A Lightweight NLP Workflow Engine for CLARIN-BE

Adriaan Lemmens
Centre for Computational Linguistics
KU Leuven, Belgium
adriaan@ccl.kuleuven.be

Vincent Vandeghinste
Instituut voor de Nederlandse Taal
Leiden, the Netherlands
vincent.vandeghinste@ivdnt.org

Abstract

This paper presents our work in progress on building a flexible workflow engine. The architecture of the engine is based on the message queue programming model and is implemented in Python. NLP tools are exposed as remotely executable tasks and wrapped in standard abstractions. The main contribution is to provide a unofirm API for defining various kinds of tasks and workflows.

The use case of the library is building a text analytics platform for digital humanities, providing a usage-friendly interface to predefined workflows.

1 Introduction

CLARIN's infrastructure boasts a wealth of tools for Natural Language Processing (NLP) that, while individually capable, are non-trivial to combine. This is because each tool makes its own assumptions about the design of its interface (whether this be a GUI or an API endpoint), the data formats it supports, and whether it processes documents individually or in batches. Users who require the combined functionality of multiple tools face the daunting task of verifying whether tools' input and output formats are compatible and, moreover, of manually coordinating the execution of these tools. To remedy this, we are developing a new text analytics dashboard application (Section 3) that bundles existing tools into curated but configurable automated annotation workflows, which are invoked from within a cohesive, user-friendly environment. The responsibility of interoperating disparate tools is no longer the responsibility of individual users, but now moves to the application backend.

Tackling the complexity of flexible tool interoperability has led us to design and implement our own custom NLP workflow engine.[1] Like alternatives, our engine assumes an agreed-upon vocabulary of abstractions for defining multi-step data processing logic (aka 'workflows'), but focuses in particular on the needs of text processing. The engine is implemented as a Python library with a carefully designed API intended for use by non-core contributors, e.g. computational linguists at other research groups, giving them a clear path for registering new tools and workflows. The design of its architecture is informed by the finite deployment and maintenance resources of research infrastructure. Its core consists of a minimum of moving parts, most of them third-party open-source; when idling, its footprint is minimal; it is easy to set up and interact with locally.

2 Related Work

There is no shortage of open-source workflow engines for data pipelining in general. Popular options include Apache Airflow,[2] Prefect,[3] and Argo.[4] These are each backed by a committed community and/or commercial entity and have well-tested APIs. A first downside is that, in order of mention, they rely partly

This work is licenced under a Creative Commons Attribution 4.0 International Licence. Licence details: http://creativecommons.org/licenses/by/4.0/

[1] The engine is developed separately from the dashboard application, but its design is informed by the latter's requirements.

[2] https://airflow.apache.org/

[3] https://www.prefect.io/

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

to fully on Kubernetes[5] for deployment.[6] We prefer to avoid a dependency on Kubernetes at this stage to avoid added complexity.[7] A second downside to these options is that, being generic workflow engines, they offer more flexibility than we need, yet their learning curves are steep. Our ideal workflow API is more limited, so that non-core maintainers need only learn a few abstractions to get started. Limitations here also ensure that contributed tasks and workflows behave more predictably. Of course, by rejecting these 'standard' options, we risk some degree of wheel reinvention, e.g. with respect to functionality such as error handling and logging as well as overall fault-tolerance. We also take on more responsibility in terms of maintaining good documentation and growing a community where fellow contributors can ask questions. The effort these last two entail ought not to be underestimated.

There appears to be less community energy for workflow engines that focus on a specific domain, such as NLP. Most effort is concentrated at the level of individual libraries, e.g. spaCy[8] (Honnibal et al., 2020) and Stanza[9] (Qi et al., 2020), where pipeline components are assumed to implement the library's abstractions.

Within CLARIN, the most established counterpart to our work is the *WebLicht* environment (Hinrichs et al., 2010), which has continued to receive updates over the years. For a user, WebLicht is a web application for chaining together linguistic annotation services and applying these chains to uploaded texts. This functionality is similar but not identical to that of our text analytics dashboard application (Section 3). At the architectural level, each linguistic annotation tool in WebLicht's inventory is exposed as a stateless HTTP web service.[10] WebLicht services process data within the life cycle of a single HTTP request, meaning requests must be conservative about both the number of input items (i.e. documents) they batch together as well as total item size. Annotation services can be implemented in any programming language in principle; however, the language of choice appears to be Java. Services are registered not in one central database, but across separate repositories hosted at CLARIN-D research centers. They are described by CMDI[11]-formatted metadata, which allows for them to be aggragated automatically into a central inventory. The accompanying metadata also includes information about input data and output data schemas, allowing annotation pipelines to be validated for correctness, an idea that we have borrowed. In Section 4 we indicate a number of important design differences between our system and WebLicht.

