K Descriptor Language (KDL)

Daniel Fox, Ivan Sinyagin, James Thomas, Jared Faucher, Matt Kubej, Qanit Al-Syed
Harvard Extension School CSCI E-599
Milestone 3 Report - 5.13.2019

Table of Contents

K Descriptor Language: Expressing KNIME Workflows with a Domain-Specific Language

Abstract

Summary

KNIME is a visual programming tool designed for analysing large amounts of data from a variety of sources, and is used across multiple fields of study. Although such visual tools can be advantageous to some, others may find a text based alternative preferable. This paper proposes a domain-specific language for KNIME workflows called KDL (K Descriptor Language). A Python application, called the KDL compiler, has been developed to convert KNIME archive files to a KDL text file and back. Edits can be made to the KDL text file, and that file can be converted back to a KNIME archive file which can be imported into the application and run as normal. The user can also use templating to build KDL files from scratch. There are several advantages to having a KNIME workflow in a text format. UNIX tools such as sed can be used to make changes faster than in the application itself. Version control tools such as Git and GitHub can be used to track changes and allow collaborative edits. The tool is intended for KNIME users with programming experience and interest in a text based editing tool for workflows.

Availability

View or learn more about the project at: https://github.com/k-descriptor-language/kdl

Introduction and Review of Literature

The Konstanz Information Miner (KNIME) is a modular data analysis tool that is used in a wide range of fields from business to cheminformatics. It is a pipeline environment consisting of nodes and edges, where data is processed or evaluated at each node sequentially. These collections of nodes and edges can be referred to as workflows within the Graphical User Interface (GUI). Although KNIME's GUI is relatively easy to use and approachable for non-programmers, it does have certain drawbacks. Minor and repetitive modifications to existing workflows can be tedious and time consuming using the GUI. Changing a workflow generally requires a combination of clicking on nodes and editing text in several pop-up windows, dragging nodes into and out of workflows, and clicking and drawing connection arrows between nodes. From the perspective of a programmer, all of these could be performed faster and more efficiently with a textual representation of the workflow.

In this paper we propose a domain-specific language called KDL (K Descriptor Language) as an alternative way to edit workflows. A single workflow in KNIME is stored as a collection of several XML files upon export. Even for a programmer familiar with XML and these files in particular, the editing process is only a slight improvement over making edits in the GUI. The KDL compiler (a Python application) takes in this collection of XML files and outputs a single text file in KDL syntax. This single text file represents the entire workflow in a programming language (not a markup language) that is easy to

read and edit. Once modifications are made, the text file can be compiled into a KNIME archive file, and then imported into KNIME where it can be viewed, edited, and run in the GUI as normal. This combination of compilation and decompilation allows a programmer to modify KNIME workflows without using the GUI or manipulating the underlying XML files.

KNIME

As ever larger collections of data have become available for analysis, so has the need for modular data analysis environments (Berthold M.R, *et al.,* 2008). Data pipeline environments are one such type of tool that has seen an increase in popularity due to their ease of use. These types of tools allow the user to read in and manipulate data from multiple sources in a single application. KNIME is one such free and open-source tool that allows for the creation, manipulation, and execution of data analysis workflows. KNIME is used in over 60 countries by data scientists and includes an ever growing community. It has held annual summits since 2010. It buffers its data in an intelligent way to allow for seemingly unlimited process (KNIME, FAQ).

KNIME's interface employs a graphical representation, with workflows consisting of nodes and the connections between them. They begin with nodes which typically read in data from different sources (e.g. files, databases) and end with nodes which output results. Nodes in the middle of the workflow take data from the previous node in the graph, analysis or manipulate the data in some way, and send the resulting data as input to the next node in the workflow (Warr *et al.,* 2012).

Extensions of KNIME

Another reason for its popularity is a result of KNIME being an open-source it offers options for extensions. KNIME is built on top of Eclipse and written in Java, and has an API that allows developers to write new nodes. Each node themself uses Java classes such as DataTable, DataRow, and DataCell (Berthold) to store data. As new types of data analysis emerge in the sciences, new custom nodes can be built in KNIME to perform such analysis. One such analysis type is "next-generation sequencing" (NGS), which is high throughput DNA or RNA sequencing. Jagla *et al.,* 2011 describes such a workflow where new KNIME nodes read SAM files and write BED files. Another set of custom KNIME nodes (Lindenbaum *et al.,* 2011) involve variant cell format (VCF) files, and the ability to filter and manipulate them. There are many more examples of extensions like these, users of KNIME leveraged the API to create many new functional node types for their fields of study. Our extension of the tool goes one step further. It offers an entirely new interface to create and manipulate workflows. Instead of using the Java API to write new nodes, we use the underlying XML to convert a workflow from the GUI to a DSL.

Domain-Specific Languages

A DSL is designed for a specific application domain with the right level of abstraction that is both functional and easily readable (Kosair *et al.,* 2000). They are typically described in contrast to general programming languages (GPL), not because GPLs are harder to use but because the generalized user interface may not be able to describe the content in a given domain as clearly as a DSL. While GPLs can be used for a wide variety of application domains, they typically require more documentation due to their generalized notation. One of the greatest benefits of a DSL is the ease of expressiveness – the ability to

describe a function or process as simply as possible. In most cases what makes a DSL succeed is not just its syntax. It is also important to have a good compiler and be able to connect to a data source, such as a database (Barzdins *et al.,* 2019). It is important for the DSL editor to be able to integrate and work with other parts of a larger system. For example, Monaco, a Pascal-like DSL, offered a similar solution to the domain of machine automation (Prähofer *et al.*, 2007). The authors of Monaco combined visual and non-visual approaches to automate programming. The language was part of the visual environment and resembled a procedural language with limited capabilities required by reactive systems. DSLs come in many varieties and their primary goal is to easily express and integrate with the application domain for which they are intended. In the case of KDL it is intended to easily express nodes and connections within a KNIME workflow, which makes a syntax associated with a graphical database query language a good starting point for research.

Graph Databases and Syntax

Graph and NoSQL databases alike have gained popularity in recent years. Manipulation and transformation of graph data structures have been extensively studied in the database community for graph engines, (Holzschuher, 2013, 2016) such engines can optimize the storage of a graph and expose a language to modify and query it. The Cypher (openCypher reference) query language is one of several declarative language specifications, similar to SQL in relational database world, that several graph databases implement including Neo4j, Redis graph, Agens graph, etc. (openCypher project). Cypher is considered a DSL for graphs, and its syntax is easily readable to database users as it is very similar to SQL. As sets of connected data continue to get larger (e.g., social networks), databases that are specially designed to handle such collections of nodes and edges are needed. Although we looked at Cypher closely as a possible base for KDL, the lack of nesting available within a node definition made it difficult to use. The data for any given KNIME workflow is not large enough to require an external database, and storing the data in memory utilizing Python data structures is sufficient. This greatly improves the ease of use of the application as it avoids planning a dependency on another platform.

Visual Programming Languages

Visual programming languages (VPL) such as KNIME have been a point of discussion for many years. There have been several studies and surveys regarding the cognitive benefits of using a VPL as opposed to a text based language (Whitley *et al.*, 1997). One survey focused on users of a tool called LabVIEW (Laboratory Virtual Instrument Engineering Workbench) which is a dataflow based VPL similar to KNIME. Contradictory survey results comparing LabVIEW to text based languages such as C raised questions regarding whether or not such cognitive benefits exist, and if the benefits to LabVIEW was its functionality and not its visual syntax. VPLs have also been contrasted with text based languages in terms of ease of use and adoption rates. One such article looks at young students learning programming, and argues that using a VPL keeps programming from being “hard and boring” (Repenning, 2017). VPLs do not have to be so complex, and can be as intuitive as a grid. Although VPLs can be more intuitive and easier to learn for non-programmers or novices, they can be cumbersome and overly simplified to someone familiar and comfortable with a text based language. In the specific case of KDL, it would be easier to designate a connection between nodes by simply typing “(n1)-->(n2)” than using the GUI to

click and drag the arrow from one node to another. We believe a seasoned programmer will find advantages in using a text based KDL over the GUI.

What Makes KDL Distinct

Unlike many of the other extensions in KNIME, our tool provides a new way to create and modify workflows rather can create new nodes. The tool is less geared towards providing new analysis tools and more towards ease of use for KNIME users. We have created a DSL that provides new functionality to read, write, and edit KNIME workflows. This DSL improves upon the current XML format because it is easier to read and edit than a verbose markup language. We understand that VPLs provide several benefits to users over text based tools, and are not suggesting KDL should in anyway replace the KNIME GUI. KDL simply provides an alternative option for KNIME users who may have some programming experience and wish to use a text based tool for editing workflows. Providing the opportunity for KNIME workflows to be created and modified in a text format expands the opportunities for automation, collaboration and version control within the developer community.

Implementation

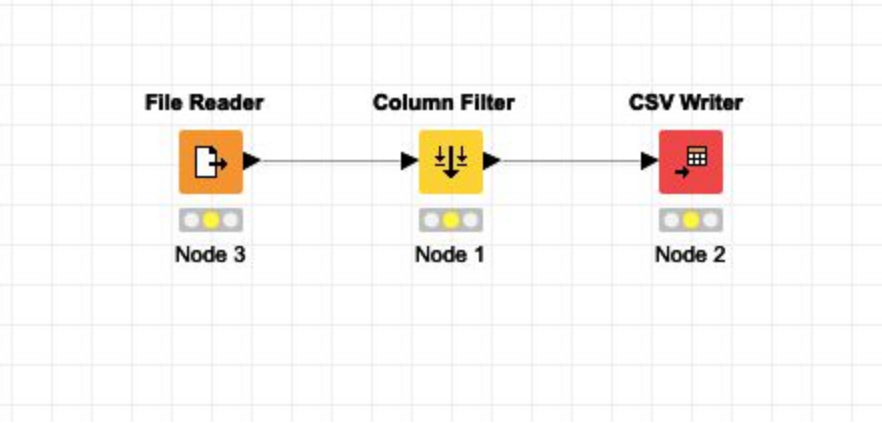


Figure 1: A simple KNIME Workflow Example

KNIME Archive File Makeup

A KNIME workflow is exported as a compressed file with a ".knwf" extension. Within such a file is a folder for each node in the workflow, and a file titled "workflow.knime". The latter is a single XML file that includes the connection data for each pair of nodes. Each of the node folders contains a single XML file titled "settings.xml" which includes all of the data for that node. These XML files contain metadata about KNIME as well as how the node is configured. Figures 2 and 3 provide a comparison of how the node data is represented both in KNIME and in the export XML file.

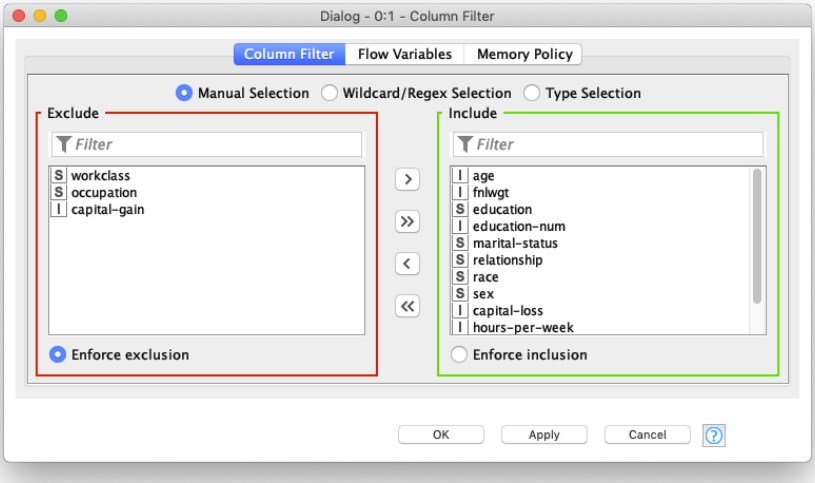


Figure 2: The configuration window of the Column Filter node

```xml
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.knime.org/2008/09/XMLConfig" xmlns:xsi="
http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.knime.org/2008/09/XMLConfig
http://www.knime.org/XMLConfig_2008_09.xsd" key="settings.xml">
    <entry key="node_file" type="xstring" value="settings.xml" />
    <config key="flow_stack" />
    <config key="internal_node_subsettings">
        <entry key="memory_policy" type="xstring" value="CacheSmallInMemory" />
    </config>
    <config key="model">
        <config key="column-filter">
            <entry key="filter-type" type="xstring" value="STANDARD" />
            <config key="included_names">
                <entry key="array-size" type="xint" value="12" />
                <entry key="0" type="xstring" value="age" />
                <entry key="1" type="xstring" value="fnlwgt" />
                <entry key="2" type="xstring" value="education" />
                <entry key="3" type="xstring" value="education-num" />
                <entry key="4" type="xstring" value="marital-status" />
                <entry key="5" type="xstring" value="relationship" />
                <entry key="6" type="xstring" value="race" />
                <entry key="7" type="xstring" value="sex" />
                <entry key="8" type="xstring" value="capital-loss" />
                <entry key="9" type="xstring" value="hours-per-week" />
                <entry key="10" type="xstring" value="native-country" />
                <entry key="11" type="xstring" value="income" />
            </config>
            <config key="excluded_names">
                <entry key="array-size" type="xint" value="3" />
                <entry key="0" type="xstring" value="workclass" />
                <entry key="1" type="xstring" value="occupation" />
                <entry key="2" type="xstring" value="capital-gain" />
            </config>
            <entry key="enforce_option" type="xstring" value="EnforceExclusion" />
            <config key="name_pattern">
                <entry key="pattern" type="xstring" value="" />
                <entry key="type" type="xstring" value="Wildcard" />
                <entry key="caseSensitive" type="xboolean" value="true" />
            </config>
            <config key="datatype">
                <config key="typelist">
                    <entry key="org.knime.core.data.IntValue" type="xboolean" value="false" />
                    <entry key="org.knime.core.data.StringValue" type="xboolean" value="false" />
                    <entry key="org.knime.core.data.BooleanValue" type="xboolean" value="false" />
                    <entry key="org.knime.core.data.DoubleValue" type="xboolean" value="false" />
                    <entry key="org.knime.core.data.LongValue" type="xboolean" value="false" />
                    <entry key="org.knime.core.data.date.DateAndTimeValue" type="xboolean" value="
                    false" />
                </config>
            </config>
        </config>
    </config>
```

Figure 3: The same node represented in XML from the archive file

Figures 2 and 3 show the same column filter node in two different formats. Figure 2 is a screenshot of the "Configure" window within the KNIME GUI. Figure 3 is a screenshot of the exported XML file representing the same node. The XML file contains two tag types, "config" and "entry". An entry tag includes a single point of data, where as a config tag can include entry tags and other nested config tags. The most important config tag in a node XML file has the key value of "model". This tag includes all the configuration details of that node, which are the values represented in the Configuration window in Figure 2 (such as which columns to include and exclude and their names).

