ELIT: The Cloud-based Research Platform Where Anyone Can Deploy Their Models

Gary Lai
Computer Science
Emory University
Atlanta, GA 30322
gary.lai@emory.edu

Bonggun Shin
Computer Science
Emory University
Atlanta, GA 30322
bonggun.shin@emory.edu

Jinho D. Choi
Computer Science
Emory University
Atlanta, GA 30322
jinho.choi@emory.edu

Abstract

This paper demonstrates a new cloud-based research platform called ELIT (Evolution of Language and Information Technology) that enables researchers to build their models for large scale computing using rich resources in the cloud. This platform makes three major contributions to the community. First, it allows researchers to request NLP output for a large amount of raw text using a web API, which can be called by any programming languages. Second, it provides an open space for developers to deploy their models so the entire community can be benefited from the most up-to-date techniques. Third, it integrates NLP components developed in different programming languages into one united framework, which provides a seamless experience of combining these components into one pipeline. The ELIT framework is currently up and running and provides SDK for Python and Java to support both the end users and the developers.

1 Introduction

The Evolution of Language and Information Technology (ELIT) project presents a cloud-based research platform for anyone to develop their models using rich resources in the cloud. We realize that out of all great models produced by many dedicated researchers, only few of them are used by people outside of their communities, by no means that less popular models are of a low quality, but because of several other reasons.

First, it is difficult for independent researchers or ones from *not so famous* organizations to advertise the greatness of their models, despite the fact that several conferences are designed for this purpose. Most prestige conferences do not value as much of practical ideas but of novel ideas, which sometimes are confused by *complicated* ideas. Evidence has been found that models based on simple ideas can perform as well as ones based on complicated ideas in practice; nonetheless, these models tend not to get enough credits in academic research.

Second, it is unwieldy for a single researcher to develop an end-to-end system that takes raw data and produces the requested output while the end-to-end system is often the only kind that users want. Let us say, you have developed a parser faster and more accurate than any. Even with its greatness, the chance of others using your parser is slim because people would expect your tool to take plain text and produce the parsing output, whereas you model is trained on annotated text that needs to be processed by several other NLP tools. As a result, your model does not gain as much attention as it should. Third, most state-of-the-art models using deep learning these days tend to consume much more resources than models traditionally used in NLP. This makes researchers with no access to power machines not being able to develop state-of-the-art models such that alternative solutions such as cloud computing need to be consulted for developing those models.

The objective of the ELIT project is to support both end users and developers with this platform independent framework so that the developers have a central place to deploy their latest models yet are not restricted by certain programming languages to develop those models, whereas the end users have access to all these models through a web API using any programming environments of their choices. We believe that this framework will greatly enhance the integration of all the state-of-the-art models and will facilitate the development of more advanced models by taking advantage of the models already integrated into the ELIT framework.[1]

