RDF-Connect: A declarative framework for streaming and cross-environment data processing pipelines

Arthur Vercruysse[1,*], Jens Pots[1,*], Julián Rojas Meléndez[1,*] and Pieter Colpaert[1,*]

[1]IDLab, Department of Electronics and Information Systems, Ghent University – imec

Abstract

Data processing pipelines are a crucial component of any data-centric system today. Machine learning, data integration, and knowledge graph publishing are examples where data processing pipelines are needed. Furthermore, most production systems require data pipelines that support continuous operation and streaming-based capabilities for low-latency computations over large volumes of data. However, creation and maintenance of data processing pipelines is challenging and a lot of effort is usually spent on ad-hoc scripting, which limits reusability across systems. Existing solutions are not interoperable out-of-the-box and do not allow for easy integration of different execution environments (e.g., Java, Python, JavaScript, Rust, etc), while maintaining a streaming operation. For example, combining Python, JavaScript and Java-based libraries natively in a single pipeline is not straightforward. An interoperable and declarative mechanism could allow for continuous communication and integrated execution of data processing functions across different execution environments. We introduce RDF-Connect, a declarative framework based on semantic standards that enables instantiating pipelines with data processing functions across execution environments communicating through well-known communication protocols. We describe its architecture and demonstrate its use for an RDF knowledge graph creation, validation and publishing use case. The declarative nature of our approach facilitates reusability and maintainability of data processing pipelines. We currently support JavaScript and JVM-based environments but we aim to extend RDF-Connect support to other rich ecosystems such as Python and to lower-level languages such as Rust, to take advantage of system-level performance gains.

Keywords

Data pipeline, RDF, Streaming, Interoperability

1. Introduction

Modern data-centric software systems are built supported on complex data processing operations. The design and implementation of these operations are organized into data pipelines, which are sequences of data processing components where the output of one component is the input of the next, thus enabling smooth data flows towards a common goal [1]. Data pipelines are present at the core of data-dependent tasks such as data science, where pipelines are used for acquisition, curation and analysis of data [2]; machine learning (ML), where pipelines support the preparation, training, validation and cleaning of data [3]; knowledge graph construction and publishing, where pipelines are used to generate semantic annotations from heterogeneous sources, validate (e.g., with SHACL) and load data into a graph database [4].

Recently, the need for streaming data pipelines has increased due to the demand for low-latency computations. Production systems often need to process high volumes of data at high velocity, rendering traditional batch processing methods insufficient [5, 6]. Batch processing systems suffer from latency issues since they need to collect input data into batches before it can be processed any further [7]. Streaming data pipelines are designed to continuously process data as soon as possible, which is required for (near) real-time analytics, monitoring and reacting to changes in data sources. Internet of Things (IoT) scenarios are a typical example where streaming data pipelines are essential [8].

SofLiM4KG'24: Software Lifecycle Management for Knowledge Graphs Workshop, co-located with ISWC'24, 12th November 2024, Baltimore, USA

*Corresponding author.

arthur.vercruysse@ugent.be (A. Vercruysse); jens.pots@ugent.be (J. Pots); julianandres.rojasmelendez@ugent.be (J. Rojas Meléndez); pieter.colpaert@ugent.be (P. Colpaert)

https://julianrojas.org (J. Rojas Meléndez); https://pietercolpaert.be (P. Colpaert)

0000-0003-3467-9755 (A. Vercruysse); 0000-0002-6645-1264 (J. Rojas Meléndez); 0000-0001-6917-2167 (P. Colpaert)

Despite how much data pipelines can aid on data management challenges through automation, monitoring, fault detection, etc; modeling and implementing data pipelines often require significant effort and expertise [1]. Moreover, data pipelines are often implemented as ad-hoc scripts [9] which are error-prone and difficult to maintain and reuse across different systems [10]. A myriad of data pipeline frameworks exist, each with its own strengths and weaknesses [11, 7, 12, 13]. However, most of these frameworks are not interoperable out-of-the-box (or only provide support for a few other specific ones) and do not allow for easy integration of different execution environments. For instance, combining Python, JavaScript and Java-based data processing libraries natively (i.e., each executed in their own native environment) in a single pipeline is not straightforward. This is particularly important, for example, in the case of RDF knowledge graph processing pipelines, where despite the maturation of RDF libraries (e.g., Apache Jena, RDFLib, RDF-JS-based libraries, etc), robust implementations of well-defined operations, like RDF generation or SHACL validation, are still scarce and are only available for few languages. This raises the need to connect the best implementations for each operation seamlessly, and while shell scripts can facilitate this, they are difficult to maintain and reuse.

The need for multilingual pipelines also arises when certain critical parts of a pipeline demand substantial resources. A pipeline should manage the required load efficiently, If a component is not efficient, it should be replaceable with a more performant version in a different language. The challenge lies not in handling the data across different programming languages but in connecting these languages effectively. Conversely, not all pipeline components are critical and may adhere to different standards. Interacting with a complex internal API to aggregate data may be more easily implemented using the company's primary languages, leveraging existing SDKs and expertise. In general, a pipeline should not mandate a particular language for specific tasks.

Designing an interoperable and declarative framework that allows for easy description, continuous communication and integrated execution of data processing functions across different execution environments, is the main goal of this work. The framework should allow for clear separation of concerns (i.e., high-level workflow definition from step-level implementation and deployment details), while focusing on the reusability of data pipelines [11], that could allow for easy deployment of new and similar data pipelines with minimal effort and adjustments. We also aim on facilitating testing, benchmarking and comparison of different data processing tools implemented in different languages.

With that goal in mind, we set out to create a declarative streaming pipeline architecture to address the previously mentioned challenges.

1. **Streaming:** The architecture should focus on streaming pipelines, allowing for continuous data processing.
2. **Multi-lingual:** The architecture should not be built around a single language and should abstract away language specifics.