Displayed within Figure 4 is the XML file (workflow.knime) showing the connections between nodes.

```
66      <config key="connections">
67          <config key="connection_0">
68              <entry key="sourceID" type="xint" value="1" />
69              <entry key="destID" type="xint" value="2" />
70              <entry key="sourcePort" type="xint" value="1" />
71              <entry key="destPort" type="xint" value="1" />
72              <entry key="ui_classname" type="xstring" value="
                org.knime.core.node.workflow.ConnectionUIInformation" />
73              <config key="ui_settings">
74                  <entry key="extrainfo.conn.bendpoints_size" type="xint" value="0" />
75              </config>
76          </config>
77          <config key="connection_1">
78              <entry key="sourceID" type="xint" value="3" />
79              <entry key="destID" type="xint" value="1" />
80              <entry key="sourcePort" type="xint" value="1" />
81              <entry key="destPort" type="xint" value="1" />
82              <entry key="ui_classname" type="xstring" value="
                org.knime.core.node.workflow.ConnectionUIInformation" />
83              <config key="ui_settings">
84                  <entry key="extrainfo.conn.bendpoints_size" type="xint" value="0" />
85              </config>
86          </config>
87      </config>
```

Figure 4: Connections in XML

Connection 0 represents the connection between node 1 and node 2 (from Figure 1), which is the connection from the Column Filter node to the CSV writer. Connection 1 represents the connection between node 3 to the node 1, which is the connection from the File Reader node to the Column Filter node. All nodes in KNIME have source port numbers, because some nodes can have more than one incoming or outgoing port. A node's ports are where connections are coming into or going out of that node to or from another node.

These XML files represent the raw data of both the node configurations and the connections between the nodes. The KDL compiler will take these files and represent them as a single editable text file in a much cleaner syntax relative to the existing XML representation.

KDL Syntax

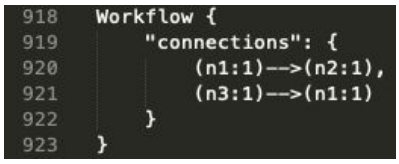
A KDL text file represents all of the data from a KNIME archive file. This includes the workflow.knime file and every node XML file (settings.xml). The file first lists all of the nodes, and then all of the connections at the bottom. Each node includes metadata and the configuration data. The latter is represented in JSON, and is same data found in the config tag with the "model" attribute. Each connection

between two nodes is represented on a single line. Figure 5 shows the beginning of the Column Filter node in KDL.

```
Nodes {
    (n1): {
        "name": "Column Filter",
        "factory": "org.knime.base.node.preproc.filter.column.
        DataColumnSpecFilterNodeFactory",
        "bundle_name": "KNIME Base Nodes",
        "bundle_symbolic_name": "org.knime.base",
        "bundle_version": "3.7.1.v201901291053",
        "feature_name": "KNIME Core",
        "feature_symbolic_name": "org.knime.features.base.feature.group",
        "feature_version": "3.7.1.v201901291053",
        "model": [
            {
                "column-filter": [
                    {
                        "filter-type": "STANDARD"
                    },
                    {
                        "included_names": [
                            {
                                "array-size": 12
                            },
                            {
                                "0": "age"
                            },
                            {
                                "1": "fnlwgt"
                            },
                            {
                                "2": "education"
                            },
                            {
                                "3": "education-num"
                            },
                            {
                                "4": "marital-status"
                            },
                            {
                                "5": "relationship"
                            },
                            {
                                "6": "race"
                            },
```

Figure 5: Column Filter in KDL

The node starts with metadata, such as the version of KNIME and type of node. The id number (n1) is the unique id number given to the node when it is created in the application. The “model” section begins all of the configuration settings for the node, and is represented as JSON in the KDL file. The connections in KDL appear at the bottom of the file as shown with Figure 6.



```
Workflow {
    "connections": {
        (n1:1)-->(n2:1),
        (n3:1)-->(n1:1)
    }
}
```

Figure 6: Connections in KDL

The syntax for connections is elegant and simple. It only takes one line to show a connection instead of four lines in the XML file. Standard connections between nodes use “-->” syntax. The syntax (n1:1) means ([node id]:[node port]). An entire connection is represented as

([sourceID]:[sourcePort])-->([destinationID]):([destinationPort]). The KDL compiler converts KNIME archive files to KDL text files and vice-versa.

KDL Compiler

The compiler is a Python application with a command line interface (CLI). Python was chosen for its extensive usage in the data science community. The application is designed to take in a KNIME archive file and output a KDL text file (extension ".kdl"), and vise-versa. To convert an archive file to a KDL file the command line argument would look like:

```
kdlc --input knime_archive_file.knwf --output textfile.kdl
```

or

```
kdlc -i knime_archive_file.knwf -o textfile.kdl
```

To convert a KDL file to a knime archive file (which could be imported back into KNIME):

```
kdlc --input textfile.kdl --output kine_archive_file.knwf
```

or

```
kdlc -i textfile.kdl -o knime_archive_file.knwf
```

In each case the input file must be in the same working directory as the KDL compiler. The output file will be created with the given name. The Python library Click (CLI Creation Kit) is used to handle the arguments given by the user. The compiler parses the input, stores the data in memory, and then writes the appropriate output file.

Conversion Process

The primary function of the KDL compiler is to convert from KNIME archive files to KDL text files and back. Each conversion process involves parsing the input, storing the data in Python data structures in memory, and then writing to the appropriate output format. When a KNIME archive file is given as input the KDL compiler unzips it. Subsequently, kdlc walks through the unzipped contents and every settings.xml file within a node folder is parsed using ElementTree, and a node object is created in Python. The workflow.knime file is parsed and for every connection, a connection object is created in Python. The application then loops through the collections of each of these objects to write the corresponding node and connection data within the KDL text file. In the reverse direction the KDL syntax is parsed using ANother Tool for Language Recognition (ANTLR). ANTLR serves as a parser generator, which kdlc employes for defining the KDL grammar and utilizes for walking KDL to build an in-memory model within Python of the KNIME workflow at hand from the sourced KDL file. These objects are once again looped through, this time using a tool called Jinja2 to create XML files from a template. A settings.xml file is created for each node object and a single workflow.knime XML file is created from every connection object. These

XML files are then compressed into a single KNIME archive file, which could be imported into the application.

Additional Features

KDL supports meta nodes. In KNIME a meta node is way to condense part of a workflow into a single a node. This can be done for organizational purposes, aesthetic purposes, or to reuse a generic sub-workflow in other larger workflows. Within the application a meta node is considered its own workflow. Within the archive file a meta node folder has its own collection of nodes and its own workflow.knime (which designates the node connections within that meta node). Figure 7 highlights how to represent a meta node with KDL.

```
196     (n14): {
197         "name": "Metanode",
198         "type": "MetaNode",
199         "connections": {
200             (n4:1)-->(META_OUT:3),
201             (n4)~~>(META_OUT:2),
202             (n13:1)~~>(n4),
203             (n13:3)-->(n4:1),
204             (n13:2)-->(META_OUT:1)
205         },
206         "meta_in_ports": [],
207         "meta_out_ports": [
208             {
209                 "1": "org.knime.core.node.BufferedDataTable"
210             },
211             {
212                 "2": "org.knime.core.node.port.flowvariable.FlowVariablePortObject"
213             },
214             {
215                 "3": "org.knime.core.node.BufferedDataTable"
216             }
217         ]
218     },
219     (n14.4): {
220         "name": "Row Filter",
221         "factory": "org.knime.base.node.preproc.filter.row.RowFilterNodeFactory",
222         "bundle_name": "KNIME Base Nodes",
223         "bundle_symbolic_name": "org.knime.base",
224         "bundle_version": "3.7.1.v201901291053",
225         "feature_name": "KNIME Core",
226         "feature_symbolic_name": "org.knime.features.base.feature.group",
227         "feature_version": "3.7.1.v201901291053",
228         "model": [
229             {
230                 "rowFilter": [
231                     {
232                         "RowFilter_TypeID": "RangeVal_RowFilter"
233                     },
234                     {
235                         "ColumnName": "age"
236                     },
```

Figure 7: KDL describing a meta node

There are two things to note. First, the id of a node within a meta node uses dot notation. This not only identifies the node but also identifies the meta node in which it is nested. Second, the connections within a meta node are designated within the meta node itself and not at the end of the file. Within Figure 7, node 14 is a meta node and a list of connections within that meta node are present. Node 14.4 is a node with an

id of 4 that exists within a meta node whose id is 14. Due to a meta node being considered a separate workflow, it is possible for there to be another node with id 4 somewhere else (either within the larger workflow or another meta node).

Another feature of KDL is the support of flow variables. In KNIME a standard workflow passes data in a table-like object from one node to the next, which is represented by a black directed arrow. In addition, a single variable can be defined at one node and passed to another node further down the workflow. This is represented as a red line connecting the two nodes. In KDL that connection is represented with tildes as "~~>", and appears in the connections section at the bottom of the file. A flow variable connection between two non-meta nodes does not include port numbers in the KDL, because the XML files always designate those ports as 0. However, such connections between at least one meta node will include port numbers, because a port going into or coming out of a meta node is not guaranteed to be 0.

Figure 8 provides a side-by-side comparison of a workflow variable in KNIME and in KDL. In this example there is a flow variable connection between node 3 and node 1. Port numbers are not present, because both nodes are non-meta nodes.

Figure 8: Comparison of workflow to KDL representation using flow variables

The KDL compiler also supports templating for nodes. A template for a given node is a single JSON file. Any information about the node, such as metadata or configuration data, is included in the file. When a conversion from KDL to a workflow is called on the command line, an optional argument ("-tp" or "--templates_path") can be appended with a path to a directory. If a template file for a particular node type in the given path matches a node in the KDL file, the compiler will dynamically merge the input from both files. Any information about a node node included in the template is expected to be included in the KDL file.

There are several benefits to this templating functionality. Currently a node in KDL must include several lines of metadata before the configuration content, such as the version of KNIME and the Java factory which created it. Template files for frequently used nodes can include this metadata, which means the metadata would only need to be written once in the template file. From that point forward, nodes in a KDL file could exclude that content and provide a path to the template directory. For example, KNIME requires the attributes and values of lines 4-10 of Figure 5, but introduce bloat within a KDL node

definition. A template file withholds these boilerplate node configurations, so KDL authors define concise node definitions with unique attribute declarations with the standard attributes absent.

```
jared@Macbook-Air:~/repos/k-descriptor-language/kdl (master)
$ kdlc -i input.kdl -o output.knwf --templates_path examples/custom_templates
```

Figure 9: Example use of custom node template feature

Additionally, a debug mode is included in the KDL compiler. This is to provide insight for future developers within the KNIME community into the internals of the kdlc. The debug flag enables verbose logging to console of what the compiler is processing in real time to assist in debugging and understanding the internals of the KDL compiler. This is accomplished by integrating Loguru into the application, a Python library which makes generating and customizing logging messages extremely simple. Presented within Figure 10 is an example of some of the output from the debug mode on a simple compilation of a KDL file into a KNIME archive.

