MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Orchestration tool for the modular computing subsystem of the ANALYZA platform

Master's Thesis

BC. RICHARD KYČERKA

Brno, Spring 2022

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Orchestration tool for the modular computing subsystem of the ANALYZA platform

Master's Thesis

BC. RICHARD KYČERKA

Advisor: RNDr. Tomáš Rebok, Ph.D.

Department of Computer Systems and Communications

Brno, Spring 2022


FACULTAS ARTIS INFORMATICÆ
UNIVERSITAS MASARYKIANA

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Richard Kyčerka

Advisor: RNDr. Tomáš Rebok, Ph.D.

Acknowledgements

I would like to thank my supervisor, RNDr. Tomáš Rebok, Ph.D., for his advice, help and support during my work on this thesis.

Abstract

In recent years, the concept of 'Big Data' gained the attention of many analysts, since the ability to collect and process data on a massive scale is getting more and more accessible. Many projects focused on Big Data emerged and one of those projects is the ANALYZA platform, which focuses on cross-domain analysis of heterogeneous data. Analysis steps are isolated mini-applications, composed into data workflows. The aim of this thesis was to review the field of workflow orchestration tools mostly operating in the Kubernetes platform and pick the most suitable one to adopt by the ANALYZA platform. The choice fell on Argo Workflows, which has been proven to be the most suitable tool to use in ANALYZA platform. The thesis justifies the choice by detailed analysis of Argo Workflows capabilities and demonstrates it on a real world data workflow use case.

Keywords

workflow, orchestration, Docker, Kubernetes, data analysis, Big Data, Airflow, Prefect, Argo Workflows, Snakemake, ANALYZA

Contents

List of Tables

List of Figures

Introduction

In recent years, Big Data has gained considerable attention from governments, academia, and enterprises. The term "big data" refers to collecting, analyzing, and processing voluminous data. Having large data sets and reasonable computational power, many projects emerged that took the chance and implemented various tools to analyze these data sets.

One such project is ANALYZA[1] platform originated as research at Masaryk University for the Police of Czech republic with a focus on cross-domain data analysis. Project ANALYZA use-cases include image processing or network data analysis. ANALYZA platform covers the whole process of data analysis, from storage in specialized distributed data lakes, through the complex analysis steps to visualization in specialized component providing various analytic functions and view on analysis results.

Individual analysis steps are stand-alone mini-applications, where one complex analysis may combine multiple steps into data workflows. The result of such analysis may be various views on the data and their relationships. Workflow steps need to be executed in the correct order to respect dependencies between steps. This composition may be non-trivial and specialized orchestrator service, as a part of the ANALYZA platform, is taking care of this scheduling.

The goal of this research is to analyze the field of workflow orchestration tools and pick one to adapt by ANALYZA platform to succeed its in-house built orchestration solution. The research puts emphasis on functionality portfolio, scalability, security, and usability in production environments. The new orchestration tool functional requirements are then demonstrated in various examples based on the workflow use case style used in ANALYZA project.

The thesis can be divided into three parts. The first part, i.e. chapters 1 to 6, is focused on research in the field of well-known workflow orchestration tools and provides a brief overview of the most common ones. The overview shows how are the workflows defined, what is the deployment and execution strategy and discussion about community and documentation.

The second part, chapter 7, introduces ANALYZA project architecture together with deployment and workflow execution strategy. Architecture analysis leads to a list of requirements that the new orchestration engine must fulfill.

The final part, chapter 8 and onwards, introduces the orchestration tool choice, describes its functionalities in detail with examples, and is finalized by a demonstration on a realistic use case.

1 Big Data analysis

Big data represents the large, diverse sets of information that grows at an exponential rate. Unfortunately, big data is so large that none of the traditional data management tools can store or process it efficiently. More than the volume of data, the way organizations utilize data matters.

1.1 Cloud computing

One of the main challenges in developing Big Data processing solutions is to define the right architecture to deploy Big Data software in production systems. Big Data systems, are large-scale applications that handle online and batch data that is growing exponentially. For that reason, a reliable, scalable, secure and easy to administer platform is needed to bridge the gap between the massive volumes of data to be processed, software applications and low-level infrastructure.

Kubernetes[1] is one of the best options available to deploy applications in large-scale infrastructures. Using Kubernetes, it is possible to handle all the online and batch workloads required to feed analytics and machine learning applications. Kubernetes works with virtualized containerized applications, where the most popular is virtualization technology is Docker[2]. Application containerizing provides portability and scalability needed in order to fully utilize cloud computing.

1.2 Workflows

In the data science field, when working with Big Data, the measurements often need to undergo some kind of transformation to provide the analysts with various views on the data.

These transformations can be called data workflows or pipelines. Workflows are composed of set of tasks, where tasks are single units of work in data processing chain.