[1] ELIT: https://elit.cloud

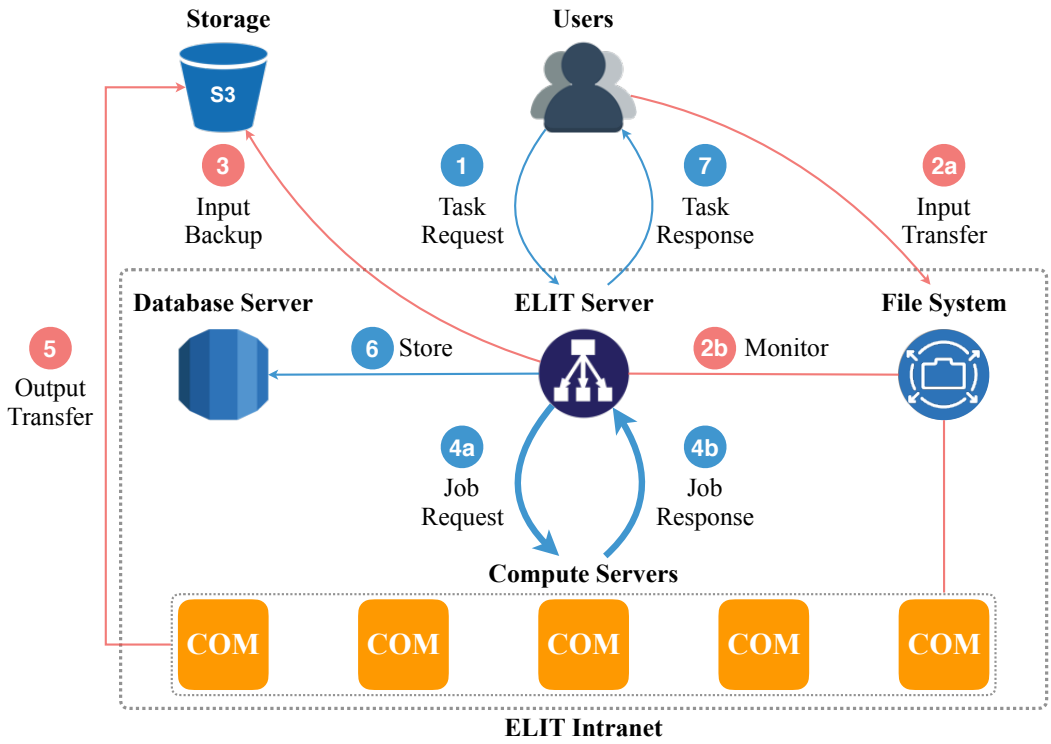


Figure 1: The stepwise overview of the ELIT framework.

2 ELIT Framework

Figure 1 describes a stepwise overview of the ELIT framework. ELIT supports two types of users, unregistered and registered, to prevent malicious use. ① An unregistered user can make a task request to the ELIT server using an HTTP request where the body of the request contains input text comprising up to 10^5 characters (roughly 100KB in disk space). Given the requested task, ④ the ELIT server dispatches jobs to one or more compute servers that process appropriate NLP components for this task. When the task is completed, ⑥ it stores all relevant information to the database server and ⑦ responses to the user with the expected output.

A registered user can request a task using a file containing input text up to 10^9 characters (roughly 1GB in disk space). When a file task is requested, ② the ELIT server guides the user to upload the file to its internal file system and monitors the progress. Once the file is uploaded, ③ the ELIT server backs up the input file to public storage so the user can have access to the file later. While the file is being transferred, ④ the ELIT server dispatches jobs to the compute servers. ⑤ The compute server then saves the expected output to the public storage, and ⑥ all relevant information is stored to the database server. Finally, ⑦ the user is notified by an email that includes a remote path to the output file.

The intranet includes the ELIT server (Section 2.1), the compute servers (Section 2.2), and the database server (Section 2.4), where the ELIT server and the compute servers share the file system (Section 2.3). Users can connect to this framework only through the ELIT server via HTTP. The public storage gives a read access to the users for all the input and output files generated for their tasks. The ELIT framework is developed on the Amazon Web Services (AWS), but can be easily imported to any other cloud computing environment.

2.1 ELIT Server

The ELIT server is the entry point to this framework and the central hub that connects all the other servers together. It is currently running on a t2.large EC2 instance with 2 CPUs and 8GB of RAM. The primary role is to keep connections between users and the framework using the HTTP. It adapts the web-application framework called *Ruby on Rails*,[2] widely used to build web servers. It is also responsible for dispatching jobs for the task, where each job is handled by an ordered list of NLP components in a compute server, managing data transfer to the public storage and the database server, load balancing to handle a large number of tasks, and securing all connections via encryption and authentication.

[2]Ruby on Rails: http://rubyonrails.org

2.2 Compute Server

Every compute server takes an ordered list of jobs from the ELIT server, runs the appropriate NLP component for each job, and generates the output. Two types of compute servers are developed, one for Java- and the other for Python-based NLP components, where each server adapts the *Spring* and the *Flask* framework, respectively.[3] Currently, both compute servers are running on r4.xlarge instances with 4 CPUs and 32GB of RAM, which will be exported to more powerful machines upon the launch.