Debug mode can be run in either direction (KDL to KNIME archive or KNIME archive to KDL). To run debug mode append "-d" or "--debug" to the end a the CLI argument:

```
kdlc -i knime_archive_file.knwf -o textfile.kdl --d
```

or

```
kdlc --input knime_archive_file.knwf --output textfile.kdl --debug
```

```
2019-05-05 18:09:51.210 | kdlc.commands:kdl_to_workflow:23 - ======= BEGIN KDL to WORKFLOW =======
2019-05-05 18:09:51.359 | kdlc.commands:kdl_to_workflow:41 - ======= Nodes from walker =======
2019-05-05 18:09:51.359 | kdlc.commands:kdl_to_workflow:42 - ['NodeID: 1 NodeName: CSV Reader', 'NodeID: 2 NodeName: Table to JSON', 'NodeID: 3 NodeName: Column Filter']
2019-05-05 18:09:51.360 | kdlc.commands:kdl_to_workflow:43 - ==== Connections from walker ====
2019-05-05 18:09:51.360 | kdlc.commands:kdl_to_workflow:44 - ['ConnectionID: 0 Source:1.1 Dest:3:1', 'ConnectionID: 1 Source:3.1 Dest:2:1']
2019-05-05 18:09:51.360 | kdlc.commands:kdl_to_workflow:49 - ======= Unflattened Nodes =======
2019-05-05 18:09:51.360 | kdlc.commands:kdl_to_workflow:50 - ['NodeID: 1 NodeName: CSV Reader', 'NodeID: 2 NodeName: Table to JSON', 'NodeID: 3 NodeName: Column Filter']
2019-05-05 18:09:51.360 | kdlc.commands:kdl_to_workflow:51 - ==== Normalized Connections ====
2019-05-05 18:09:51.360 | kdlc.commands:kdl_to_workflow:52 - ['ConnectionID: 0 Source:1.1 Dest:3:1', 'ConnectionID: 1 Source:3.1 Dest:2:1']
2019-05-05 18:09:51.360 | kdlc.commands:build_knwf:124 - ======= BEGIN BUILD KNWF =======
```

Figure 10: Example of debugging output produced by kdlc

Results

The KDL compiler is designed to take a KNIME workflow (via an archive file) and convert it to a text file in KDL syntax. Based on extensive testing, the compiler should be able to handle and parse most nodes available in the application. Testing was primarily performed using several of the example workflows that are included in KNIME. These workflows were exported, converted to KDL, and then converted back to archive files and imported into the application. Success criteria included verifying the workflow rendered and ran correctly. It should be noted that a workflow created from a KDL file will not include coordinates for node locations in the GUI, and upon import the nodes will appear stacked on top of each other. The "auto layout" button fixes this by spreading the nodes out across the screen.

When a data file is included within an archive file, by default the URL to that file (which would be a configuration within a File Reader node) is a relative path. Because the data file is not preserved if the archive file were to be converted to KDL and back again, the workflow would be unable to run in KNIME. Relative paths must be changed to absolute paths to avoid this issue. If the KDL compiler finds a File Reader node while converting from KDL to a KNIME archive file, it will produce a warning such as the one shown within Figure 11.

```
$ kdlc -i 02_Column_Filter.kdl -o 02_Column_Filter.knwf
====== WARNING =======
Node File Reader contains relative paths: NodeID:3
The following settings have relative workflow paths and must be manually adjusted in KNIME:
{'DataURL': 'knime://knime.workflow/data/sales_2008-2011.csv'}
```

Figure 11: Warning message regarding relative path to data file

Most nodes appear to follow a standard pattern based on the XML files from the export. This includes the config tag with the model attribute that describes all of the configuration settings for that node. Nodes that have a significantly different structure, such as custom nodes which were created by a third-party developer, may not be converted correctly by the compiler.

The time it takes for the compiler to run in both directions is fairly quick. Large workflows with several hundred nodes have been found to convert to KDL in under a minute, and large KDL files with several hundred nodes have been found to convert to workflow archives files in only a few minutes.

Discussion

KDL is an alternative to the KNIME GUI for building and editing workflows. Unlike many of the KNIME extensions that already exist (such as creating new nodes), KDL offers a user an entirely new way to view and edit workflows. The KDL compiler does not require an external database to store the data, which reduces the dependency on another platform. The compiler can be run locally as a Python application. Although the KNIME GUI is still great for KNIME novices and non-programmers, KDL is a great tool for a programmer who would like to take advantage of a text-based language with automation opportunities.

Text Edit Tools

Having a workflow in a text format has many advantages. Any tool that can be used to edit text can now be used to edit a KNIME workflow. Simple text searches can easily find specific nodes, or nodes with specific values. The entire UNIX toolkit becomes available to perform tasks such as search and replace. Bulk edits can be made, such as changing the configuration of multiple nodes at the same time. Nodes can be duplicated by copying and pasting. This kind of editing is made even easier by the fact that the entire workflow is viewable at once, rather than in multiple files.

Single File

One key advantage of KDL is viewing and editing the entire workflow in a single text file. This may seem like a minor point, but the ability to view a complex workflow (especially one with several meta nodes) at one time on one screen can greatly help in grasping what that workflow does and how it is laid out. Having all data regarding meta nodes in a single file can help developers see and understand the nesting that is occuring at a high level. Instances of multi-level nesting of meta nodes in the KNIME GUI can be hard to follow. The ability to make all edits in a single file also makes versioning easier.

Version Control

Another advantage of KDL is version control. As with any text-based language, version control can be used to keep track of changes and revert back if necessary. KNIME workflows no longer need to be thought of just existing and being updated in the application itself. Collaborative version control tools such as GitHub could allow multiple users to work together on a workflow before it is run on a machine. Thinking of a workflow as a text based language opens up several possibilities for version control and collaboration. Figure 12 depicts an example of where the introduction of a flow variable to an existing workflow can be easily captured in the Git commit history of the repository containing the KDL file of the workflow. This type of collaborative editing is currently not possible because KNIME is a desktop application that must be run on a local machine.

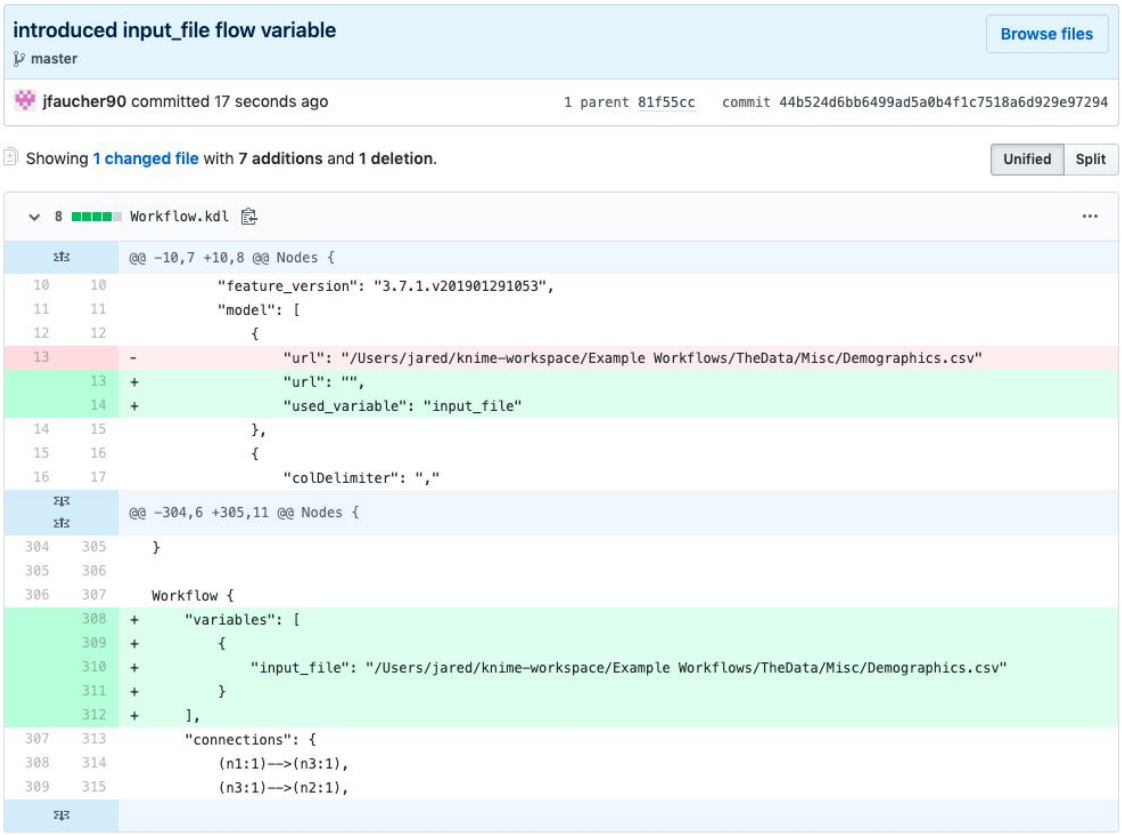


Figure 12: Example git diff of KDL changes

Conclusion

KDL is a domain-specific language for representing KNIME workflows in a text-based format. Although VPLs such as the KNIME GUI offer advantages to those new to the application, we believe KDL offers advantages to more seasoned KNIME users with programming experience and a need for alternative editing functionality. UNIX commands, version control, and other text based tools open up new opportunities for the automation, collaboration, and editing of KNIME workflows. The fact that the entire workflow appears in a single file can making reading and editing easier and more straightforward. We hope the KNIME community finds KDL to be a useful and exciting tool.

References

KNIME (https://www.knime.com)

Berthold M.R., et al.Preisach C., et al.KNIME: The Konstanz Information Miner, Data Analysis, Machine Learning and Applications: Studies in Classification, Data Analysis, and Knowledge Organization, 2008, vol. V(pg. 319-326)

Warr, W.A. J Comput Aided Mol Des (2012) 26: 801. https://doi.org/10.1007/s10822-012-9577-7

Bernd Jagla, Bernd Wiswedel, Jean-Yves Coppée, Extending KNIME for next-generation sequencing data analysis, *Bioinformatics*, Volume 27, Issue 20, 15 October 2011, Pages 2907–2909, https://doi.org/10.1093/bioinformatics/btr478

Pierre Lindenbaum, Solena Le Scouarnec, Vincent Portero, Richard Redon, Knime4Bio: a set of custom nodes for the interpretation of next-generation sequencing data with KNIME, *Bioinformatics*, Volume 27, Issue 22, 15 November 2011, Pages 3200–3201, https://doi.org/10.1093/bioinformatics/btr554

Kosar, Tomaz, et al. “Comparing General-Purpose and Domain-Specific Languages: An Empirical Study.” *Digital Library of IPB*, 2000, bibliotecadigital.ipb.pt/bitstream/10198/2286/1/ComSisDSLCameraReady.pdf.

Barzdins, Janis, et al. “Domain Specific Languages for Business Process Management: a Case Study.” *DMS Forum*, 2019, www.dsmforum.org/events/dsm09/papers/barzdins.pdf.

Prähofer, Herbert, et al. “The Domain-Specific Language Monaco and Its Visual Interactive Programming Environment.” *IEEE Xplore Digital Library*, 2007, ieeexplore-ieee-org.ezp-prod1.hul.harvard.edu/document/4351334.

Holzschuher, R. Peinl. Performance of graph query languages: comparison of cypher, gremlin and native access in Neo4j. EDBT '13 Proceedings of the Joint EDBT/ICDT 2013 Workshops Pages 195-204

Holzschuher, R. Peinl, Querying a graph database - language selection and performance considerations. Journal of Computer and System Sciences Volume 82 Issue 1, February 2016

openCypher reference https://www.opencypher.org/references

openCypher project https://www.opencypher.org/projects

Whitley, K. N, Blackwell, Alan F. Visual Programming: The Outlook from Academia and Industry, 1997. *https://www.cl.cam.ac.uk/~afb21/publications/ESP97-kirsten.html*

Repenning, Alexander. "Moving Beyond Syntax: Lessons from 20 Years of Blocks Programing in AgentSheets", Journal of Visual Languages and Sentient Systems, July 2017 https://sgd.cs.colorado.edu/wiki/images/2/21/20YearsofBlockProgramingLessonsLearned_published.pdf

Software Design

Overview

This section covers the design of the KDL compiler. This includes the structure of the Python application, the supporting third party libraries, and an in-depth look at implementation details

Technology Stack

- Language: Python 3
- Supporting Libraries/Tools
    - XML parsing + generation:
        - ElementTree
        - Jinja2
    - KDL parsing:
        - ANTLR
    - CLI Parsing:
        - Click
    - Testing
        - pytest
    - Code Coverage
        - Coverage.py
    - Code Quality
        - Codacy
    - Static Typing
        - mypy
    - Logging
        - loguru

KNIME workflow to KDL process flow

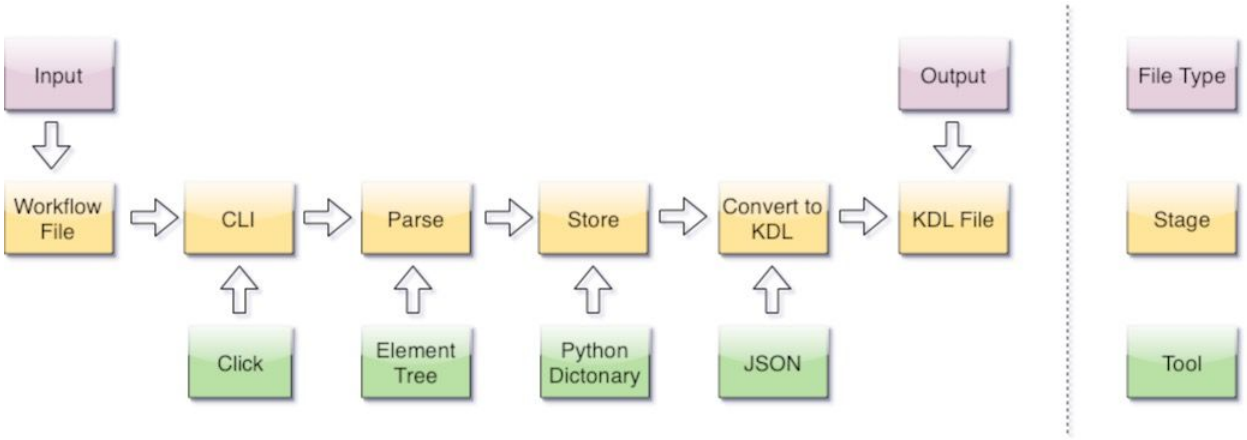


Figure 13: KNIME workflow to KDL process flow

This process takes as input a KNIME archive file (*.knwf extension), which can be generated by exporting a workflow from the KNIME GUI. Using the Click library, the CLI checks that the options and arguments are set correctly (see Command Line Arguments below). From there it parses the underlying XML files within the KNIME workflow and stores the resulting data in a Python dictionary. From there it writes the data to a KDL text file using the KDL syntax, which is primarily JSON. Lastly, the kdlc outputs a KDL text file, the name of which was given as an argument to the CLI.