A second CLARIN project with similar goals is the distributed parallel NLP processing architecture built by CLARIN-PL (Walkowiak, 2017). This powers *ws.clarin-pl.eu* and implements the LMPN language for planning processing pipelines and orchestrating services. It has been in use for many years and according to on-page statistics it has processed millions of requests by services and applications built on top of it. Like Weblicht, it is an open source project[12]. From the code base, we glean several architectural similarities with our system, which we indicate in Section 4.

3 Use Case: Text Analytics Platform for Digital Humanists

The primary client of our workflow engine is a text analytics platform for digital humanists, which we are simultaneously developing. This future contribution to CLARIN-BE's infrastructure is essentially a dashboard application that lets non-technical users manage corpora, execute carefully curated analysis workflows, visualize results, and export annotated datasets to one of several standard formats.

An important difference between *Manatee* and an environment such as *WebLicht* (see Section 2) is that users are not expected to define workflows themselves, that is, starting at the level of individual tools and chaining them together.[13] Instead, users choose from an assortment of workflow 'presets' and configure them to suit their needs. From the point of view of our workflow engine, these presets are full-fledged

[5] `https://kubernetes.io/`

[6] Granted, Airflow can be deployed without Kubernetes, but such a setup nevertheless entails many moving parts.

[7] While Kubernetes is the de facto standard for container orchestration, affording great power and flexibility, it has a steep learning curve, its passive compute requirements are higher, and maintaining a cluster is extra work.

[8] `https://spacy.io/`

[9] `https://stanfordnlp.github.io/stanza/`

[10] `https://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/WebLicht_in_Detail`

[11] `http://www.clarin.eu/cmdi`

[12] `https://gitlab.clarin-pl.eu/nlpworkers`

[13] WebLicht does offer predefined processing chains in its so-called 'Easy Mode', albeit it less prominently.

workflows behind the scenes. Our main aim in taking this approach is to shield user interfaces from complexity.

4 Workflow Engine Architecture

The architecture of our workflow engine is based on the message queue[14] (MQ) programming model. Decoupled services communicate among each other by posting and pulling messages to/from a centralized queue data structure. In our particular case, the messages themselves contain instructions to execute a unit of work (or *task*) in the consuming service (*worker*). The effect is a form of remote procedure calling (RPC).

Our system is implemented as a Python library. Each worker service in our system is a containerized Python process that exposes one or more tools (CLARIN-affiliated or other) as remotely executable tasks. Tools are wrapped in standard abstractions defined by the library. Both clients and workers are aware of these abstractions, simplifying interoparability, as components are able to introspect each other's metadata and all components respect certain fixed contracts of behavior. The main contribution of the library is to provide a uniform API for defining various kinds of tasks as well as workflows. The API has been designed carefully so as to be easy-to-use for computational linguists in an academic setting, who by and large write code at an intermediate level and are relatively inexperienced with software engineering in general. A few important principles guiding the API's design are 1) that users can rely on their IDE to spot mistakes (for the same reason, we prefer metadata to live in-code rather than in, e.g. ad hoc YAML schema), 2) that testing locally is easy, and 3) that no knowledge of the system as a whole is required.

While CLARIN-PL's engine (Section 2) implements its own logic for interacting with a message queue, our framework delegates instead to the very established third-party Python library *Celery*[15]. As a result, we get at no extra cost functionality such as task planning, logging, fault recovery, best practices, and monitoring (not to mention extensive documentation) The most complex and crucial component of our system is thus in a sense offloaded to the open source community.

Another third-party library that we rely on is JinaAI's *DocArray*[16], which provides data structures for modeling documents and batching them together. All workflow steps are defined to operate on (possibly large) batches of documents, making techniques such as clustering and sentence alignment possible. Memory usage is not a concern, since DocArray provides on-disk versions of its document collection abstraction with random read and write access. The only data exchanged by workflow steps are pointers to the location of these on-disk datasets in shared storage.

Our engine requires Python 3.6 or greater[17]. Note, however, that tools implemented in other languages or in older versions can still be wired up via the Python API by wrapping them in subprocess calls.