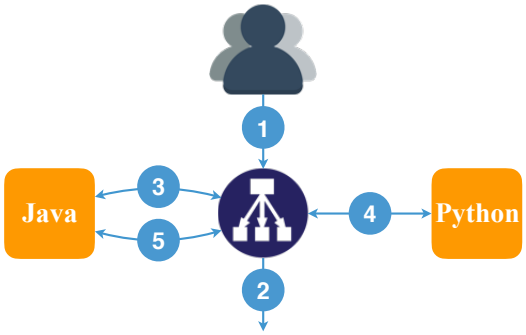


Figure 2: An example of job dispatching.

Figure 2 illustrates how NLP jobs are dispatched. ① When a task is requested, ② the ELIT server figures out the pipeline of jobs required to complete the task, e.g., [J_1, J_2, P_3, P_4, P_5, J_6]. It then groups those jobs with respect to the compute servers such that ③ [J_1, J_2] are dispatched to the Java compute server and the output of those jobs are sent to the ELIT server. Then, ④ [P_3, J_4, J_5] are dispatched to the Python compute server along with the output of the first two jobs, and the output of these three jobs is sent back to the ELIT server. Finally, ⑤ [J_6] is dispatched again to the Java compute server along with the output of the first four jobs, then the output of the last job is sent to the ELIT server and merged with the other outputs. The network between these servers uses 25 Gbps of bandwidth. Dispatching jobs in groups minimizes the network traffic, which significantly enhances the overall speed of the task.

2.3 File System

The ELIT server and all the compute servers share the same file system from the Amazon Elastic File System (EFS), which allows them to transfer multiple GBs of data per second for an unlimited amount. This file system is used to temporarily store input files for the compute servers to process locally.

[3]Spring: `spring.io`, Flask: `flask.pocoo.org`

2.4 Database Server

For each task, the database server stores relevant information such as the remote paths to the input and output files in the storage, timestamps and logs from each job, user profile information, etc. It uses the Amazon Relational Database Service (RDS) and adapts the *PostgreSQL*.[4]

3 Components

3.1 NLP Pipeline

What distinguishes ELIT from other frameworks the most is its seamless connections between NLP components developed in different programming environments so that it provides a true platform independent experience to developers and end users. For a demonstration, we integrate components from two of popular NLP tools, NLP4J and spaCy, which are developed in the Java and Python environments, respectively.[5] Additionally, two more components developed by the ELIT team are integrated to test out the scalability of this framework. Table 1 shows the NLP pipelines provided by these tools.

Tool	Lang.	Pipeline
`nlp4j`	Java	`tok` → `pos` → `lem` → {`ner`,`dep`}
`spacy`	Python	`tok` → `pos` → `dep` → `ner`
`elit`	Python	`tok` → `sent`