In this paper, we introduce RDF-Connect, a declarative, multilingual and streaming pipeline framework. The framework defines a simple RDF vocabulary and architecture that abstracts languages by using language-specific *Runners*, which are able to communicate over language-agnostic *Channels*. These channels enable the streaming of data as messages, facilitating the construction of streaming pipelines. The pipeline and its components (*Processors*) are described using SHACL shapes, which allows to declare the expected input and output data types and constraints of each processor. To demonstrate the fulfillment of the proposed requirements, we describe a real use case, which consists of a pipeline that annotates, validates and loads incoming sensor data into an RDF triple store. Currently RDF-Connect supports JavaScript and JVM-based environments. In the future, we aim to extend it to support rich environments such as Python and lower-level languages such as Rust, to take advantage of system-level performance gains.

The remainder of the paper is structured as follows. First, we present an overview of related work, noting the ubiquity of pipelining frameworks. Next, we describe RDF-Connect, discussing various design decisions and the provision of multiple runners to support multi-lingual pipelines. The subsequent section4 elaborates on an existing pipeline used in the field, addressing some of the hurdles that were overcome. The final sections cover the conclusion and future directions.

2. Related work

In this section, we present an overview related works that address the requirements, characteristics, and challenges of data processing pipelines.

2.1. Data pipelines

Foidl et al. [10] provide a comprehensive overview of data pipelines, starting with a definition where a data processing pipeline can be understood from a theoretical perspective, as a directed acyclic graph (DAG) composed of a sequence of nodes that process data; and from a practical perspective, as a piece of software that automates the manipulation of data and moves them from diverse source systems to defined destinations [10]. The authors also define a generic architecture for data pipelines, which consists of three main components:

1. **Data Source:** The origin of the data, which can be a database, a file, a message queue, or a Web API.
2. **Data Processing:** The transformation of the data, which can include filtering, aggregation, enrichment, and validation.
3. **Data Sink:** The destination of the data, which can be other pipelines, an application or external storage systems.

Lastly, they mention a classification of data pipelines based on their processing characteristics, such as processing mode (batch or streaming), data flow (ETL (Extract-Transform-Load) or ELT (Extract-Load-Transform)), and use case (visualization, analysis tools, ML and deep learning applications, or data mining) [10].

Similarly, Matskin et al. [11] provide a survey of big data pipeline orchestration tools. They propose a set of criteria for evaluating pipeline tools and frameworks, emphasizing reusability, flexible pipeline communication modes, and separation of concerns. The authors use these criteria to analyze 37 different tools for big data pipeline orchestration. They also note that few tools support a graphical input language for the description of pipelines, and they conclude that Apache Airflow (open-source), Argo Workflow and Snakemake (closed-source) are the best suited frameworks for their needs, despite Airflow failing to fullfil the separation of concerns criterion and lacking support for streaming processing [11]. Among the criteria, the authors do not directly consider the support for multilingual pipelines, however the step-level containerization could be seen as a way to support different languages. Thanks to its language-oriented *Runners*, RDF-Connect can support native multilingual execution without resorting to containerization, although pipeline-level containerization can be used for facilitating dependency management.

Mbata et al. [12] also provide a survey of pipeline tools for data engineering. In this survey, pipelines are also categorised according to their characteristics, such as (i) ETL/ELT pipelines, (ii) data integration, ingestion and transformation pipelines, (iii) orchestration and workflow management pipelines, and (iv) ML pipelines. Apache Spark-based pipelines are highlighted as a popular choice for ETL/ELT data processing with support for Scala, Java, Python and R, although its performance is compromised when used for large pipelines. Apache Kafka and Apache NiFi are mentioned as streaming integration tools. Kafka provides interfaces for multiple languages and is streaming oriented but entails a steep learning curve. NiFi is user-friendly and supports live streaming pipelines, but is mainly Java-based. For orchestration and workflow management, Apache Airflow and Apache Beam are highlighted. Apache Airflow and Apache Beam primarily concentrate on running pipelines defined in a single language across different execution environments, with a particular emphasis on big data. Although Beam supports multilingual pipelines, its implementation is highly verbose. Beam relies on job servers to execute parts in a different language which has to be set up before starting the pipeline. The connections to these job servers are complex to code and challenging to modify. On the other hand, Apache Airflow does not support declarative pipelines, streaming data, nor multilingual pipelines (being mainly Python-oriented).

Related works focused on streaming pipelines include the work of Isah et al. [7] which surveys distributed data stream processing frameworks such as Apache Flink, Apache Storm, Apache Spark

Streams, and Apache Kafka Streams. They define a taxonomy for these frameworks based on their characteristics. For instance the programming model (native or micro-batch) and the type of transformations supported per record (Map, Filter and FlatMap). RDF-Connect may be considered as a native data stream processing framework with support for all types of transformations per record. The work of Dias et al. [14] also provides a survey of stream processing frameworks with a focus on resource elasticity (i.e., the capacity to automatically scale resources based on the workload). The authors highlight the need for high-level abstractions to facilitate the development of stream processing applications.

Some works that relate to workflow standards include the work of Dehury et al. [15], which introduces a data pipeline architecture for serverless platforms, based on the OASIS TOSCA (Topology and Orchestration Specification for Cloud Applications) standard[1]. The authors implement the proposed architecture using Apache NiFi components. The Common Workflow Language (CWL) is an open standard designed to describe the execution of command-line tools and their integration into workflows[2]. Its main components are command-line tools, which are essentially wrappers around commands (such as `ls`, `echo`, `tar`, etc.). CWL allows for the chaining of these tools to form workflows. Workflows themselves can also have inputs and outputs and can be nested within each other. The primary strength of CWL lies in its ability to chain the output of one tool to the input of another. Workflows described in CWL can be executed using different CWL runners. These range from the most basic `cwltool`[3], which runs workflows locally, to more advanced distributed computing platforms such as AWS and Azure via Arvados[4] and others[5].