Architecturally, our engine assumes one or more worker services and any number of clients, all of them communicating in the vocabulary of abstractions defined by our Python library. Additional essential engine components are 1) shared storage, where workflow step outputs are cached and 2) a message broker (in our case, Redis). Contributors of workflows and tasks do not define worker services directly. Instead, they instantiate a *Toolset* object and register tasks and workflows with it. Typically, a single Toolset should correspond to an independent code repository. Each Toolset must specify a Dockerfile specification of the required environment. Toolsets connect to the rest of the engine through a separate *Worker* abstraction (which non-core contributors need not know of). There are several kinds of tasks, or *Operator*s, that can be registered with a Toolset, each with their own function signature. The most common is the *Processor*, which takes a batch of documents, assigning one or more annotations to each document. Tasks are registered by decorating ordinary Python functions with a decorator method of the Toolset object, e.g. `@toolset.processor()`. These Python functions wrap CLARIN tools in their body. These tools are run on demand only. Toolset authors are recommended to keep actions that impact memory usage (e.g. loading a large model) inside the function body and rely on garbage collection to

[14] `https://en.wikipedia.org/wiki/Message_queue`
[15] `https://docs.celeryq.dev/en/stable/`
[16] `https://github.com/jina-ai/docarray/`
[17] Because we rely on Python type hints.

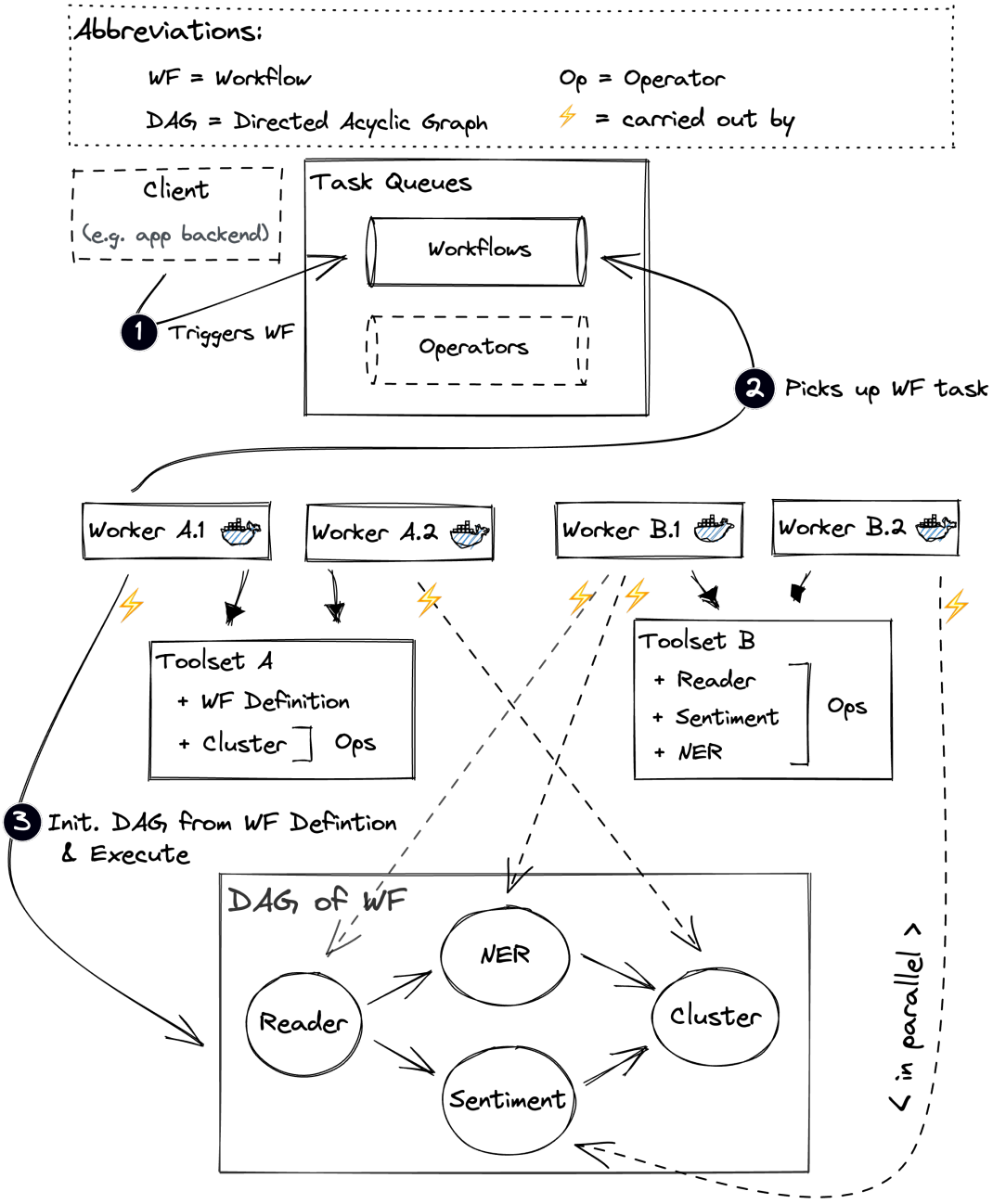

Figure 1: Life cycle of a workflow

release memory. It is the additional responsibility of the author of the task-wrapped function to implement the logic of transforming corpus data so as to match the input and output formats of the tool in question. Task metadata is provided through the Python API. Among other things, metadata is used by the engine to validate whether certain tasks are mutually compatible.