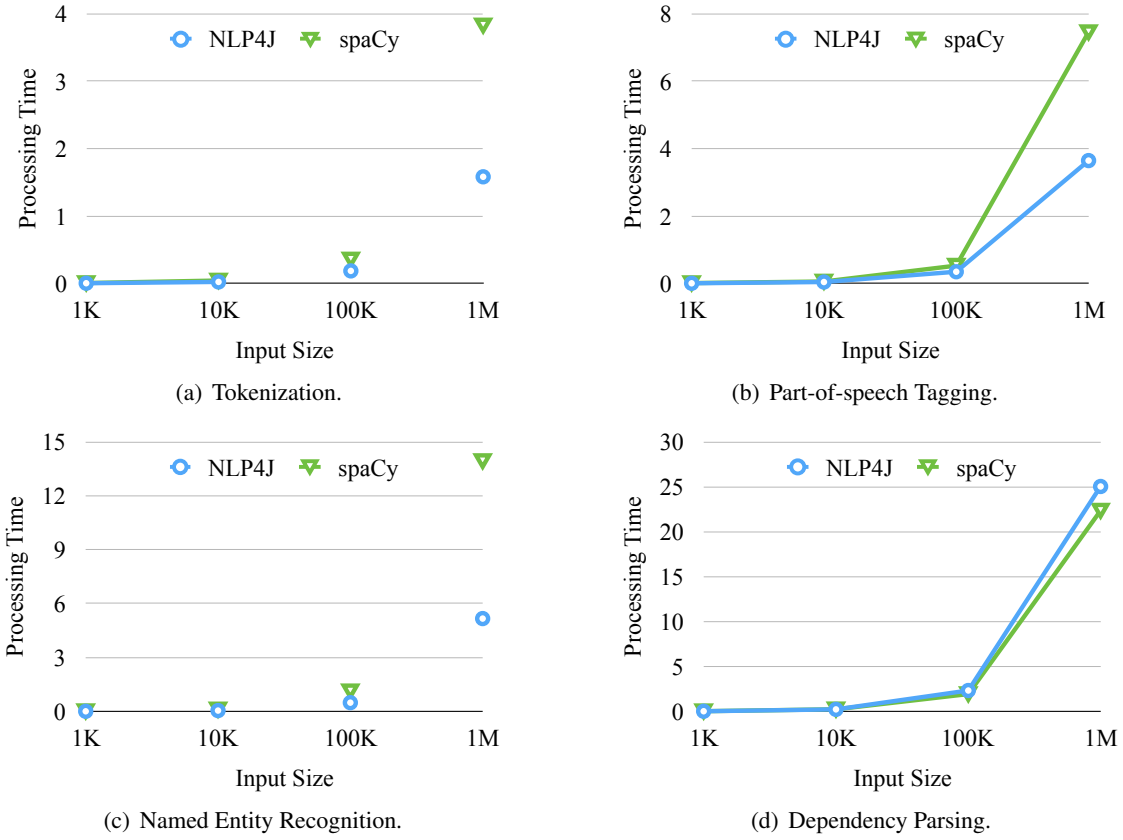
Table 1: The NLP pipelines provided by three tools. `tok`/`pos`/`lem`/`ner`/`dep`/`sent`: tokenization, part-of-speech tagging, lemmatization, named entity recognition, dependency parsing, and sentiment analysis.

NLP4J and spaCy are chosen because they provide similar pipelines covering several tasks in NLP, as well as APIs that make it easy to modulate those components from their pipelines. As benchmarked by Choi et al. (2015), these tools implement two of the fastest dependency parsers, which often are the bottlenecks in NLP pipelines; comparisons between their individual and combined performance using ELIT highlights the strength of this framework that makes this integration possible.

3.2 Benchmarks

Figure 3 shows comparisons between NLP4J and spaCy with respect to four NLP tasks, tokenization, part-of-speech tagging, named entity recognition, and dependency parsing. The input data of sizes 1K, 10K, 100K, and 1M are used, where each data

[4]PostgreSQL: `https://www.postgresql.org`
[5]NLP4J: `emorynlp.github.io/nlp4j`, spaCy: `spacy.io`



(a) Tokenization.

(b) Part-of-speech Tagging.

(c) Named Entity Recognition.

(d) Dependency Parsing.

Figure 3: Speed comparisons between NLP4J and spaCy for different sizes of input data.