Other related works include the work of Dessalk et al. [16] which proposes and implements an approach for big data workflows based on Docker containers which communicate using the message-oriented middleware KubeMQ. The authors also define a Domain Specific Language (DSL) for the description of workflows, although it seems not to be used in their implementation. In principle, this approach could be used to support multilingual pipelines, however the authors only implement individual data processing tasks using bash scripts. Lastly, Agrawal et al. [17] introduce RHEEM, a cross-platform data processing framework that decouples applications from underlying platforms. This tool allows to define data processing tasks from a limited number of mapping operators, which are then split and assigned to different execution environments based on a cost model. Although, this approach does not qualify as a general pipeline framework, it provides an interesting approach for multilingual and cross-platform data processing. A continuation of this work is currently being developed as an incubating Apache project called Apache Wayang [18].

2.2. Semantic-related data pipelines

Linked Pipes [19, 20] (LP) was developed to provide a more user-friendly approach to creating Linked Open Data pipelines. Its primary objectives are to extract data, transform it into Linked Data, and load it into RDF triple stores. The tool features a web-based GUI that facilitates the easy construction and debugging of pipelines. Pipelines and processors in Linked Pipes are defined and configured using RDF. Each pipeline has a dereferenceable IRI, allowing it for example, to be imported into another LP-ETL instance. Linked Pipes is also built on the JVM, and all processors are implemented in JVM languages. While extending processors is possible, it demands considerable effort. Engineers also may find it challenging to adapt the generated RDF configurations for processors. Linked Pipes emphasizes the transportation of Linked Data between processors. It includes predefined processors such as extractors, transformers, loaders, as well as quality assessment and special processors. Quality Assessment processors ensure that messages are processed correctly and can halt the pipeline if issues are detected. Special processors can execute remote commands, adding to the system's flexibility. The main drawbacks of Linked Pipes is its lack of support for multilingual pipelines and its batch-based

[1] https://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf
[2] https://www.commonwl.org/
[3] https://github.com/common-workflow-language/cwltool
[4] https://arvados.org/
[5] https://www.commonwl.org/implementations/

processing paradigm.

Grassi et al. [21] introduced *Chimera*, a semantic data transformation and integration tool based on Apache Camel. Camel is a Java-based integration framework that provides a set of predefined components, that can be extended for custom tasks. Camel can support multilingual pipelines through WebAssembly, which adds an additional layer of complexity. Although Camel can also support streaming-based operations, the authors relied on batch processing for Chimera. Guash et al. [22] presented a pipeline for RDF knowledge graph construction that uses Apache Airflow workflow as a workflow orchestrator together with Celery[6]. The authors argue in favor of using Airflow due to its scalability, the capacity of defining tasks in code and being open source. Another example of a semantic framework for pipelines is *UnifiedViews* [23], a batch-based ETL framework with native support for RDF data, that allows wrapping existing Java libraries as *data-processing units* and compose pipelines using a graphical user interface. The open source version of UnifiedViews seems not to be maintained anymore, in favor of a commercial version integrated to the PoolParty Semantic Suite[7].

Bonte et al. [24] presented a survey of stream reasoning systems and specify a lifecycle model for Streaming Linked Data. The lifecycle model consists of 6 stages:

1. **Name:** Identify data streams with an IRI.
2. **Model:** Use a data model that accounts for both data and metadata.
3. **Shape:** Define the smallest unit of data in a stream (e.g., triple-based or graph-based).
4. **Annotate:** Transform non-RDF data into RDF.
5. **Describe:** Include interoperable metadata for discovery purposes.
6. **Serve:** Define format and protocol for data sharing.
7. **Query:** Consume the data stream via a querying process.

Although this work does not provide a specific pipeline framework, the lifecycle model can be used to define a set of processors for a streaming pipeline. Klironomos et al. [25] introduced *ExeKGLib*, a Python library to build ML pipelines described as a knowledge graph. The library is able to generate the corresponding Python scripts from the knowledge graph description, based on a set of data science operations (e.g, feature engineering, visualization, model training, etc). Pipeline executions are done in a batch-based manner and so far, there is only support for Python-based operations. Sicilia et al. [26] introduced an ontology that attaches semantic descriptions to data and analytic transformations that are integrated with data pipeline code. Concretely they apply this approach to data pipelines from the Apache Beam framework. Lastly, Chakraborty et al. [27] present a survey of ETL tools for semantic data integration. It concludes with a set of open challenges for the field, such as the need for automation and visualization tools.

3. RDF-Connect

RDF-Connect defines a declarative and streaming pipeline framework for cross-environment data processing. It aims to enhance interoperability among data processing libraries that excel in specific tasks. This necessity is particularly prominent within the Linked Data ecosystem due to the limited number of robust implementations. RDF libraries are primarily written in three major programming languages: JavaScript, Java, and Python. Developing software at a high TRL (Technology Readiness Level) demands significant effort. To advance the RDF ecosystem, it is inefficient to reimplement the same functionality across all programming languages. Instead, the best implementations should be able to interoperate within a single pipeline. RDF-Connect addresses this interoperability challenge by defining a common vocabulary to describe data processing units and their interactions, and implementing abstraction layers per execution environment that conceals the underlying programming language specifics of a given data processing unit.

[6] a distributed python-based task-queue system: https://github.com/celery/celery
[7] https://www.poolparty.biz/poolparty-unifiedviews

For instance, consider SHACL validation as an example. The task is to determine whether a given RDF graph is validated according to a given SHACL shape. While the SHACL specification clearly defines the requirements for a SHACL engine implementation, providing full specification coverage can be challenging for developing engines from scratch. Performance is another critical factor; the validation step must not become a bottleneck in the pipeline, which is a risk if a validator is implemented in a slower programming language. With RDF-Connect, only a single high-TRL SHACL validation implementation is necessary, so that high performance validation can become possible across different pipelines.

Another example of the value of RDF-Connect is evident when integrating a pipeline into an organisation's infrastructure and some custom code is necessary. This code, being specific to the organisation, is unlikely to be reused by others and thus can be written in a language that facilitates rapid development and suits developers expertise. RDF-Connect mitigates the tension between application speed and development speed.

3.1. General idea

RDF-Connect focuses on three main objectives:

1. The choice of programming language should not impair the pipeline.
2. New components should be easy to build.
3. Pipelines should be able to be validated before running.