KDL to KNIME workflow process flow

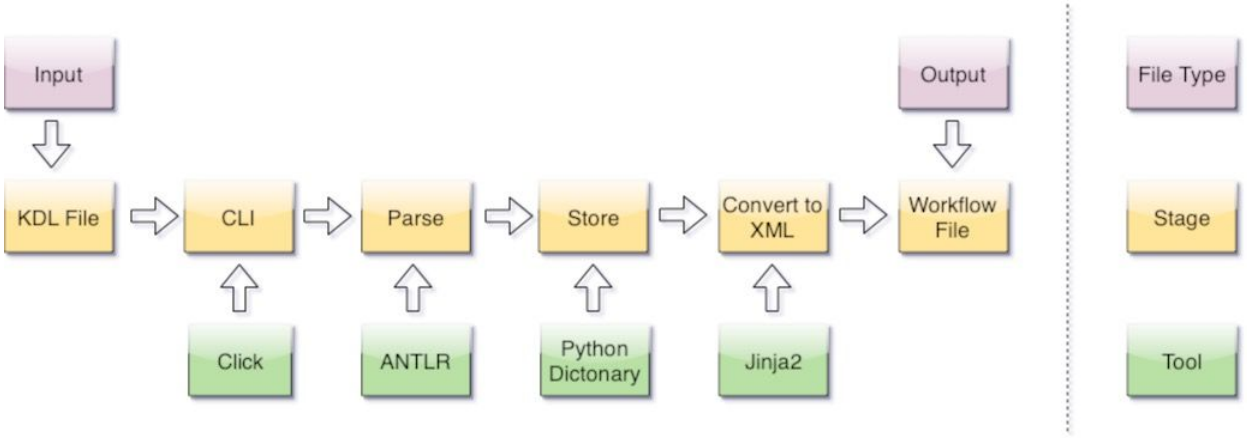


Figure 14: KDL to KNIME workflow process flow

This process takes as input a KDL text file. Using the Click library, the CLI checks that the options and arguments are set correctly (see Command Line Arguments below). ANTLR is then used to parse the KDL syntax, and the resulting data is stored in Python dictionaries. The content from those dictionaries are then fed into XML template files using the Jinja2 library. Those XML files are then compressed into a

single KNIME archive file (*.knwf extension) which is the output. The name of the new file is given as one of the arguments in the CLI.

Command Line Arguments

These are examples of the command line arguments being supplied to the program
To convert a KNIME archive file (.knwf) to a KDL text file:

```
kdlc -i workflow.knwf -o workflow.kdl
kdlc --input workflow.knwf --output workflow.kdl
```

To convert a KDL text file to a KNIME archive file (.knwf):

```
kdlc -i workflow.kdl -o workflow.knwf
kdlc --input workflow.kdl --output workflow.knwf
```

For help:

```
kdlc --help
```

To run in debugger mode append the following onto either of the conversation argument lists

```
-d
--debug
```

To use node templates:

```
kdlc -i workflow.kdl -o workflow.knwf -tp path
kdlc --input workflow.kdl --output workflow.knwf --templates_path path
```

Use and Success of Languages and Platform

Python

Python has worked out well. It is a relatively easy language to pick up, and its widespread community results in plenty of documentation and tutorials. The biggest advantage to using Python for this project has been the wide range of libraries. The supporting libraries mentioned in the technology stack have been able to do exactly what was needed and have been easy to learn and use. These libraries have meant not needing to reinvent the wheel, and have changed the question from "how do we do this complicated task" to “how do we integrate this tool into our application". Python was favored by our customer because it is the language most data scientists are likely to be familiar with, including the ones who will be maintaining the application with our customer after release.

Graph Databases

We decided not to use a graph database. The original plan was to use Cypher as the syntax for the KDL. However, the language did not meet the requirements. KDL syntax must be able to handle the nesting properties of node settings and Cypher does not provide a sufficient option for the nesting of node properties. Once ANTLR was discovered as a tool for both creating custom syntax and parsing it, Cypher and an accompanying graph database (neo4j) was no longer needed. The primary interest in using Cypher and neo4j was to use an already defined syntax to describe nodes and edges and a mechanism to parse and store them. ANTLR does all of this and integrates easily with Python.

Main Python Files and Flow Control

application.py
Entry point into the program. The file's only purpose is to call kdlc.prompt(), which will parse the arguments from the CLI.

cli.py
Includes the prompt() function which parses the arguments given to the CLI. All correctly typed arguments will be handed to the appropriate function in the commands.py file. All incorrectly typed arguments returns an appropriate error message to the user and exits. All CLI parsing is done using Click.

commands.py
Currently includes both high level functions mirroring the two primary processes:

- Convert a KDL file to a KNIME archive file (.knwf file)
- Convert a KNIME Workflow (.knwf file) to a KDL file.

Each of these high level functions include several subroutines which are within the core.py file.

core.py
A collection of subroutine functions, which handles a majority of the work within the application.

objects.py
Nodes and connections are the key elements of a KNIME workflow, and thus are the main objects used in the application. There are abstract classes for nodes, connections and workflows which all contain various methods, including “kdl_str” which returns the KDL string representation of the object. Using abstract classes and inheritance allowed us to separate out business logic for the various domain objects within KNIME and easily maintain and extend our application. These abstract classes are implemented as the following:

- A Node object maintains its name (type of node) and ID number, as well as several top level values such as which KNIME node factory it belongs to. It also maintains a list of attributes, which relate to the values in the configuration flow in the KNIME GUI. These are the same

values listed in the <config key="model"> in the settings.xml file. This content is rendered as JSON within a KDL document.

- A MetaNode object maintains its name and ID as well as the child nodes and connections contained within the meta node. MetaNodes are distinguished from WrappedMetaNodes by the “type” attribute with the value “MetaNode”.
- A WrappedMetaNode object inherits its functionality from the MetaNode parent class, containing a name and ID as well as the child nodes and connections. WrappedMetaNodes are distinguished from MetaNodes by the “type” attribute with the value “SubNode”.
- A Connection object represents a connection between two nodes. It defines the nodes by their ID, and includes the port number for each node, as well as holding a reference to the source and destination Node objects.
- A VariableConnection object represents a flow variable connection between two nodes. Similarly to the Connection, it defines the nodes by their ID, and includes the port number for each node. However, VariableConnections have a separate implementation of the kdl_str method which allows the connections to be represented as a tilde arrow (~~>) in the KDL.
- A Workflow object represents the list of top-level connections between nodes as well as the global flow variables which are stored as Python dictionaries.

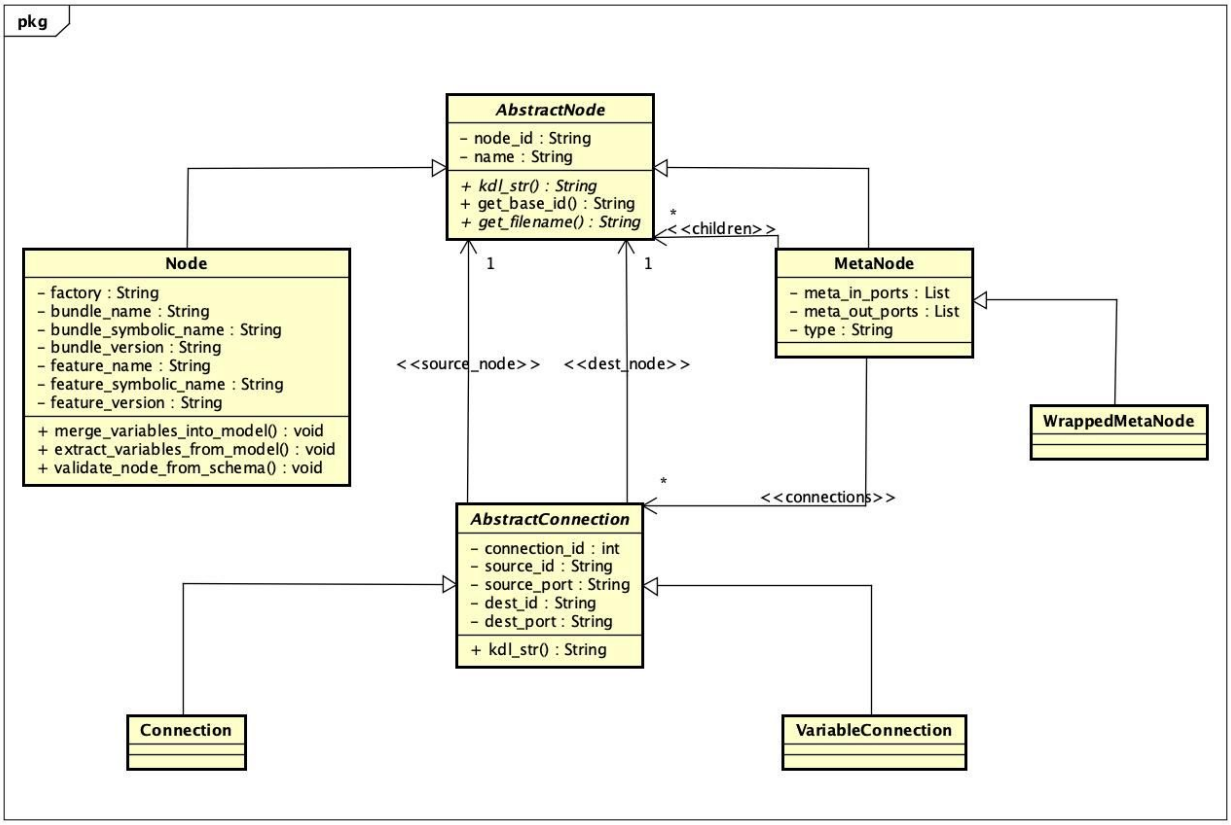


Figure 15: Node and Connection Classes

KDLLoader.py
An implementation of the KDLListener class generated by our ANTLR grammar file. This class is used to load the nodes and connections into memory while walking the Abstract Syntax Tree (AST) that is produced by the ANTLR generated parser.

template_catalogue.py
A TemplateCatalogue object which is used for retrieving node templates and merging them with settings extracted from the KDL during the compilation process.

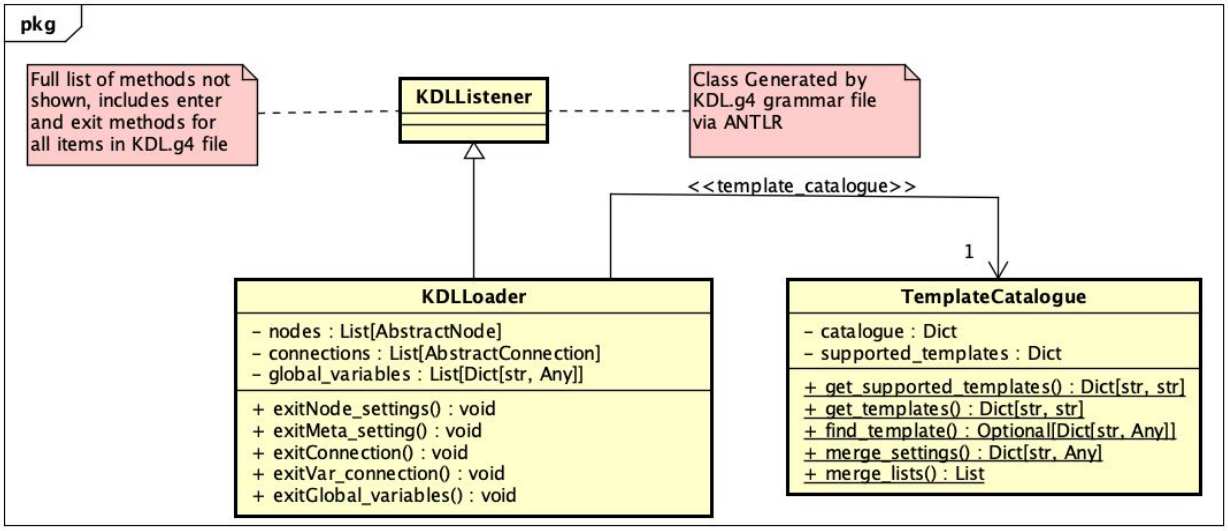


Figure 16: The KDLLoader and TemplateCatalogue Classes

Support Files and Tools

ANTLR and grammar files

ANother Tool for Language Recognition (ANTLR) is used to parse the KDL syntax. The grammar folder contains two .g4 files, which are used to define a specific kind of the grammar. The file titled KDL.g4 defines the KDL syntax, which describes how to define a node and a workflow. It also describes how to define node connections and node settings, which relies on the JSON.g4 for JSON syntax. ANTLR has a steeper learning curve than some of the other tools utilized within the kdlc implementation. Building a language, even a simple one, is not trivial. Consider English as an example, it starts with defining a single character (either an uppercase or lowercase letter). From there a word can be defined, which is a collection of letters typically all lower case except for proper names or at the beginning of a sentence. From there a sentence is a collection of words delimited by spaces and ending with a period or a question mark. Defining such a syntax starts with the most basic character and builds out.