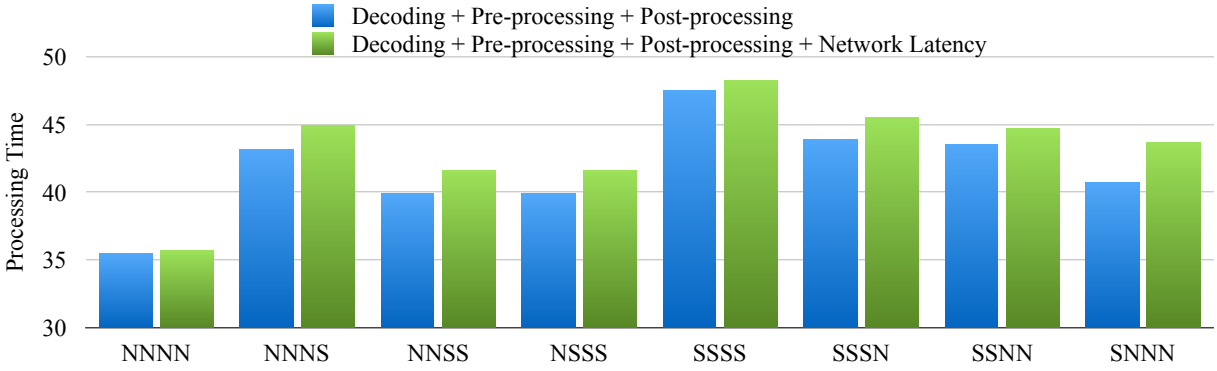


Figure 4: The overall times for decoding, pre-processing, post-processing, and network latency, taken by various combinations of components in NLP4J (`N`) and spaCy (`S`) to process input data of size 1M. The network latency implies the time it takes to transfer data between the ELIT and compute servers.

contains raw text from random Wikipedia articles. The processing time is measured in seconds, which includes the decoding time as well as pre- and post-processing times to handle I/O for each tool. Given the input data $\leq$ 100K, both tools show competitive speeds for all tasks. However, NLP4J outperforms spaCy to handle large data for the first three tasks, and shows similar speeds for dependency parsing.

Figure 4 shows the total time that each pipeline takes to process the input data of size 1M, where the pipeline includes components from either NLP4J or spaCy, and performs all four tasks in Figure 3 sequentially, which is represented by four characters. For instance, `NNSS` implies a pipeline consisting of the first two components, a tokenizer, and a part-of-speech tagger, from NLP4J and the last two components, a named entity recognizer, and a dependency parser, from spaCy. The fastest pipeline is achieved when all components are from NLP4J, `NNNN`, which shows almost no network latency. The slowest is when all components are from spaCy (`SSSS`). It is possible to achieve a relatively fast speed by replac-

ing its tokenizer with the one from NLP4J, NSSS, which makes the second fastest pipeline. It is worth mentioning that marginal differences are observed between the processing times with and without the network latencies, which implies that it is possible to achieve a pipeline almost as fast as (or even faster than) the one comprising all components from the same tool. This brings out complete freedom for researchers to develop tools in any languages, and integrate them into this framework so they can take advantage of all the other components in ELIT in order to create their own pipelines.

4 Software Development Kit

4.1 Input Request

ELIT provides a Software Development Kit (SDK) that allows users to send raw text and receive the requested output from various combinations of NLP components in the framework using web-API. The following shows a request containing the input text (`input`), the end task (`dep`), and the tool of the component specified that the user wants (`spacy`):

```
request = {
  "input": "Hello World! Welcome to ELIT.",
  "task": "dep",
  "tool": "spacy"
}
```

When only the end task is specified, ELIT figures out the most optimized pipeline to get the requested output, in this case, the tokenizer (`tok`) and the part-of-speech tagger (`pos`) from `spacy`. It is possible to specify custom components for the earlier stages in the pipeline. The following shows an example specifying `tok` from `elit` and `pos` from `nlp4j`:

```
request = {
  "input": "Hello world! Welcome to ELIT.",
  "task": "dep",
  "model": "spacy",
  "dependencies": [
    {
      "task": "tok",
      "tool": "elit"
    },
    {
      "task": "pos",
      "tool": "nlp4j"
    }]}
```

All the tools and components described in Table 1 are supported in ELIT. Additionally, a special task called `all` can be specified, in which case, all components within the same tool will be used to generate the output. In the following example, both the tokenizer (`tok`) and the sentiment analyzer (`sent`) from `elit` are used for the requested pipeline:

```
request = {
  "input": "Hello world! Welcome to ELIT.",
  "task": "all",
  "tool": "elit"
}
```

4.2 Python

The following Python code requests the pipeline using all components in `nlp4j` for the input data through HTTP, and prints the output:

```
import requests

request = {
  "input": "Hello world! Welcome to ELIT.",
  "task": "all",
  "tool": "nlp4j"
}

url = "https://elit.cloud/api/public/decode"
r = requests.post(url, json=request)
print(r.text)
```