To achieve these objectives, RDF-Connect is divided into four components:

1. **Processors:** These are small execution units that perform a single and usually simple task to foster reusability.
2. **Runners:** Runners target a specific execution environment (like Javascript) in which they initiate one or more processors with the provided pipeline configuration. They also construct the configured channels before starting the pipeline.
3. **Channels:** These are abstract entities responsible for data transfers among processors.
4. **Pipeline Configuration:** An RDF document describing various interlinked processor instances working together to accomplish a task.

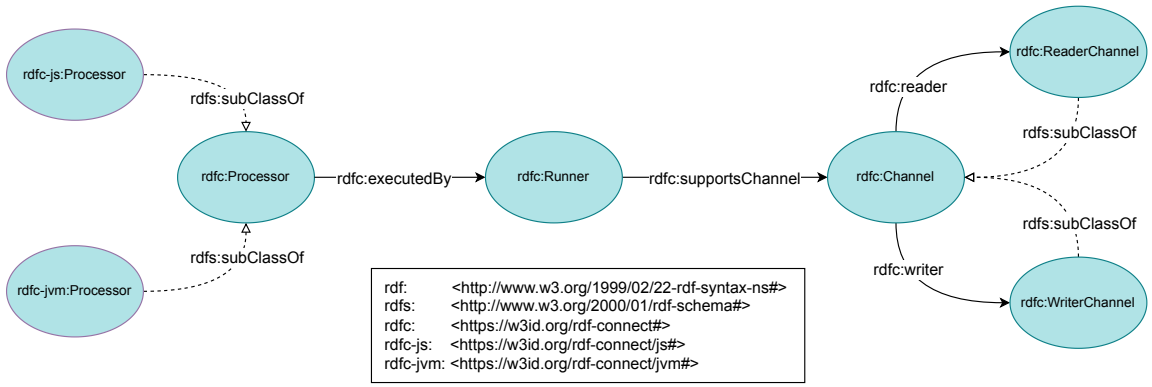

Figure 1: Main classes of the RDF-Connect vocabulary. Two language-specific processor definitions are shown for JavaScript and JVM-based environments, for which corresponding implementations are currently available.

Figure 1 shows the main concepts of the RDF-Connect data model, published online at https://w3id.org/rdf-connect. Language-specific *Processors* are *executed by* language-specific *Runners*, which *support* specific types of streaming communication *Channels*. SHACL shapes[8] accompany concrete processor and channel definitions providing descriptions for input and output parameters and their

[8]not visible in figure 1

constraints. The vocabulary can be easily extended to define coverage for additional execution environments and communication means. Programming language barriers are overcome by splitting logic and sending/receiving data over channels. Channels are programming language-independent and based on well-known communication protocols, such as HTTP request/response interactions, since most languages provide support for sending HTTP messages and starting an HTTP endpoint. However, channel logic is abstracted from processors so that each processor acts on event-driven messages regardless of how those messages are delivered. Each type of channel has its advantages and disadvantages. HTTP channels are easy to set up and relatively fast but lack replay-ability and fault tolerance. Kafka streams, for example, are another option that offers replay-ability and logging, offering a more feature-rich type of channel to connect processors. For consecutive processors written in the same language, in-memory channels can be used share data efficiently. Note that this does not compromise the first objective; all pipelines remain configurable, although some configurations may offer some trade-offs in terms of performance. As mentioned before, processors are channel-agnostic to ensure maximum reusability, allowing the pipeline manager to choose the optimal channel for each step. A complete list of supported channels is described in Section 3.2.

Creating a new processor is language-dependent and should be approached from the corresponding runner's perspective. For instance, to run a JavaScript processor, only the JavaScript file and the processor function are required. This simplicity facilitates the creation of new processors, as each processor merely needs to describe the file, the function and annotate the required parameters. This information ensures consistent and reliable initiation of the processor. Each processor should be accompanied by a SHACL shape, which runners can use to determine the processor's parameters. SHACL shapes enable the validation of a pipeline before it is executed. Additionally, these shapes and processor configurations allow for the derivation of provenance information from pipelines.

A pipeline is runner-agnostic, which makes it possible to refer to a pipeline, without knowing the underlying implementation, or the underlying runners. All information required for the runner to start a processor is already present in the processor definition. Listing 1 shows a pipeline with three processors of type *rdfc-js:Send*, *rdfc-js:Append* and *rdfc-jvm:Print*. The purpose of the pipeline is to send the message `"Hello"` from the first processor to the second, which appends the message with `"World"` and sends it to the third processor, which in turn prints the message in the console. This pipeline should be started with the JVM and the JavaScript runner. The runners expands all *owl:import* statements to find the required processor definition files, both locally and remotely. Each runner starts all processors that are defined against that runner. When everything is started, the application runs.

3.2. JavaScript runner, processors and channels

As mentioned before, a runner is the abstraction layer between the executing processors and the complete pipeline that constitutes the application. It handles parsing arguments, setting up channels and providing useful feedback when the execution fails. We implemented a JavaScript runner that can execute a JavaScript function as a processor. The code is open source and available with an MIT license[9].

This was our first runner and gave us initial experience in understanding what channels were useful. We implemented the following channels for the JavaScript Runner:

- **HTTP:** The HTTP channel is the main channel used for crossing language barriers. The only configuration the HTTP channel needs is a port and a host to start the endpoint. When the channel is set up, messages can cross programming language barriers.
- **Websocket:** The WebSocket channel is very similar to the HTTP channel but might improve throughput slightly. Note that web sockets are bidirectional, but the reverse direction is not used, as this would break channel-agnostic implementations.
- **File:** The file channel is an interesting channel. It allows communication over files. Writing to this channel is writing a file and the reader part of the channel can use file watcher to detect changes.