This is the approach taken with defining a syntax for KDL. First a node ID is defined, which is just a number. Single units such as node id numbers are referred to as lexer rules. A node is then defined as including a node ID. A node is an example of a parse rule, as it is made up of lexer rules. Lexer rules can

include parser rules, but not the other way around. Once a single node is defined a set of nodes can be defined. Also, once a node is defined a connection can be defined, as it is made up of two nodes. Using ANTLR to create the KDL syntax took a lot of work to determine the building blocks of the language.

Figure 17 presents the KDL grammar file. The lexer rules are written first in all caps, followed by the parser rules in lowercase. The file starts with simple atomic lexer rules and then continues to simpler and then to more complex parser rules. The last three parser rules are for the node collections, variable collections, and node connection collections. The parser rules tell ANTLR what should be expected in what order. For example the first two lines in the meta node settings are “STRING COLON STRING”, which will be the name and type of node (see Figure 24 regarding meta nodes). This will be followed by at least one connection type (the bar means “or”), followed by the meta_in_port and meta_out_port before the closing curly brace. If KDL syntax broke any of these rules the program would be unable to continue parsing.

```
/*
 * KDL
 */

grammar KDL;
import JSON;
WS          : [ \t\n\r]+ -> skip ;
ARROW       : '-->' ;
VARIABLE_ARROW: '~~>' ;
NODEPREFIX  : 'n' ;
COLON       : ':' ;
COMMA       : ',' ;
DOT         : '.' ;
CONNECTION_TAG : '"connections"';
META_IN_TAG : '"meta_in_ports"' ;
META_OUT_TAG : '"meta_out_ports"' ;
META_IN : 'META_IN' ;
META_OUT: 'META_OUT' ;

node_id     : NUMBER (DOT NUMBER)* ;
port_id     : NUMBER ;
port        : COLON port_id ;
node        : '(' NODEPREFIX node_id port? ')' ;
node_settings: node COLON json ;
source_node   : node ;
destination_node  : node ;
connection  : source_node ARROW destination_node ;
var_connection : source_node VARIABLE_ARROW destination_node ;
meta_in_node : '(' META_IN port ')' ;
meta_out_node : '(' META_OUT port ')' ;
meta_connection: meta_in_node ARROW destination_node
    | source_node ARROW meta_out_node
    | meta_in_node ARROW meta_out_node ;
meta_var_connection: meta_in_node VARIABLE_ARROW destination_node
    | source_node VARIABLE_ARROW meta_out_node
    | meta_in_node VARIABLE_ARROW meta_out_node ;
meta_in_ports: META_IN_TAG COLON json ;
meta_out_ports: META_OUT_TAG COLON json ;
meta_settings: node COLON '{' STRING COLON STRING ','
                              STRING COLON STRING ','
                              CONNECTION_TAG COLON '{'
                              (connection | var_connection | meta_connection | meta_var_connection)
                              (COMMA (connection | var_connection | meta_connection| meta_var_connection))*
                              '}' COMMA
                              meta_in_ports COMMA
                              meta_out_ports
                            '}';
node_list: (node_settings | meta_settings)
                    (COMMA (node_settings | meta_settings))* ;
nodes       : 'Nodes {' node_list? '}' ;
global_variables: '"variables": ' json COMMA? ;
workflow_connections: CONNECTION_TAG COLON '{'
                            (connection|var_connection)
                             (COMMA (connection|var_connection))*
                         '}' ;
workflow: 'Workflow {' global_variables?
                       workflow_connections?
                    '}' ;
```

Figure 17: KDL Grammar Rules (KDL.g4)

Jinja2 and XML Templates

Jinja2 is a template creator for Python. The kdlc/templates folder includes three templates:

- settings_template.xml: This template creates the settings.xml files which represent every node. This template is built by recursively looping through the settings attribute in the node object and building <config> and <entry> tags. It is recursive because both <config> and <entry> tags could be nested in other <config tags>.
- workflow_template.xml: This templates creates the workflow.knime XML file which represents all of the connections between the nodes. The template is built by recursively looping through the connection objects and building <entry> tags representing node IDs and Ports.
- workflow_settings_template.xml: Wrapped meta nodes require an additional settings.xml in the node's directory, which differs significantly from the original settings template. The team introduced an additional template for wrapped meta nodes to avoid introducing additional complexity to the original settings_template.xml file.

Jinja2 allows easy template creation by passing Python lists and dictionaries into an XML document. It then provides a syntax to iterate through those Python collections and insert the desired content. Jinja2 uses open and close curly brackets to inject itself within an XML document, similar to how PHP injects itself into an HTML document using <?php and ?> angle brackets. Once inside the curly brackets content can be accessed from the passed in Python collection. Control logic, such as loops, and if statements can be used to dynamically determine which data should be placed within a given tag or attribute. Almost all of the XML file creation is done with the given XML template file itself, which makes the process concise and readable.

Figure 18 illustrates a brief example of a Jinja2 Template file.

```
<?xml version="1.0" encoding="UTF-8"?>
<config xmlns="http://www.knime.org/2008/09/XMLConfig" xmlns:xsi="http://www.w3.
org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.knime.org/2008/09/
XMLConfig http://www.knime.org/XMLConfig_2008_09.xsd" key="settings.xml">
<entry key="node_file" type="xstring" value="settings.xml"/>
<config key="flow_stack"/>
<config key="internal_node_subsettings">
<entry key="memory_policy" type="xstring" value="CacheSmallInMemory"/>
</config>
<config key="model">
    {% for item in node.model recursive %}
    {% if item.data_type=="config" %}
        {% set config_items = {} %}
        {% for key, value in item.items() %}
            {% if key=="data_type" %}
            {% else %}
                {% do config_items.update({"key": key}) %}
                {% do config_items.update({"value": value}) %}
            {% endif %}
        {% endfor %}
        <config key="{{config_items['key']}}">
        {{ loop (config_items['value']) }}
        </config>
    {% else %}
        {% set entry_items = {} %}
        {% for key, value in item.items() %}
            {% if key=="data_type" %}
                {% do entry_items.update({"type": value}) %}
            {% elif key=="isnull" %}
                {% do entry_items.update({"isnull": "true"}) %}
            {% elif key=="exposed_variable" or key=="used_variable" %}
            {% else %}
                {% do entry_items.update({"key": key}) %}
                {% do entry_items.update({"value": value}) %}
            {% endif %}
        {% endfor %}
        {% if entry_items.isnull is defined %}
            <entry isnull="true" key="{{entry_items['key']}}" type="
            {{entry_items['type']}}" value="{{entry_items['value']}}" />
        {% else %}
            <entry key="{{entry_items['key']}}" type="{{entry_items['type']}}" value="
            {{entry_items['value']}}" />
        {% endif %}
    {% endif %}
    {% endfor %}
</config>
{% if node.variables %}
```

Figure 18: Jinja2 Template for settings.xml file

As mentioned earlier the information regarding the configuration of the node is located within the config tag with the value “model”. Jinja2 is given the Python dictionary which holds all of the model information within the node object, and recursively loops through that dictionary printing either a config tag or an entry tag with the appropriate content. Jinja2 uses conditional logic to determine the next item in the loop and handles it appropriately.

Click and CLI Parsing

Click is an easy to use tool for defining acceptable CLI arguments. The argument syntax (including options and file extensions) were already defined in the requirements section, so it was just a matter of implementation. Click essentially lets the programmer state which options are expected and discards all the rest. Certain options can be made to be required. Counting the number of arguments is not necessary.

Click also provides easy to use error handling, which provides a message to the user and explains why the given arguments were not accepted.

Figure 19 highlights the click options in the cli.py file. For each command line argument option the user can specify the short and long naming conventions, as well as whether or not the argument is required and text for the help message.

```
@click.command()
@click.option(
    "--output",
    "-o",
    "output_file",
    required=True,
    help="The output file, either .knwf or .kdl",
)
@click.option(
    "--input",
    "-i",
    "input_file",
    required=True,
    help="The input file, either .knwf or .kdl",
    type=click.Path(exists=True),
)
@click.option(
    "--debug",
    "-d",
    "debug_logging",
    required=False,
    help="Print debug logging to stdout",
    is_flag=True,
)
@click.option(
    "--templates_path",
    "-tp",
    "templates_path",
    required=False,
    help="Path to a custom templates catalogue",
    default=None,
    type=click.Path(exists=True),
)
```

Figure 19: Click Annotation

Development of the Tool

Initial Proof of Concept

The application started as a proof of concept, to learn how KNIME archive files could be parsed and replicated in Python. This version of the application took two arguments in the CLI, the name of an existing exported workflow, and the name of the "new" workflow to be created. New is in quotes, because the intent of the application is to take in a KNIME archive file and output an exact match. The application began by decompressing the archive file and getting to the XML files. Within the archive file is an XML file (settings.xml) for each node and one XML file (workflow.knime) describing the connections between the nodes. The Python library ElementTree was used to parse these XML files and store them in Python dictionaries. One dictionary for each node and one for all of the connections. This step helped us discover

how to parse XML files and store them in memory. These dictionaries would later be converted to Node and Connections Objects as the application evolved.

The next step was to take these Python dictionaries and recreate the XML files. This was done with the help of the Jinja2 Python library. The application stored two bare-bones XML templates, one representing a KNIME node and the other a set of connections. Jinja2 was used to feed the dictionaries into the templates and place the content in the appropriate XML tags and attributes. Once this was done the XML files were zipped up following the same folder structure of the KNIME archive file (*.knwf). The application then output the new archive file. This file was imported into the KNIME GUI and successfully run. The only noticeable difference was a workflow with multiple nodes would appear with all the nodes stacked on top of each other. The XML files from an exported workflow does include coordinate values dictating where a node appears on the KNIME workbench grid, we (with the approval of the customer) left these fields blank which resulted in all of the nodes on top of each other. The KNIME GUI has a button to highlight multiple nodes and lay them out in order, which easily solves this issue.
This initial proof of concept proved that XML files from KNIME could be parsed and recreated all within a Python application. It also provided insight on the complexity of the XML files as well as how attributes in those files differ from node to node. The next step was to flesh out the CLI arguments and testing.

CLI Development and Testing

Click was introduced to handle all expected CLI arguments. Even though the application did not have the infrastructure to handle such requests, the scaffolding was put in place to handle such cases at a later date. Pytest and pytest-cov were introduced, and Travis CI was hooked up with our GitHub repository to assess on every build. Moving forward any changes made to the code would be immediately tested.

The Introduction of KDL

At this point the application could not handle either of the primary processes (kdl → workflow, workflow → kdl), because a KDL syntax had not yet been defined. As mentioned earlier Cypher did not work out and ANTLR became the tool that would solve this issue. It was logical to solve the kdl → workflow process first, because it would be much more difficult parsing KDL than recreating it. The goal was to get the content from KDL parsed in such a way that it was identical to the Python dictionaries created in the POC. If that was doable, we already had the code in place to create the KNIME archive file. The most complex part of a KNIME node is the content within the model config tag in XML, which is equivalent to the content in the configure window in the GUI. This content is typically nested. The solution was to represent this in JSON, which follows a very similar structure to a Python dictionary. Custom syntax was created to represent a node and a connection, while the bulk of the node content was represented via JSON. ANTLR stores the content read in in a tree structure, and allows the programmer to walk that tree. The custom syntax divided up the KDL into two main parts: Nodes and Connections. The application then walks the nodes tree and stores the data in a Python dictionary, the same is done for connections. Once those dictionaries were created, the POC already provided a way for the XML files to be created. Thus, the kdl → *.knwf conversion was complete. The *.knwf → kdl process was easier, as the KDL simply needed to be printed out.

Flow Variables

Flow variables in KNIME allow variables created in one node to be passed to another node further in the workflow. This can be thought of as passing a variable from one function to another further down the stack. In the KNIME GUI a flow variable moving from one node to another is represented by a red line with dots at each node, this is to distinguish it from a standard connection which passes all data from one node to another. In KDL we were able to add a flow variable directly into a node's attribute field with a key called "exposed_variable". In addition to this we have also provided a "variables" attribute within the workflow section of the KDL document where global flow variables can be defined. A node later in the workflow would have a corresponding key called "used_variable" which the same flow variable, describing how the variable was moved from one node to another. In the connection section of KDL, a tilde arrow ("~~>") is used to designate a flow variable connection between nodes, which distinguishes it from the standard arrow ("-->") between two nodes.

Figure 20 provides a side-by-side comparison of a flow variable in KNIME and in KDL.

Figure 20: Side by side comparison of flow variables in KNIME GUI (left) and KDL (right)

In this example there is a flow variable connection between node 3 and node 1.

Meta nodes

A meta node is a unique node in KNIME. Any subset of a workflow can be condensed into a single node, which is called a meta node. This can help to organise larger workflows by reducing the number of nodes on the screen.

Figure 21 depicts an example of workflow without meta nodes.

