded and fetched, regardless of whether it already exists locally (optional)
    .withImageFetching());

// Adding a UIMA annotator for writing the result of the pipeline as XMI files.
composer.add(new DUUIUIMADriver.Component(
               createEngineDescription(XmiWriter.class,
                       XmiWriter.PARAM_TARGET_LOCATION, sOutputPath,
               )).withScale(iWorkers));

// The document is processed through the pipeline. In addition, files of entire repositories can be processed.
composer.run(jc);
```

Figure 14: A lightweight implementation of a example pipeline with DUUI based on Docker images as well as on XMI writer.

```lua
-- Bind static classes from java
StandardCharsets = luajava.bindClass("java.nio.charset.StandardCharsets")
Taxon = luajava.bindClass("Taxon")

-- This "serialize" function is called to transform the CAS object into an stream
    that is sent to the annotator (Analyse Engine running in a docker container)
-- Inputs:
-- - inputCas: The actual CAS object to serialize
-- - outputStream: Stream that is sent to the annotator, can be e.g. a string,
    JSON payload, ...
function serialize(inputCas, outputStream)
    -- Get text from CAS, other methods are possible i.e. all Tokens, all Lemmas,
        etc.
    local doc_text = inputCas:getDocumentText()

    -- Encode data as JSON object and write to stream
    outputStream:write(json.encode({
        text = doc_text
    }))
end

-- This "deserialize" function is called on receiving the results from the
    annotator that have to be transformed into a CAS object
-- Inputs:
-- - inputCas: The actual CAS object to deserialize into
-- - inputStream: Stream that is received from to the annotator, can be e.g. a
    string, JSON payload, ...
function deserialize(inputCas, inputStream)
    -- Get string from stream, assume UTF-8 encoding
    --local inputString = luajava.newInstance(Taxon, inputCas)
    --print(inputStream)
    local inputString = luajava.newInstance("java.lang.String",
        inputStream:readAllBytes(), StandardCharsets.UTF_8)

    -- Parse JSON data from string into object
    local results = json.decode(inputString)

    -- Add taxons
    for i, tax in ipairs(results["taxons"]) do
        if tax["write_token"] then
            print("-----------------")
            local taxon_anno = luajava.newInstance("Taxon", inputCas)
            taxon_anno:setBegin(tax["begin"])
            taxon_anno:setEnd(tax["end"])
            taxon_anno:setValue(tax["text"])
            taxon_anno:addToIndexes()
        end
    end
end
```

Figure 15: Lua example for (de-)serialization XMI or accessing a UIMA document via Lua. In this example the text is taken from an annotator via *serialize* and sent to the associated component. Afterwards, the result of the analysis will directly sent to the invoker via the *deserialize* function creating directly - in this example - taxon.

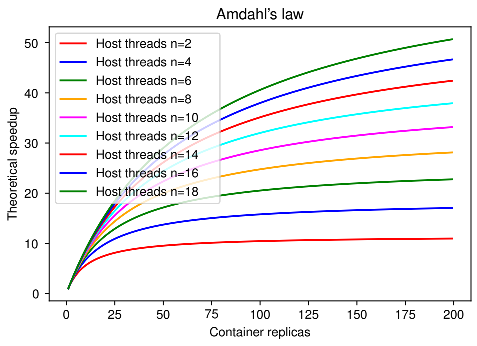

Figure 16: Parameter space of Amdahl's law for spaCy.

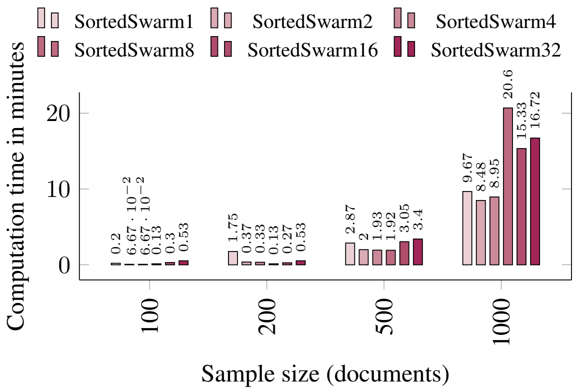

Figure 17: Unlike Figure 8, a sample with the same number but very small documents was selected.