[9] https://github.com/rdf-connect/js-runner/tree/rdf-connect-paper

```
@prefix rdfc: <https://w3id.org/rdf-connect#>.
@prefix rdfc-js: <https://w3id.org/rdf-connect/js#>.
@prefix rdfc-jvm: <https://w3id.org/rdf-connect/jvm#>.
@prefix owl: <http://www.w3.org/2002/07/owl#>.

<> owl:imports <./processor/send.ttl>,
    <./processor/append.ttl>,
    <./processor/print.ttl>,
    <./ontology.ttl>.

[ ] a rdfc-js:JSChannel;
    rdfc:reader <jsr>;
    rdfc:writer <jsw>.
<jsr> a js:JSReaderChannel.
<jsw> a js:JSWriterChannel.

[ ] a rdfc:HttpChannel;
    rdfc:reader <hr>;
    rdfc:writer <hw>.
<http-r> a rdfc:HttpReaderChannel;
    rdfc:httpPort 3333.
<http-w> a rdfc:HttpWriterChannel;
    rdfc:httpEndpoint "http://localhost:3333";
    rdfc:httpMethod "POST".

[ ] a rdfc-js:Send;
    rdfc-js:msg "Hello";
    rdfc-js:sendWriter <jsw>.

[ ] a rdfc-js:Append;
    rdfc-js:inputReader <jsr>;
    rdfc-js:payload "World";
    rdfc-js:outputWriter <http-w>.

[ ] a rdfc-jvm:Print;
    rdfc-jvm:input <http-r>.
```

Listing 1: Example pipeline that defines three processors. The processors of type `rdfc-js:Send` and `rdfc-js:Append` communicate with each other using a JavaScript in-memory channel (`rdfc-js:JSReaderChannel` and `rdfc-js:JSWriterChannel`), which suggest that the processors are implemented in JavaScript. The third processor of type `rdfc-jvm:Print` communicates with the processor of type `rdfc-js:Append` using an HTTP channel which uses an HTTP POST to send data. The processors' descriptions are found in the external documents imported by the `owl:imports` property.

A change is interpreted as a new message. This channel is particularly useful for integrating configuration files in an idiomatic way into RDF-Connect.

- **Kafka:** The Kafka channel is the only channel that has replay-ability and fault tolerance. If some part of the pipeline breaks, the messages are not lost but still exist on the Kafka topic created by the channel. It also facilitates to integrate a pipeline with already existing Kafka streams.
- **In-memory:** This channel is the main channel used to let JavaScript processors communicate with each other. It is very fast and easy to use as the message never leaves the JavaScript heap and avoids data copying.

The JavaScript runner requires all processors to denote the location of their implementation file and function name. This gives enough information to import the file and call the function at runtime. Each processor also comes with a SHACL shape denoting the required parameters. In previous iterations, we used the Function Ontology (FnO) parameter mappings for this but it was found too verbose to write and has been deprecated (although some processor description may still use it). Now each processor

gets a single argument built with RDF Lens[10] derived from the SHACL shape. RDF Lens gives great flexibility in the objects that are used as arguments. An example of a processor definition for the JavaScript runner can be found on Listing 2, the corresponding implementation is found at Listing 3.

```
rdfc-js:Resc a rdfc-js:Processor;
    rdfc-js:file <./test.js>;
    rdfc-js:function "resc";
    rdfc-js:location <./>.

[ ] a sh:NodeShape;
  sh:targetClass rdfc-js:Resc;
  sh:property [
    sh:datatype xsd:string;
    sh:path rdfc-js:msg;
    sh:name "msg";
    sh:maxCount 1;
    sh:minCount 1;
  ], [
    sh:class rdfc:ReaderChannel;
    sh:path rdfc-js:rescReader;
    sh:name "input";
    sh:maxCount 1;
  ].
```

Listing 2: Example configuration of JavaScript processor including location, file, function name and required argument shape.

```
export function resc(args) {
  args.input.data((input) => {
    console.log(args.msg, input)
  });
}
```

Listing 3: Example implementation of JavaScript processor.

The JavaScript runner works in three steps, first, it loads the pipeline to determine all configured JavaScript processors and try to extract their arguments. Then each processor is started one by one. Each processor can return a function that will be called when all processors are started, this alleviates race conditions and allows for particular start-up sequences, like connecting with a database. Then all functions returned by processors are called and the pipeline starts its execution.

3.3. Targeting the JVM

The Java ecosystem for RDF contains many mature, performant and feature rich RDF applications and libraries, making it a valuable target for RDF-Connect. We set out to build a runner which bridges these applications with those targeting JavaScript, in order to greatly increase the potential for future pipelines. We have written a proof-of-concept runner using the JVM-compatible language Kotlin[11], capable of executing not only processors written in Kotlin or Java, but essentially all languages which are capable of generating JVM class files, such as Scala.

In contrast to the JavaScript runner, we do not implement our processors as a single function. Instead, we have adopted a more traditional object-oriented approach, defining a common abstract class which all processors must inherit. For example, we have delegated initialization to the extendable constructor, and execution of the main logic into to the abstract `exec` method. Handling state and complex logic is more straightforward in this manner, since many additional methods and fields can be defined as desired. Arguments are passed to the constructor as a simple string-to-object map wrapper, removing

[10] https://github.com/ajuvercr/rdf-lens

[11] The code is open source and available with an MIT license https://github.com/rdf-connect/orchestrator/tree/rdf-connect-paper

the requirement for a specific method signature. Additionally, using Kotlin's reified generic types, we provide a convenient and simple API which allows for type-safe retrieval of arguments at runtime using compiler inferred types.

```kotlin
class Transparent(args: Arguments) : Processor(args) {
  private val input: Reader = arguments["input"]
  private val output: Writer = arguments["output"]

  override suspend fun exec() {
    output.push(input.read())
  }
}
```

Listing 4: A simple Kotlin processor which reads data from an incoming channel and pushes it verbatim to a writer.

Channels are implemented much like their JavaScript counterparts, but instead of relying on promises and callbacks, we make heavy use of the Kotlin Coroutine library. Most of the communication here is implemented using Kotlin's channel primitives, which are highly similar to those more popularly found in the Go language. Interoperability with HTTP, Kafka, and so forth, is simply implemented as a collection of data producers and consumers of those channel primitives. As a result, the actual data source and destination are completely abstracted by a common reader and writer interface.