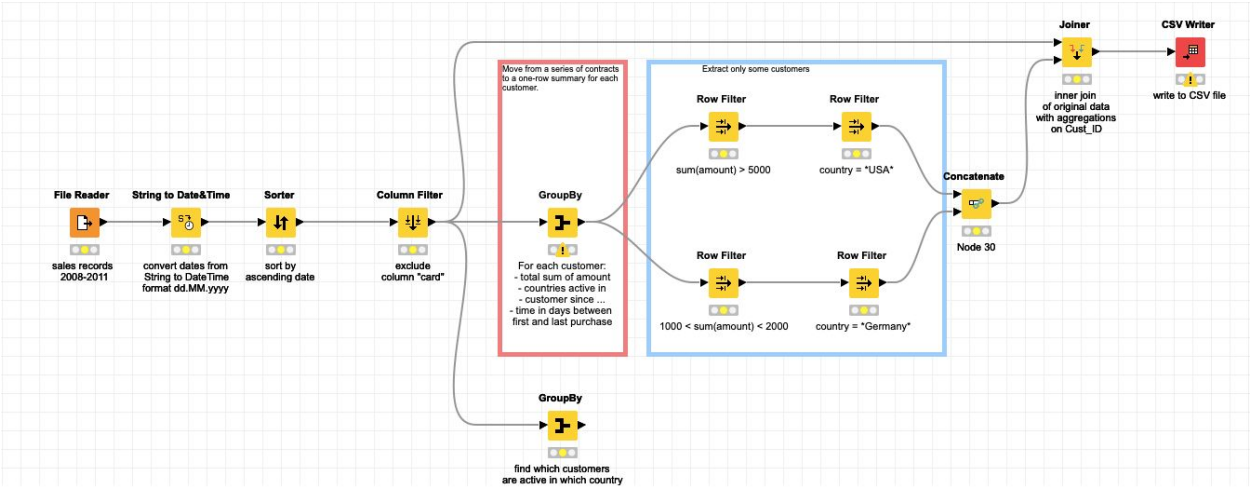


Figure 21: Example of workflow without meta nodes

Figure 22 presents the same workflow after the four Row Filter nodes are condensed into a meta node.

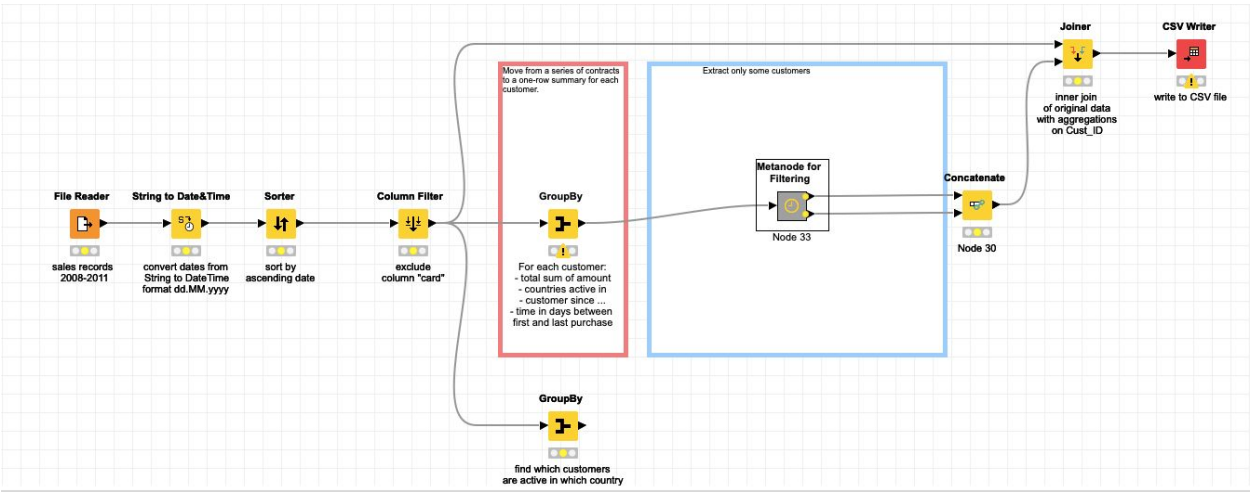


Figure 22: Same workflow as Figure 21 with some nodes condensed into a meta node

Figure 23 shows inside the meta node, which serves as its own workflow.

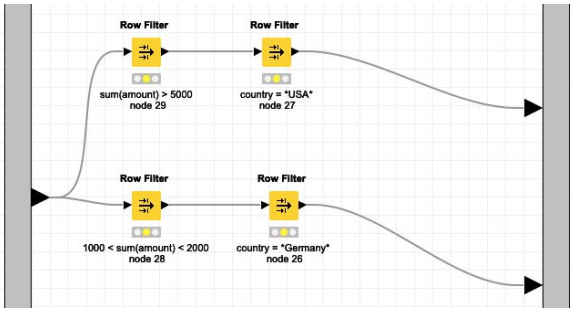


Figure 23: Inside meta node from Figure 22 workflow

The gray bars on either side represent the meta in and meta out ports. They function as barriers that handle incoming and outgoing connections. Figure 24 shows the beginning of the KDL syntax for the meta node shown within Figure 23.

```
(n33): {
    "name": "Metanode for Filtering",
    "type": "MetaNode",
    "connections": {
        (META_IN:1)-->(n29:1),
        (META_IN:1)-->(n28:1),
        (n26:1)-->(META_OUT:2),
        (n27:1)-->(META_OUT:1),
        (n28:1)-->(n26:1),
        (n29:1)-->(n27:1)
    },
    "meta_in_ports": [
        {
            "1": "org.knime.core.node.BufferedDataTable"
        }
    ],
    "meta_out_ports": [
        {
            "1": "org.knime.core.node.BufferedDataTable"
        },
        {
            "2": "org.knime.core.node.BufferedDataTable"
        }
    ]
},
(n33.26): {
    "name": "Row Filter",
    "factory": "org.knime.base.node.preproc.filter.row.RowFilterNodeFactory",
    "bundle_name": "KNIME Base Nodes",
    "bundle_symbolic_name": "org.knime.base",
    "bundle_version": "3.7.1.v201901291053",
    "feature_name": "KNIME Core",
    "feature_symbolic_name": "org.knime.features.base.feature.group",
    "feature_version": "3.7.1.v201901291053",
    "model": [
        {
            "rowFilter": [
                {
                    "RowFilter_TypeID": "StringComp_RowFilter"
                },
                {
                    "ColumnName": "Unique concatenate(country)"
                },
```

Figure 24: KDL equivalent of meta node in Figure 23

In KDL meta nodes keep track of their own internal connections. This is because in the archive files a meta node has its own workflow.knime XML file with its own connections. All nodes within a meta node will immediately follow the meta node in the KDL file. The node ids for the internal nodes are represented in dot notation, with the meta node id on the left and the id of the node itself on the right. Due to a meta node being considered a separate workflow in the application, it may be possible to have two nodes with the same id both inside and outside of the meta node. The dot notation helps keep the id values unique. It is possible for a meta node to be nested within another meta node, so the dot notation would list the highest level meta node through the nested meta nodes to the regular node from left to right. KDL supports both meta nodes and wrapped meta nodes.

Templates

The KDL compiler supports templating, which allows the user to create template files of frequently used nodes. Each individual file corresponds to a specific node type and can include pre-populated values for that node in KDL syntax. A user can then run a KDL to KNIME workflow conversion and include a path to a folder of such template files. If the compiler finds a match between a node in the given KDL document and a template of the same node, the compiler will dynamically merge values from both files before conversion to the .knwf file. Templating is a step closer to authoring workflows from scratch rather than always relying on a pre-populated KDL file from a previous conversion. One use case is using a template file to pre-populate all of a node's metadata, such as the KNIME version and bundle information. Another is creating multiple folders, each with the same set of nodes populated with different values. A user could then create different KNIME workflows from the same KDL file just by providing different folder paths in the command line. Templating has the potential to expand KDL from a simple back and forth conversion tool to a workflow creation tool.

Figure 25: Equivalent node definitions before and after the introduction of template

Testing

In the early stages of development of our project we scaffolded numerous quality checks, continuous integration, and mechanisms for communicating short feedback loops to insure we instill quality within our solution. The quality checks included unit and integration testing, static type checking, linting, style guide validation, and documentation generation. Through providing a simple mechanism to enforce quality with the coupling of these various validation steps the team received regular assurance in the robustness of their application as well as instilled merit in solution for the customer.

In terms of unit and integration testing, we utilized pytest, pytest-cov, and pytest-mock. We strive to not introduce any untested code into our main git branch as well as require test code coverage to remain above 80%. At the moment, we have 117 tests and 93% code coverage. Establishing this practice early within our project has allowed our team to quickly and confidently refactor our code as well as layer on additional functionality.

Additional quality checks included popular tooling within the Python community, such as mypy for static type checking, flake8 for style guide enforcement, black for standardizing our code format, and sphinx for documentation generation. Due to Python being a dynamically typed language, the team decided to introduce mypy coupled with static typing annotations to provide compile-time type checking. The flake8 utility lints our code and insures we abide by PEP-0008, the style guide for Python code, which assists with identifying and eliminating syntactical errors. The black utility serves as an opinionated code formatter, which establishes consistency in our deliverables and improves our code reviews by limiting our differences to business logic rather than style. Lastly, we validate the generation of our documentation, which we build with a utility called sphinx. Due to the nonfunctional requirement for providing documentation of our domain-specific language and a user guide we wanted to insure our source-controlled documentation compiles to static assets appropriately. Each of these utilities serves a unique purpose for instilling quality in our solution and has allowed us to move fast as we introduce new logic into our application to support additional requirements.

We encapsulated each of these steps within a shell script for local development as well as utilize it as the driver for our continuous integration pipelines via Travis CI. The team integrated the GitHub repository storing the project with Travis CI via webhooks, such that with each delivery Travis CI runs the quality checks against the branch at hand. Due to the team following a gitflow strategy for development, where we have a long running release branch and short-lived feature branches, we can rely on Travis CI to inform us on pull requests whether the new feature adheres to our quality standards. Travis CI has served as a critical piece of infrastructure for the team and the team has already triggered more than 500 builds for validating the solution. Alongside Travis CI, we integrated Travis CI with Codecov for maintaining code coverage reports of our solution. This enables the team to quickly inspect the health of the codebase and identify opportunities for improving code coverage.

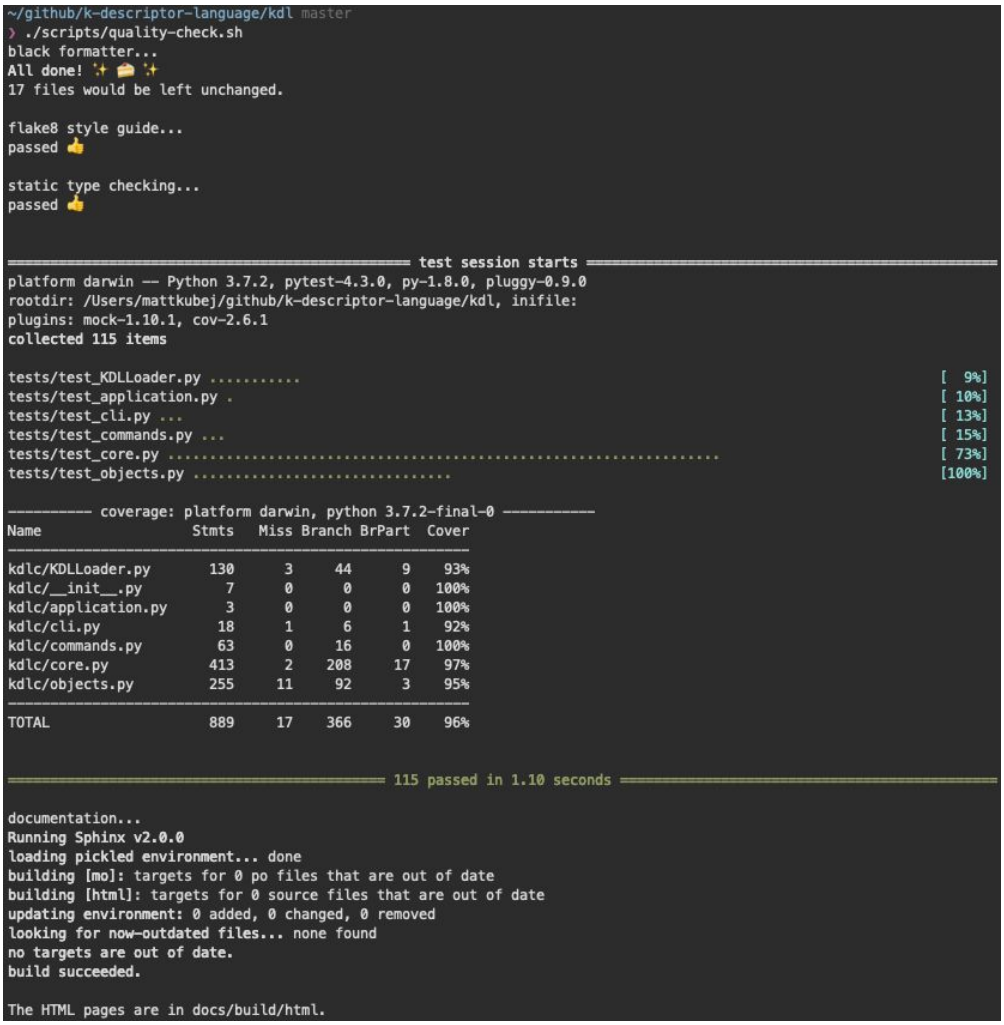


Figure 26: Quality check shell script validating style, syntax, and tests.

Figure 27: Travis CI pipeline running quality checks on the master branch.