Workflow definitions are registered with Toolsets in much the same way as are tasks, i.e. with a corresponding Python decorator. Our workflow definitions can describe not only sequential data flow logic – aka 'chains'/'pipelines' – but also more general *Directed Acyclic Graphs*[18] (DAG). The advantage of a DAG is that it allows for parallelization in the execution graph. A workflow defined in one Toolset can invoke Operators from a different Toolset, so long as both Toolsets are registered with the workflow engine. Moreover, because workflow exection logic is part of the core library shared by all worker services, there is no need for a central workflow orchestrator.

Our approach differs from WebLicht's (Section 2) in that a) the workflow steps in our system can handle long-running batch jobs; b) the tools to which steps delegate run only when needed, c) our engine supports not only pipelines (serial step execution, aka 'chains' in WebLicht's terminology), but also more advanced execution graphs with parallel branches; d) metadata and configuration are always specified through code. As for CLARIN-PL's engine, which follows a similar task-as-message queue model to ours, we note that a) our workflow tasks are more strict and explict with respect to the types of input they accept; b) partly due to (a), our engine is less flexible; c) whereas CLARIN-PL's engine separates functions cleanly across workers, our workers expose bundles of functionality (as Toolsets). CLARIN-PL is able to set precise resource usage limits on their workers and can scale horizontally more easily; by contrast, we need to take special care to ensure that workers don't process too many messages at once, so as to avoid memory contention within the same worker (and possible deadlocks as a result).

A schematic overview of the life cycle of a workflow invocation is shown in Figure 1.

5 Early Results & Project Status

Our workflow engine is still undergoing active development and the API is far from stable. Nevertheless, early (and tentative) results suggest that our approach has merit. We are able to invoke a large number of workflows at once: once the task queue becomes saturated, remaining invocations are simply queued up. This is acceptable for our needs, since there are no system components that require low latency, e.g. a pending HTTP request somewhere, and users of our dashboard application (*Manatee*) will be conditioned through UX to expect longer processing times.

An alpha release of *Manatee* is planned for December 2022, with a beta release planned for Spring 2023. Around the same time, all code repositories will be made publicly available. After a v1.0 release mid-2023, incremental improvements will continue into 2024.

6 Discussion

At least two of our design choices warrant a critical look. First, our choice for a batch processing model, where a batch corresponds to an entire corpus, has the consequence of making it harder to predict the duration of a workflow step. The larger the input data, the longer a step will take to finish processing it. Without restrictions on corpus size, this can lead to a single workflow job holding a worker service hostage, causing slowdowns for other jobs. Our current solution to this is to deploy replicas of each worker service, insofar as our compute budget permits.

Second, because each worker service bundles multiple operators of varying memory requirements, and supposing that we permit workers to spawn multiple operator processes in parallel, it is hard to predict how much memory a worker will use at any given time. In order to keep the memory usage of the system as a whole reasonable, we set resource limits on the containers hosting the worker services based on observed behavior. An alternative solution, which is unappealing at the moment, is to differentiate between 'heavy' and 'light' operators and associate the heavier sort with a separate queue that lets fewer tasks through at once.

[18] `https://en.wikipedia.org/wiki/Directed_acyclic_graph`

References

Erhard W. Hinrichs, Marie Hinrichs, and Thomas Zastrow. 2010. WebLicht: Web-Based LRT Services for German. In *Proceedings of the ACL 2010 System Demonstrations*, pages 25–29.

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

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A Python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 101–108, Online. Association for Computational Linguistics.

Tomasz Walkowiak. 2017. Language processing modelling notation - orchestration of NLP microservices. In Wojciech Zamojski, Jacek Mazurkiewicz, Jaroslaw Sugier, Tomasz Walkowiak, and Janusz Kacprzyk, editors, *Advances in Dependability Engineering of Complex Systems - Proceedings of the 12th International Conference on Dependability and Complex Systems DepCoS-RELCOMEX, July 2-6, 2017, Brunów, Poland*, volume 582 of *Advances in Intelligent Systems and Computing*, pages 464–473. Springer.