The design decisions above showcase how developers of new RDF-Connect runners have a large degree of freedom implementation-wise. Despite these differences, declaring processors is still done in a similar manner to the JavaScript runner, including SHACL shapes defining the required arguments. Adding a processor as part of a pipeline is identical to the JavaScript runner, as the pipeline configuration is runner-agnostic.

In its current state, coordinating the two runner must be done manually, since one runner cannot start or terminate another. Executing a pipeline spanning multiple runners typically requires a simple shell script which starts each runner individually, all pointing to the same configuration file. This, as well as other limitations and opportunities, are a key focus of our future work.

3.4. NiFi runner: interacting with other frameworks

Apache NiFi is not a programming language, but a pipelining tool to design an automated dataflow pipeline, with a focus on the ability to operate within clusters, security using TLS encryption, and extensibility. Apache NiFi also has an API, which can be used to list processors and build pipelines. This is enough to build a NiFi runner. In a previous version of RDF-Connect, we built a NiFi runner that can generate processor descriptions based on NiFi processors retrieved with the API. These processors could be used within a pipeline just like any other processor To instantiate those processors, the NiFi runner made the correct API calls on the NiFi cluster. Just like the JavaScript Runner has in-memory channels, the NiFi Runner also had NiFi channels as its main way of communication. Other channels like HTTP, were also supported by instantiating a NiFi template that worked like an HTTP endpoint in other languages. We dropped support for NiFi in newer versions because we had no demand for it within our use cases at the moment. However, it was shown that it was feasible and bringing back support for it remains possible. The original code is still available on github[12].

4. Interoperable pipeline

This section covers a real-world pipeline that loads incoming sensor data into a triple store. The sensor data originates from The Things Network[13], a global collaborative Internet of Things ecosystem using LoRaWAN®. The pipeline and its installation instructions can be found on GitHub[14].

[12] https://github.com/ajuvercr/nifi-runner
[13] https://www.thethingsnetwork.org/
[14] https://github.com/ajuvercr/rdf-connect-paper-pipeline

4.1. Pipeline structure

A visual representation of the pipeline can be seen in Figure 2. The pipeline contains multiple processors working together to achieve the goal. The blue hexagons represent incoming data, while the green hexagon represents the output.

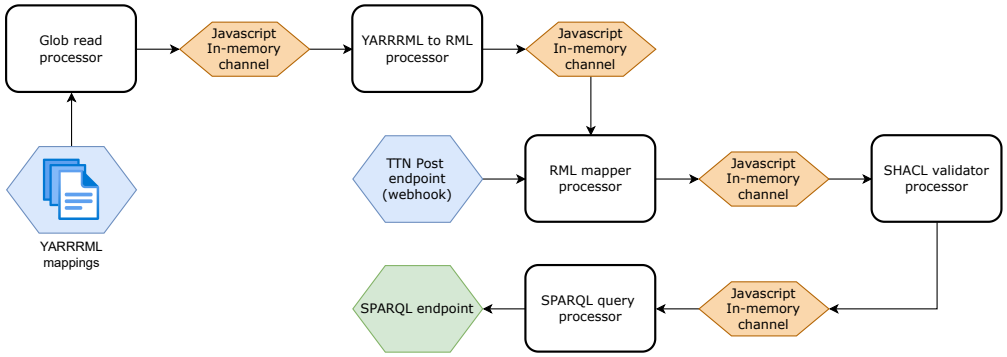

Figure 2: Visual representation of our example pipeline, loading sensor data from The Things Network into a triple store

The first processor creates RML [28] from YARRRML [29] rules, a more user-friendly notation for mappings[15]. The processor is implemented in JavaScript and relies on the yarrrml-parser library written in JavaScript[16]. The YARRRML file is sourced from a `GlobRead` processor, which has a single function: reading files according to a specified glob pattern[17].

The next processor executes an RML mapping for each incoming JSON object received from The Things Network (TTN). This is a JavaScript processor that wraps around the RMLMapper JAR file. Attempts to implement this processor in Java were unsuccessful due to various difficulties using the application code as a library[18]. If the GlobRead processor finds two files, the RML mapper processor executes both mappings.

The third processor checks if the generated RDF is valid according to KWG-SHACL[19]. This step is crucial, as invalid data should not contaminate the triple store. SHACL validation is implemented in multiple languages, and with RDF Connect, it is easy to swap one implementation for another. This pipeline was executed twice: once with a JavaScript processor wrapping around `rdf-validate-shacl`[20], and once with a Java processor wrapping around `org.apache.jena.shacl`[21]. The shape had to be adapted in minor ways to allow for simple RDF generation and allow for differences in the validator processors. For example, the observations are observed by a sensor, but only a stub for the sensor is generated using our simple mappings. This results in invalid objects, but to demonstrate the functionality, we altered the shape file.

Since the JavaScript SHACL validator processor did not exist, we developed a new processor to integrate into the pipeline. The processor is minimal, comprising only 25 lines of JavaScript code and 40 lines of RDF configuration. This experience highlights how straightforward it is to create new processors, reinforcing the practice of building small, reusable processors that perform a single task efficiently.

The final processor is again a custom processor, creating a SPARQL INSERT query from the incoming

[15] https://rml.io/yarrrml/
[16] https://github.com/RMLio/yarrrml-parser/blob/development/README.md
[17] https://en.wikipedia.org/wiki/Glob_(programming)
[18] https://github.com/RMLio/rmlmapper-java
[19] https://github.com/KnowWhereGraph/KWG-SHACL
[20] https://github.com/zazuko/rdf-validate-shacl
[21] https://jena.apache.org/documentation/shacl/index.html

triples, optionally placing the triples in a specific graph and executing the SPARQL INSERT query on a triple store using the `fetch-sparql-endpoint` library.