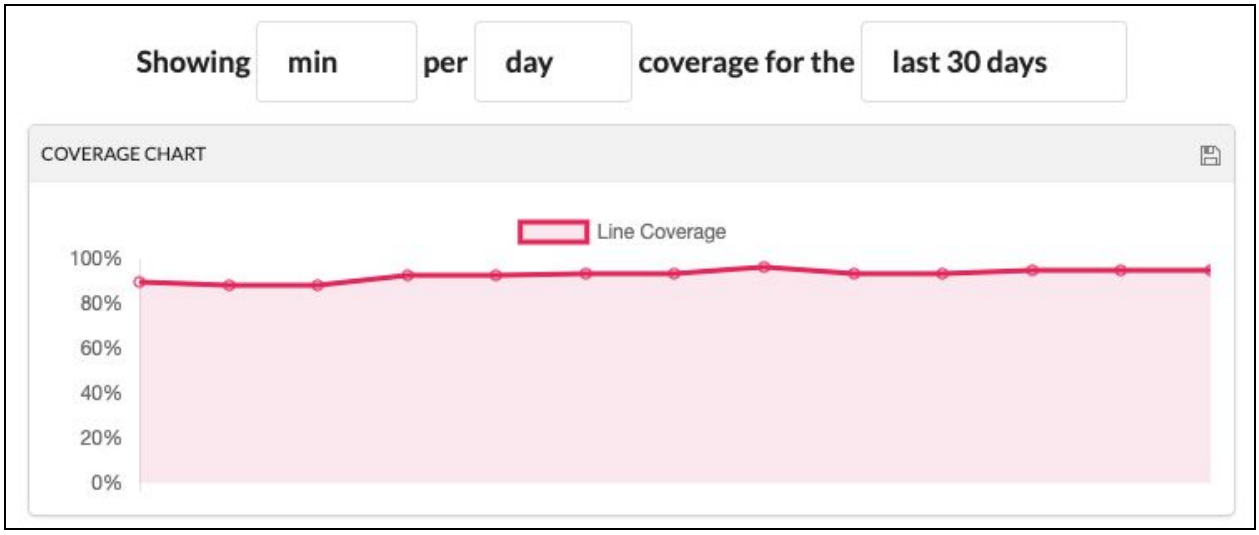


Figure 28: Codecov report illustrating test line coverage trends.

Nearing the end of the project, the team identified and introduced an additional tool for supplementing the quality control of our deliverable. We integrated our continuous integration pipeline with Codacy, which provided automated code reviews and additional static code analysis not previously found in our suite of quality checks. It provided insights on potential bugs, code duplication, code complexity, and highlights security issues, which the team subsequently addressed. Upon addressing the prominent issues identified by Codacy, the project received a letter grade of an A, which further instilled confidence in the quality of our codebase and robustness of our solution.

Figure 29: Codacy project overview on the master branch.

We surfaced quick feedback loops with this group of tooling via Slack integration and badges on our GitHub repository. A Slack bot provided by Travis CI prints results after each build to one of the team's

Slack channels. This provides a transparent view of development and serves as a mechanism for engaging the developers to act quick to failures before the issue continues to persist. The GitHub badge provides a similar level of transparency, but predominantly serves as a mechanism for announcing the current health of the codebase.

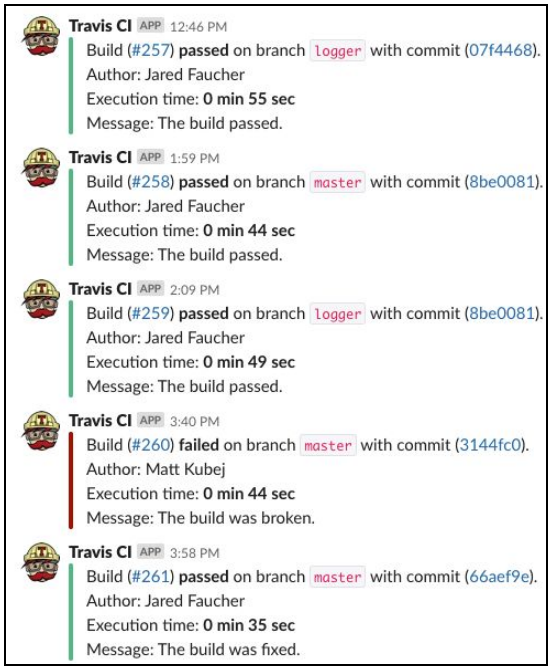


Figure 30: Travis CI reporting results to Slack.

Figure 31: Badges on KDL GitHub repository highlighting the health of the project.

Ultimately, each of these tools assists the team with measuring and monitoring the health of the solution. The short feedback loops allow the team to iterate quickly and validate the quality of code during efforts of refactoring or introducing new features. The early adoption and reliance on this toolset set the team on a path with guardrails, which allowed us to focus on the deliverable, identify issues early, and quickly remediate them.

Outside of the continuous integration pipeline, the team constructed a couple supplemental tests to highlight potential opportunities for improvement or issues with the running application. Specifically, this included performance testing and end-to-end testing. The performance test constructed a large workflow consisting of a user defined amount of nodes, which allowed the team to gauge the speed of the main operations of the application. The largest workflow experienced by the customer consisted of over 400 nodes, so the team doubled that count and received satisfactory results from the performance tests. The end-to-end testing served as deterministic validation of the compilation and decompilation functions by executing these operations on an installed kdlc instance and inspecting the outputs. We did not include these particular tests within our continuous integration pipeline due to the time cost and disinterest in

installing kdlc on the Travis CI nodes for execution. The team desired quick feedback loops from the continuous integration pipeline, so the installation of kdlc along with the running of these tests would greatly increase the length of the job and serve as a less effective information radiator. If the pipeline takes longer to run, then the development team loses interest and takes less action in responding to the feedback it provides.

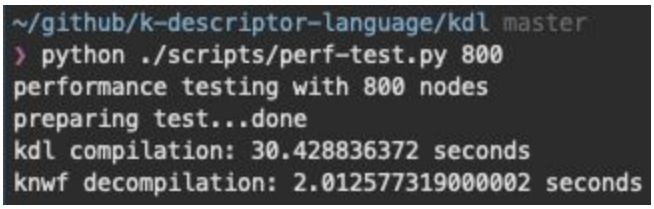


Figure 32: Performance test run with 800 nodes on a MacBook Pro.

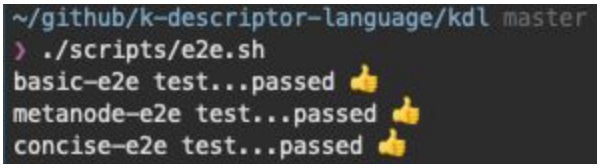


Figure 33: Example execution of end-to-end test suite.

Additionally, we incorporated manual testing to validate the generated workflows produced by our application opened within KNIME without errors and ran appropriately. The team established a test plan on Confluence to strategically divide and conquer the efforts for manually validating the large families of nodes with kdlc. Alongside this, the team utilized exploratory testing for generating workflows with a variety of node combinations. We specifically target the nodes the customer highlighted as important and the nodes he incorporated within the example workflows he initially presented us with. If we identify any issue, then we plug the issue by introducing additional integration and unit tests into our pipeline, thus preventing the issue from reappearing. Along these lines, exploratory testing allows the team to investigate how our customer may utilize the tool and allows us to discover further opportunities for improvement in our solution with regards to usability. We continuously strive to push as much of our testing as possible into an automated process, in order to garner time savings and eliminate human error.

Name	Does it Load in KNIME	Does it run in KNIME	KNIME Example	Template Created	Notes
IO					
IO/Read					
Excel Reader	Yes	Yes	05_Read_all_sheets_from_an_XLS_file_in_a_loop		
File Reader	Yes	Yes	01_Data_Loading_Example		
ARFF Reader					
CSV Reader	Yes	Yes	01_Data_Loading_Example	Yes - @Ivan Sinyagin	
Line Reader	Yes	yes	07_Visualize_Scatterplot_on_file		
Table Reader	Yes	Yes	06_Random_combination_of_two_sets_of_data		
PMML Reader	Yes	Yes	01_Example_for_Using_PMML_for_Transformation_and_Prediction		Updated existing Example workflow
Model Reader	Yes	Yes	_Learner_flow		Updated existing example to read written output model
Fixed Width File Reader					
List Files	Yes	Yes	07_Reading_Excel_Files		
Read Excel Sheet Names (XLS)	Yes	Yes	05_Read_all_sheets_from_an_XLS_file_in_a_loop		
Read Images	Yes	Yes	01_Network_Traffic_Reporting		
Explorer Browser	Yes	Yes	01_Example_Speech-to-Text		Could not test execution of the full workflow due to required API keys but this node was fine.

Figure 34: Subsection of manual test plan within Confluence.

Overall, with the number of quality checks in place, automated reports, and manual testing efforts, the team achieved a high level of confidence in the solution produced and maintained a working model for iteratively introducing new features with short feedback loops. Each of these tools and exercises served an important purpose in allowing the team to move quickly while attacking additional requirements and understand the impacts of each change. Due to the team's limited experience with Python these practices held the team's hand and assisted in ensuring the project instilled quality within the solution as well as abided by best practices.

Development Process

Estimates

The list below illustrates our initial estimates with our actual time for completion. We noted a set of an initial estimates moved out of scope in favor of new requirements with higher priority as discussed with our customer. The length of our sprints were one week.

Epic - kdl document to knwf archive

As a KDL user, I want to provide a standalone document as input to an application and receive a knwf archive as output, so that I can build KNIME workflows without using the graphical user interface.

User Story	**Estimates (Sprints)**	**Actual (Sprints)**
As a user, I want to write workflows with a specified subset of node types in KDL, so that I can create a knwf archive with them.	2.5	1.5
As a user, I want to connect one node to another node in KDL, so that I can connect nodes in a knwf archive.	0.5	0.5

Epic - knwf archive to kdl document

As a KDL user, I want to provide a knwf archive as input to an application and receive a kdl document as output, so that I can review workflows without using the graphical user interface.

User Story	**Estimates (Sprints)**	**Actual (Sprints)**
As a user, I want to convert knwf archives containing specified nodes into KDL, so that I can review the workflow in KDL.	2.5	1.5
As a user, I want to convert knwf archives containing node relationships into KDL, so that I can review the workflow in KDL.	0.5	0.5

Epic - support flow variables

As a KDL user, I want to compile KDL with flow variables and decompile knwf archives with flow variables, so that I can effectively pass parameterized variables within KDL.

User Story	**Estimates (Sprints)**	**Actual (Sprints)**
As a user, I want to decompile knwf archives with flow variables, so that I can review the workflow in KDL.	0.5	1
As a user, I want to compile KDL with flow variables, so that I can pass parameterized variables within KDL.	0.5	1

Epic - support meta nodes

As a KDL user, I want to compile KDL with meta nodes and decompile knwf archives with meta nodes, so that I construct more complex and real-world workflows with KNIME.

User Story	**Estimates (Sprints)**	**Actual (Sprints)**
As a user, I want to decompile knwf archives containing meta nodes into KDL, so that I can review the workflow in KDL.	1	1
As a user, I want to compile KDL documents containing meta nodes into knwf archives, so that I can build complex workflows in KNIME.	1	1
As a user, I want to decompile knwf archives containing wrapped meta nodes into KDL, so that I can review the workflow in KDL.	Not estimated (outside of MVP)	.5
As a user, I want to compile KDL documents containing wrapped meta nodes into knwf archives, so that I can build complex workflows in KNIME.	Not estimated (outside of MVP)	.5

Epic - REST Web Services node types

As a KDL user, I want to utilize KDL for employing the GET and POST REST Web Services node types, so that I can orchestrate and expose RESTful web services in workflows.

User Story	Estimates (Sprints)	Actual (Sprints)
As a user, I want to utilize KDL for employing the GET and POST REST Web Services node types, so that I can orchestrate and expose RESTful web services in workflows.	0.5	.25

Epic - node templates

As a KDL user, I want to allow the use of node templates within KDL, so that I am able to author KNIME workflows with reduced set of node settings.

User Story	Estimates (Sprints)	Actual (Sprints)
As a KDL user, I want to allow the use of node templates within KDL, so that I am able to author KNIME workflows with reduced node settings.	2	2

Epic - update knwf archive with kdl (no longer in scope for MVP)

As a KDL user, I want to provide a knwf archive with a kdl document containing updates to an application and receive an updated knwf archive, so that I manipulate workflows without using the graphical user interface.

User Story	Estimates (Sprints)	Actual (Sprints)
As a user, I want to connect/disconnect a node in a workflow automatically, so that I can change node relationships without the graphical user interface.	0.5	Out of scope
As a user, I want to automatically modify a node in a workflow so that I can change node attributes without the graphical user interface.	0.5	Out of scope

As a user, I would like to add a node to a workflow automatically, so that I can introduce new nodes to workflows without the graphical user interface.	0.5	Out of scope
As a user, I would like remove a node from a workflow automatically, so that I can remove nodes from workflows without the graphical user interface.	0.5	Out of scope

Based on our actuals, we overestimated the amount of work for supporting the requirements of defining KDL, compiling it to knwf archives, and decompiling knwf archives to KDL. Due to the number of unknowns involved with regards to our inexperience with KNIME and Python, we tended to overestimate with regards to our stories as a means of accounting for long running research and design efforts. Through dividing and conquering the requirements, the team managed to expedite production. Also, by building end-to-end prototypes the team achieved early wins at the advent of the project and established strong momentum for satisfying these initial requirements.

The team proactively negotiated requirements with the customer, and as a result the team reprioritized the list of requirements and identified requirements that provide more upfront value to our customer. By demonstrating progress to the customer and allowing the customer to work with the solution independently, the team and customer discovered some previously out of scope requirements were ultimately more valuable and could be accomplished before milestone three. During milestone three the team was able to complete all additional functionality scoped before the milestone as well as some features outside of the MVP, including supporting wrapped meta nodes within KDL, effectively building off of previously completed work.

Risks

Listed below highlight the risks we identified for our project and their state at the end of milestone three.