No external library is needed to make this API call. The following shows the output from the request:

```
{
  "output": [
    {
      "sen_id": 0,
      "tok"   : ["Hello", "world", "!"],
      "lem"   : ["hello", "world", "!"],
      "pos"   : ["UH", "NN", "."],
      "ner"   : [],
      "dep"   : [(1,"intj"),(-1,"root"),...],
    },
    {
      "sen_id": 1,
      "tok"   : ["Welcome, "to, "ELIT", "."],
      "lem"   : ["welcome, "to, "elit", "."],
      "pos"   : ["VBP", "IN", "NNP", "."],
      "ner"   : [(2, 3, ORG)],
      "dep"   : [(-1,"root"),(2,"case"),...],
    }],
  "pipeline": {
    "tok":"nlp4j","lem":"nlp4j","pos":"nlp4j",
    "ner":"nlp4j","dep":"nlp4j","dep":"nlp4j"
  }}
```

The output is formated in JSON so it can be comprehensible for most researchers. The same output can be retrieve by using the API provided in our SDK, which can be installed via the standard package manager for Python (`pip`):

```
!pip install elitsdk

from elit.sdk.api import Client
c = Client()
print(c.decode(request))
```

4.3 Java

Since ELIT uses the generic HTTP protocols, the web API can be called by any language. ELIT also provides the SDK for Java, which can be installed via the standard package manager (`maven`):

```
<dependency>
    <groupId>cloud.elit</groupId>
    <artifactId>elit</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

Given the Java SDK, the following code retrieves the same output in Section 4.2:

```
import cloud.elit.sdk.api.Client;
import cloud.elit.sdk.api.TaskRequest;

Client c = new Client();
String input = "Hello world! Welcome to ELIT.";
TaskRequest request
      = new TaskRequest(input, "all", "nlp4j");
System.out.println(c.decode(request));
```

4.4 Deployment

Not only for end users, the Python and Java SDKs provide templates for developers as well, allowing them to create components compatible to the ELIT framework. Both the Python and Java SDKs give the abstract class called `Component` that guides the developers to implement the following two methods, which are used to decode input data and load any pre-trained model for the component:

```
@abc.abstractmethod
def decode(self, input_data, *args, **kwargs):
    pass

@abc.abstractmethod
def load(self, model_path, *args, **kwargs):
    pass
```

Note that since this is an early stage of the project, our SDK is still getting actively developed. Please visit the project website to check the status of the latest updates in the SDK.[6]

5 User Interface

ELIT also provides a web interface that allows registered users to access the web API. Once registered users are logged into the ELIT website, they can make a task request using either through HTTP or a file, and retrieve the NLP output using this interface. This is convenient for those who need a quick access to the ELIT framework.

[6]ELIT SDK: https://elit.cloud/sdk

6 Future Development

In the ELIT framework, each instance communicates with the others through HTTP. Transferring data inside the ELIT intranet is about 500-600MBs per second. However, when a compute server proceeds with large data, the ELIT server cannot keep the HTTP connection and wait for the result from the server because it times out before the process finishes. To resolve this problem, we are currently developing an asynchronous job queue system, also called delayed job, for each of the compute servers. This delayed job queue is already implemented on the ELIT server, but not on the compute servers.

We will also create a callback API endpoint on the ELIT server to receive the callback from the computer servers. This callback API will wait for the message from the compute server after the job is done. In this case, when the task dispatcher from the ELIT server sends jobs to the compute server, the connection can be closed once the compute server receives the message from the ELIT server. Based on this mechanism, we can avoid the timeout happens in the HTTP protocol.

Acknowledgments

We gratefully acknowledge the support of the AWS Machine Learning Research Awards. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of Amazon Web Services (AWS).

References

Jinho D. Choi, Amanda Stent, and Joel Tetreault. 2015. It depends: Dependency parser comparison using a web-based evaluation tool. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*. Beijing, China, ACL'15, pages 387–396.