4.2. Mapping entities

Integrating the pipeline into the *real* world is accomplished using channels. New data is ingested via a POST request at the beginning of the pipeline. With The Things Network, one specifies a webhook that sends a JSON object for each measurement made by sensors. This JSON object contains information about the node that made the observation as well as a *decoded payload*, which is custom for each data logger node. An example of our payload is shown in Listing 5.

```
{
  "battery": 3.3299999237060547,
  "humidity": 40.12785339355469,
  "pressure": 1007.492919921875,
  "temperature": 21.81758689880371,
  "version": 2
}
```

Listing 5: Example of the decoded payload in a TTN message

Our YARRRML mapping file reflects this structure by generating a `sosa:Observation` for each field in the payload separately. To test our pipeline, we intentionally made a mistake in the mapping for *humidity*, creating invalid `sosa:Observations`. When running the pipeline, the configured triple store will not contain any humidity observations as these are rejected by the SHACL validator.

4.3. Swapping a processor

Currently, all processors are JavaScript-based. This simplifies running the pipeline; a single command, `npx js-runner pipeline.ttl`, initiates and executes the pipeline. However, if we are not satisfied with the current SHACL validator (e.g., for performance reasons), it can be easily exchanged by an alternative implementation, such as a Java-based processor mentioned above.

This can be achieved by modifying the configuration to utilize a Java processor instead of the JavaScript processor. The Java runner cannot integrate JavaScript in-memory channels, so the incoming and outgoing channels need to be changed to, for example, HTTP channels. Both the JavaScript runner and the Java runner will establish HTTP endpoints, allowing for message transfer across the language barrier. The concept of the configuration remains unaffected by these changes.

This example, although artificially fabricated, effectively demonstrates the versatility of RDF-Connect. The ability to effortlessly swap out a processor to verify an implementation or replace it with a more performant processor is advantageous. Additionally, file channels exist, which read and write to a file. This feature is invaluable for debugging the pipeline, as it allows for immediate input and output verification by writing to and reading data to/from disk.

5. Conclusion and Future Work

In this paper we introduced our ongoing work on RDF-Connect, a declarative and streaming pipeline framework. We showed how RDF-Connect is able to abstract programming language specifics and enable communication of data processing functions, implemented in different languages over language-agnostic channels. In this way we accomplish our goals of providing support for multi-lingual and streaming pipelines. We also showed an example of how we are applying RDF-Connect in a real use case, where we are able to load and validate incoming sensor data into an RDF triple store.

As previously mentioned, executing pipelines which span multiple runtimes and environments is currently not as simple as launching a simple binary. More importantly, the development of runners targeting new runtimes and languages is also limited due to the high complexity of parsing the configuration file, setting up inter-process communication channels and routing messages. Therefore, we wish

to greatly simplify RDF-Connect by designing and developing a new platform-agnostic orchestrator, which will do much of the heavy lifting currently required by all runners individually. The orchestrator should be responsible for starting the runners, either locally or remote, and facilitate communication between them. This leaves only one channel type without configuration, reducing the complexity of pipeline configurations. Reducing the complexity of the pipeline also reduces the potential for human errors. Finally, individual runners should not need to interpret the pipeline configurations themselves. Rather, we will map the RDF model to a simple and intuitive configuration representation at runtime in order to relief the individual processors of building and querying their own triple store. By doing so, we want to make it significantly easier to bring RDF-Connect to new runtimes and environments, such as low-level languages like Rust, to take advantage of system-level performance gains.

We will also set out to define a standard way of publishing and finding processors, greatly improving convenience and ease-of-use, to facilitate an ever-growing ecosystem of processors.

Acknowledgements

This work was supported by the MAREGRAPH project (id: 101100771) which is co-funded by the European Union under the Digital Europe Programme.

References

[1] A. Raj, J. Bosch, H. H. Olsson, T. J. Wang, Modelling data pipelines, in: 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2020, pp. 13–20. doi:10.1109/SEAA51224.2020.00014.

[2] S. Biswas, M. Wardat, H. Rajan, The art and practice of data science pipelines: A comprehensive study of data science pipelines in theory, in-the-small, and in-the-large, in: Proceedings of the 44th International Conference on Software Engineering, ICSE '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 2091–2103. doi:10.1145/3510003.3510057.

[3] N. Polyzotis, S. Roy, S. E. Whang, M. Zinkevich, Data lifecycle challenges in production machine learning: A survey, SIGMOD Rec. 47 (2018) 17–28. doi:10.1145/3299887.3299891.

[4] U. Simsek, K. Angele, E. Kärle, J. Opdenplatz, D. Sommer, J. Umbrich, D. Fensel, Knowledge graph lifecycle: Building and maintaining knowledge graphs, in: Proceedings of the 2nd International Workshop on Knowledge Graph Construction (KGC 2021) co-located with 18th Extended Semantic Web Conference (ESWC 2021), 2021.

[5] K. Rengarajan, V. K. Menon, Generalizing streaming pipeline design for big data, in: S. Agarwal, S. Verma, D. P. Agrawal (Eds.), Machine Intelligence and Signal Processing, Springer Singapore, Singapore, 2020, pp. 149–160.

[6] T. Hlupić, J. Puniš, An overview of current trends in data ingestion and integration, in: 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), 2021, pp. 1265–1270. doi:10.23919/MIPRO52101.2021.9597149.

[7] H. Isah, T. Abughofa, S. Mahfuz, D. Ajerla, F. Zulkernine, S. Khan, A survey of distributed data stream processing frameworks, IEEE Access 7 (2019) 154300–154316. doi:10.1109/ACCESS.2019.2946884.

[8] A. Shukla, Y. Simmhan, Benchmarking distributed stream processing platforms for iot applications, in: R. Nambiar, M. Poess (Eds.), Performance Evaluation and Benchmarking. Traditional - Big Data - Internet of Things, Springer International Publishing, Cham, 2017, pp. 90–106.

[9] C. Yang, S. Zhou, J. L. C. Guo, C. Kästner, Subtle bugs everywhere: generating documentation for data wrangling code, in: Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering, ASE '21, IEEE Press, 2022, p. 304–316. doi:10.1109/ASE51524.2021.9678520.

[10] H. Foidl, V. Golendukhina, R. Ramler, M. Felderer, Data pipeline quality: Influencing factors, root

causes of data-related issues, and processing problem areas for developers, J. Syst. Softw. 207 (2024). doi:10.1016/j.jss.2023.111855.