Unavoidable Risks

Risk	Impact	Description	Solution
Unfamiliarity of KNIME	Scope/Timeline	The team had limited exposure to KNIME due to quickly ramping up on the application from the inception of the project. This presented itself as a risk, because the team may	The team relied on knowledge of customer's expertise with KNIME. The team also created proof of concepts for KNIME specific interactions early within project, effectively mitigating any

		have uncovered initial unknowns during the technical implementation that could have lead to scope change. This could have lead to rework or unaccounted for scope.	unknowns and pain points as quickly as possible
Lack of time due to external conflicts	Timeline	Each team member has a day job and other external conflicts which had the potential to interfere with the progress of the project.	The team mitigated this risk by leveraging Slack for transparent and frequent communication about project timeline and deliverables. This allowed the team to proactively adjust as outside conflicts arose while allowing the team to effectively complete deliverables on time.
Team members in different timezones	Communication	The team consists of geographically dispersed individuals across multiple time zones, specifically three different time zones within the United States and another time zone within Asia. This introduced complications with scheduling meetings and staying synchronized across efforts.	The team relied on Slack, Jira, and Confluence to assist with remote collaboration.
New nodes created by KNIME or KNIME user	Tool Breaks	KNIME is an open-source solution that allows extension via the creation of new nodes. It is unknown how the KDL compiler will handle nodes not native to KNIME.	The team documented the expected XML structure of a node and highlighted any fields that are susceptible to slight changes. In addition to this the team tested the tool against a wide variety of nodes outside of the four main families and custom nodes. These tests did not uncover any gaps in the

			solution and has given us confidence in our solution to support the vast majority of node types within KNIME.

Intentional Risks

Risk	Impact	Description	Solution
Choice of programming language	Timeline	The team needed to make an early decision on the appropriate programming language to use to build the solution. Our customer preferred the use of Python due to familiarity amongst his coworkers and colleagues. However, the team did not have expertise in Python and had limited exposure to the language. The team lacked professional experience in this area, so the team needed to invest time to learn the language and risks introducing amateur mistakes into the project.	Teammates supported each other in the learning of Python 3. Independent exploration of the Python ecosystem to surface helpful tooling and libraries. Short feedback loops with our customer. The team created proof of concepts for any areas of concern within the technical implementation early in the project.

Each of the risks listed above have common underlying themes in relation to inexperience with KNIME, Python, and geographical dispersion. With regards to inexperience with KNIME, we entered the project with a significant amount of unknowns with regards to how the tool works, technical opportunities for solutions, and the feasibility of the requests of the customer in relation to KNIME. We followed through with our plan in relation to this risk by relying on our customer's expertise, spending significant time ramping ourselves up on KNIME, and prototyping our solution end-to-end within the early development of our project to mitigate this risk. This strategy gave us more confidence in our understanding and capability to deliver based on these actions. As for lack of experience with Python posing a risk to our project due to limited experience on the team, it presented itself as a time sink, which could impact our ability to deliver a working solution to our customer. We approached learning Python the same way we approached learning KNIME by spending significant time upfront learning and constructing proof of concepts of various technical unknowns that we required to support the requirements. The team obtained sufficient prowess to deliver at an adequate rate as validated by the positive feedback from our customer. Our team consisted of heavily geographically dispersed teammates that span across four time zones. We had three members in New England, one team member in Texas, one team member in California, and another team member in Singapore. Our ability to meet and communicate due to the variability in time

zones alongside each of us working full-time posed a risk to our project delivery, because it hindered our ability to effectively collaborate on a regular cadence. However, with the use of Jira, Confluence, and Slack the team established a high level of transparency, frequent touch points, and daily ad-hoc conversations as team members became available to combat this risk. By the end of the third milestone this risk was mitigated by the team as demonstrated by accomplishing not only our MVP deliverables, but by also introducing additional functionality to the customer outside of the MVP scope.

Team Dynamic

The team strongly favored transparent communication due to the geographic dispersion and a reliance on a number of tools including Slack, Confluence, Jira, and Zoom for effectively collaborating. The team utilized Slack for ad-hoc communications amongst themselves and the customer, which proved itself as the most valuable tool for collaborating. Within Slack, the team established a channel for customer communication, team communication, team announcements, and build statuses provided by the continuous integration environment in Travis CI. The team employed Confluence and Jira for gathering notes, documenting decisions, prioritizing a backlog of work, and sharing technical solutions. Confluence specifically served as a medium for maintaining a history of the project and allowed the team to effectively collaborate on the milestone deliverables. Lastly, Zoom served as a valuable tool for meeting with the teaching staff, the customer, and ultimately group meetings with the team. The face-to-face webcam meetings expedited the team's ability to norm as well as more effectively communicate information verbally and visually. The team met with the assigned teaching fellow on a weekly basis, met with the customer on a near weekly basis, and met as a team multiple times ad-hoc throughout the week. Each of these tools served their own purpose with regards to collaboration and greatly assisted in marching towards the goal of successfully delivering the requested requirements.

One challenge the team encountered revolved around conflicts outside of the capstone project and volatile levels of contribution within the team. This particular challenge made it difficult for the team to adequately plan work for each week, which put the team's ability to deliver at risk. The team attempted to combat this issue by constructing backlogs of work and assigning ownership as a means of enforcing responsibility and tracking progress. Another technique the team attempted includes utilizing a scrum stand up page for providing improved transparency of each individual's efforts and whether any individual required help. However, the overhead of maintaining this document and lack of contribution from the entire team exposed this strategy as ineffective. Eventually, the team engaged in an open and pointed discussion on the issue, which served as the catalyst for achieving the desired level of inclusion. The team effectively normed after this event and had more productive collaboration by employing more regular real-time discussions. Through the advent of more inclusive working sessions, the team's morale improved and led to a successful finish of the project.

Lessons Learned Reflection

The team has unearthed a number of lessons learned through the advent of the capstone project, which serve as beneficial carryover into professional software engineering. Specifically, the team found great value in maintaining short feedback loops with our stakeholder, the learnings elicited from iterative prototyping, the importance of continuous integration for instilling quality in a solution, and the team's gratefulness for selecting Python as the implementation language per the recommendation of the customer. Also, near the culmination of the project, the team learned a substantial amount about the importance of engaging in hard discussions about effectively working as a team and voicing concerns within a project. Due to an exposure of these concepts through the class, the team feels as though these learnings easily translate to their engineering careers and provide opportunities for utility immediately. These lessons learned surfaced through the well-rounded exposure to the software development lifecycle encapsulated within the course as well as the experience of solving a challenging software problem with a newly formed group of diverse individuals.

An immediately apparent valuable lesson the team encountered stemmed from establishing a working relation to a remote customer as well as collaborating remotely amongst each other. The team found by communicating with the customer via pseudo code, working prototypes, and diagrams on a regular cadence allowed the team to cement their understanding of the requirements and appropriately prioritize the work. These short feedback loops reduced the risk of spending unnecessary hours on misunderstood objectives or items the customer found no value in. Alongside this value, the team also utilized this strategy for preventing scope creep by having transparent communication with the stakeholder via living documentation of the requirements. This greatly assisted the team in maintaining a narrow focus on the prominent requests of the customer.

With regards to prototyping, the team began the project with a number of unknowns and inexperience with KNIME as well as Python. Through prototyping end-to-end solutions during initial development of the project, the team gained confidence in an ability to effectively build solutions with Python and deliver on the requirements outlined by the customer. Similarly, the team mustered a stronger understanding of the internals of KNIME through iterating on a number of potential solutions. Ultimately, this allowed the team to fail quickly and adjust their path towards a more appropriate solution.

In order to validate the quality of the solution the team produced; the team scaffolded a continuous integration pipeline with a number of popular quality checks the team identified as they explored the Python ecosystem. Specifically, by introducing this pipeline at the beginning of the development efforts, it instilled quality within the product from the start and eliminated the need to circle back on these efforts. This tooling quickly surfaced syntactical issues, styling issues, validating functionality through testing, and documentation generation. The continuous integration pipeline served as an alarm in the development process for when an issue entered the product and allowed the team to self-police by preventing functionality with errors from entering a release. The team received a high level of confidence from the work they produced by employing these mechanisms within the infancy of the project and served as a safety net for the team due to the inexperience with Python.

Furthermore, even though the team had initial hesitation in employing Python as the primary language for feature development, the team fully embraced the language and has found great value in it. The low barrier of entry in utilizing the language to produce solutions greatly expedited the team's ability to deliver. Even though the team had a slight learning curve at the beginning of the project, the team overcame it and found that the project requires little effort to produce a tangible deliverable that offered value to the customer. Tangentially, the rich Python ecosystem has a surplus of libraries that supports the various needs of the application, which included templating, XML parsing, syntax grammar generation, and tooling for quality enforcement. As a result, the team could quickly iterate on new ideas and provide working solutions in short periods of time. Going forward, the team sees this language as another great tool for prototyping ideas as well as creating supporting tooling in other software engineering endeavors.

Lastly, the team unearthed an extremely valuable lesson near the tail-end of the project with regards to forming as a team, effectively working together, and having difficult conversations amongst each other. As a result of varying levels of contribution during the lifetime of the project and inconsistent understandings of expectations across the team, concern arose around effectively completing deliverables and instilling the appropriate level of inclusion within work. The team found engaging in directed conversations about the issue and voicing concerns in a professional manner provided a positive path forward for finishing the remainder of the project strongly as a team. Ultimately, the team learned the importance of addressing these concerns early and directly, even though the conversation may introduce discomfort.

Appendix

User Manual

The kdlc application runs on Python 3.7. The details below illustrate an installation procedure and how to utilize the application for working with KDL.

Installation Procedure

1. Install Python 3.7 from https://www.Python.org/
2. Install Git from https://git-scm.com/
3. Open a terminal
4. Navigate to a working directory with `cd`
5. Clone the kdlc project by executing
   `git clone git@github.com:k-descriptor-language/kdl.git`
6. Navigate into the kdlc project space by executing `cd kdl`
7. Install kdlc by executing `python3 setup.py install`

Features

The following sections demonstrate the various available operations with kdlc.

Help

Execute `kdlc --help` within your terminal for an overview of available flags.

Compile KDL to knwf Archive

Execute kdlc with a KDL as the input argument and define a filename for the output knwf archive.

```
kdlc -i complex.kdl -o workflow.knwf
```

Compile KDL to knwf Archive with a custom templates path

Execute kdlc with a KDL document as the input argument and define a filename for the output knwf archive.

```
kdlc -i complex.kdl -o workflow.knwf -tp PATH_TO_CUSTOM_TEMPLATE_LIBRARY
```

Decompile knwf Archive to KDL

Execute kdlc with a knwf archive as the input argument and define a filename for the output KDL document.

```
kdlc -i complex.knwf -o workflow.kdl
```

Debug mode

Execute kdlc in either direction with the debug flag for additional debug logging statements to be printed to stdout.

```
kdlc -i complex.knwf -o workflow.kdl -d
kdlc -i workflow.kdl -o workflow.knwf -d
```

Developer Manual

The project team established a handful of supporting development tools and scripts, which can be found within the scripts folder.

Before executing any of the scripts, download the development dependencies by executing the following command.

```
pip install -e '.[dev]'
```

build-docs.sh

In order to construct the documentation found at https://kdl.readthedocs.io, the team has utilized sphinx for generating this static website. It takes the rst files as defined within docs/source of the project's root directory as input to generate the static assets. You can learn about it at http://www.sphinx-doc.org.

To build the documentation for viewing locally, then execute scripts/build-docs.sh from the root directory of the project.

To view the generated documentation, then open docs/build/html/index.html from the root directory into a web browser.

e2e.sh

The end-to-end script provides a deterministic test for validating the kdlc application by compiling kdl and decompiling knwf archives.

To run the script, firstly install kdlc by following the kdlc installation instructions. Then execute scripts/e2e.sh from the root directory of the project.

format.sh

The format shell script runs an opinionated code formatter across the project called black. More information about black can be found at https://github.com/ambv/black.

To run the code formatter, then execute scripts/format.sh from the root directory of the project.

generate-parser.sh

To support the parsing of KDL, the team elected to utilize ANTLR (ANother Tool for Language Recognition). The generate parser shell script executes ANTLR against our defined grammar files within grammar folder found within the root of the project.

Before executing the shell script, you must install ANTLR first in the matching location found within the shell script. The installation instructions for ANTLR can be found at https://www.antlr.org/. At the time of writing this documentation, ANTLR is on version 4.7.2 and our shell script expects you to place the jar within the /usr/local/lib directory.

To run the generate parser shell script after installing ANTLR, then execute scripts/generate-parser.sh from the root directory of the project.

perf-test.py

To performance test kdlc with load, the performance test script constructs a KDL document with a string of node connections, then compiles the document to a knwf archive and subsequently decompiles the archive to a KDL document while capturing time metrics. The script helps identify performance degradation as a result of implementation decisions.

To run the script, firstly install kdlc by following the kdlc installation instructions. Then run `python3 scripts/perf-test.py` to run with 2 nodes by default or provide an argument to run with a desired amount of nodes, such as `python3 scripts/perf-test.py 10`.

quality-check.sh

The quality check runs a suite of tooling to insure the quality of our deliverable. This particular script gets run within our continuous integration pipeline on code deliveries.

It runs the following tooling and will exit on failure.

- black for code formatting
- flake8 for style guide enforcement
- mypy for static type checking
- pytest for test execution
- sphinx for validating documentation compilation

To run the quality check script, then execute scripts/quality-check.sh from the root directory of the project.