[11] M. Matskin, S. Tahmasebi, A. Layegh, A. Payberah, A. Thomas, N. Nikolov, D. Roman, A survey of big data pipeline orchestration tools from the perspective of the datacloud project, in: CEUR Workshop Proceedings, Research Council of Norway (RCN) / 323325 EC/H2020 / 101016835, 2021, pp. 63–78. doi:https://doi.org/10.5281/zenodo.5910891.

[12] A. Mbata, Y. Sripada, M. Zhong, A survey of pipeline tools for data engineering, arXiv preprint arXiv:2406.08335 (2024).

[13] T. R. Rao, P. Mitra, R. Bhatt, A. Goswami, The big data system, components, tools, and technologies: a survey, Knowledge and Information Systems 60 (2019) 1165–1245. doi:10.1007/s10115-018-1248-0.

[14] M. Dias de Assunção, A. da Silva Veith, R. Buyya, Distributed data stream processing and edge computing: A survey on resource elasticity and future directions, Journal of Network and Computer Applications 103 (2018) 1–17. doi:https://doi.org/10.1016/j.jnca.2017.12.001.

[15] C. Dehury, P. Jakovits, S. N. Srirama, V. Tountopoulos, G. Giotis, Data pipeline architecture for serverless platform, in: Software Architecture, Springer International Publishing, Cham, 2020, pp. 241–246.

[16] Y. D. Dessalk, N. Nikolov, M. Matskin, A. Soylu, D. Roman, Scalable execution of big data workflows using software containers, in: Proceedings of the 12th International Conference on Management of Digital EcoSystems, MEDES '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 76–83. doi:10.1145/3415958.3433082.

[17] D. Agrawal, S. Chawla, B. Contreras-Rojas, A. Elmagarmid, Y. Idris, Z. Kaoudi, S. Kruse, J. Lucas, E. Mansour, M. Ouzzani, P. Papotti, J.-A. Quiané-Ruiz, N. Tang, S. Thirumuruganathan, A. Troudi, Rheem: enabling cross-platform data processing: may the big data be with you!, Proc. VLDB Endow. 11 (2018) 1414–1427. doi:10.14778/3236187.3236195.

[18] K. Beedkar, B. Contreras-Rojas, H. Gavriilidis, Z. Kaoudi, V. Markl, R. Pardo-Meza, J.-A. Quiané-Ruiz, Apache wayang: A unified data analytics framework, SIGMOD Rec. 52 (2023) 30–35. doi:10.1145/3631504.3631510.

[19] J. Klímek, P. Škoda, M. Nečaský, Linkedpipes etl: Evolved linked data preparation, in: H. Sack, G. Rizzo, N. Steinmetz, D. Mladenić, S. Auer, C. Lange (Eds.), The Semantic Web, Springer International Publishing, Cham, 2016, pp. 95–100.

[20] J. Klímek, P. Škoda, Linkedpipes etl in use: practical publication and consumption of linked data, in: Proceedings of the 19th International Conference on Information Integration and Web-Based Applications & Services, iiWAS '17, Association for Computing Machinery, New York, NY, USA, 2017, p. 441–445. doi:10.1145/3151759.3151809.

[21] M. Grassi, M. Scrocca, A. Carenini, M. Comerio, I. Celino, Composable semantic data transformation pipelines with chimera, in: Proceedings of the 4th International Workshop on Knowledge Graph Construction (KGC 2023) co-located with 20th Extended Semantic Web Conference (ESWC 2023), 2023.

[22] C. Guasch, G. Lodi, S. V. Dooren, Semantic knowledge graphs for distributed data spaces: The public procurement pilot experience, in: The Semantic Web – ISWC 2022, Springer International Publishing, Cham, 2022, pp. 753–769.

[23] T. Knap, M. Kukhar, B. Macháč, P. Škoda, J. Tomeš, J. Vojt, Unifiedviews: An etl framework for sustainable rdf data processing, in: The Semantic Web: ESWC 2014 Satellite Events, Springer International Publishing, Cham, 2014, pp. 379–383.

[24] P. Bonte, R. Tommasini, Streaming linked data: A survey on life cycle compliance, Journal of Web Semantics 77 (2023) 100785. doi:10.1016/j.websem.2023.100785.

[25] A. Klironomos, B. Zhou, Z. Tan, Z. Zheng, G.-E. Mohamed, H. Paulheim, E. Kharlamov, Exekglib: Knowledge graphs-empowered machine learning analytics, in: The Semantic Web: ESWC 2023 Satellite Events, Springer Nature Switzerland, Cham, 2023, pp. 123–127.

[26] M.-Á. Sicilia, E. García-Barriocanal, S. Sánchez-Alonso, M. Mora-Cantallops, J.-J. Cuadrado, Ontologies for data science: On its application to data pipelines, in: E. Garoufallou, F. Sartori, R. Siatri,

M. Zervas (Eds.), Metadata and Semantic Research, Springer International Publishing, Cham, 2019, pp. 169–180.

[27] J. Chakraborty, A. Padki, S. K. Bansal, Semantic etl — state-of-the-art and open research challenges, in: 2017 IEEE 11th International Conference on Semantic Computing (ICSC), 2017, pp. 413–418. doi:10.1109/ICSC.2017.94.

[28] A. Dimou, M. Vander Sande, P. Colpaert, R. Verborgh, E. Mannens, R. Van de Walle, RML: a generic language for integrated RDF mappings of heterogeneous data, in: C. Bizer, T. Heath, S. Auer, T. Berners-Lee (Eds.), Proceedings of the 7th Workshop on Linked Data on the Web, volume 1184 of *CEUR Workshop Proceedings*, 2014. URL: http://ceur-ws.org/Vol-1184/ldow2014_paper_01.pdf.

[29] P. Heyvaert, B. De Meester, A. Dimou, R. Verborgh, Declarative rules for linked data generation at your fingertips!, in: The Semantic Web: ESWC 2018 Satellite Events, Springer International Publishing, Cham, 2018, pp. 213–217.