---

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

[2] `https://docs.docker.com/get-started/overview/`

Responsibility of a task can be for example downloading a dataset from a website into a local data warehouse. Another task can take this data and do some corrections, like unifying formats of records in data sets and saving results into a local database. The next task can take the data from the database and run machine learning model training. All these tasks together form a workflow.

Intuitively, these steps need to be executed in the correct order, since one can't run machine learning model training before downloading data sets from the website. This implies natural dependencies between tasks.

For easier representation, an abstraction of Directed Acyclic Graphs (DAG)[2] has been introduced, where graph nodes represent tasks involved in the process and edges represent dependencies between tasks.

Workflow management tools are here to help data scientists with data transformations. Each tool has its own implementation of tasks and workflows, but the idea of DAGs remains the same.

2 Airflow

Apache Airflow is a workflow management tool founded by Airbnb in October 2014, later open-sourced under Top-Level Apache Software Foundation[1] project. Airflow was one of the first widely used tools of the workflow management kind. In December 2020, the massive upgrade to Airflow 2.0 was released, targeting the biggest issues, like scheduler performance or the introduction of TaskFlow API allowing task definition in a functional way.

2.1 Steps

Workflow steps, or *tasks* in Airflow terminology, can be arbitrary pieces of Python code. There are two ways to define tasks, depending on the Airflow version. Both task types can be combined together within one workflow.

2.1.1 Pre-TaskFlow era tasks

The original Airflow implementation of tasks was heavily relying on *Operator*[2] usage. Operators are pre-defined templates for tasks and Airflow comes with a set of pre-defined Operators, but user can write his own custom Operators. An example of an Operator can be *PythonOperator*, which takes the Python function as an input argument and transforms the function into a workflow step.

[1] `https://www.apache.org/`

[2] `https://airflow.apache.org/docs/apache-airflow/stable/concepts/operators.html`

```
def print_function(x):
    print x

t1 = PythonOperator(
    task_id = 'print',
    python_callable = print_function,
    op_kwargs = {"x" : "Hello World"},
    dag=dag,
)
```

Listing 2.1: Airflow task definition in pre-TaskFlow era.

2.1.2 TaskFlow era tasks

The TaskFlow API introduced the *@dag* and *@task* concepts, creating the abstraction, where steps are @task decorated functions, and workflows are @dag decorated functions calling multiple @task functions inside. This approach has greatly reduced the verbosity of definitions and automatically dependency calculations.

```
@task()
def print_function(x):
    print x
```

Listing 2.2: Airflow task definition in TaskFlow era.

2.1.3 XComs

Airflow's XComs[3] are the inter-task communication mechanism. It can be thought of as a shared board, where tasks push to and pull information from. Having two tasks A and B, where A calculates some value, push it to the shared board and task B pulls the value and adjusts its own calculation based on this information.

It's important to remember that XComs are designed to transfer just small amounts of metadata rather than big volumes of data.

[3] https://airflow.apache.org/docs/apache-airflow/stable/concepts/xcoms.html

2.2 Workflows

DAG[4] is a collection of tasks, with some additional flow properties, like schedules or parameters. DAG definition again differs depending on whether TaskFlow API is used or not. DAGs are stored in the metadata database, which is part of the Airflow deployment. Submission of DAGs can be done via either REST API, UI, or CLI.

2.2.1 Pre-TaskFlow era DAGs

DAGs defined in the traditional way is a DAG object with a collection of Operators and DAG properties. Dependencies are set as upstream/downstream between operators, signalized by » sign, e.g. *t1»t2* signalizes *t2* is dependent on *t1*.

```
1  def print_function(x):
2      print x
3
4  with DAG(
5      'example',
6      default_args={
7          'depends_on_past': False,
8          'email': ['airflow@example.com'],
9          'email_on_failure': False,
10         'email_on_retry': False,
11         'retries': 1,
12         'retry_delay': timedelta(minutes=5),
13     },
14     description='Hello world DAG',
15     schedule_interval=timedelta(days=1),
16     start_date=datetime(2022, 1, 1),
17     catchup=False,
18     tags=['example'],
19 ) as dag:
20     t1 = BashOperator(
21     task_id='print_date',
22     bash_command='date',
23     dag=dag,
24     )
25     t2 = PythonOperator(
26     task_id = 'print',
27     python_callable = print_function,
28     op_kwargs = {"x" : "Hello World"},
29     dag=dag,
30     t1 >> t2
31 )
```

Listing 2.3: Airflow DAG definition, pre-TaskFlow

[4] https://airflow.apache.org/docs/apache-airflow/stable/concepts/dags.html

2.2.2 TaskFlow era DAGs

Utilizing the *@dag* and *@task* decorators, the DAG definition becomes a lot clearer, better readable, and easier to develop. Dependencies can be set manually, or when possible, they are calculated by Airflow.

```
1  @dag
2  def example():
3      t1 = BashOperator(
4      task_id='print_date',
5      bash_command='date',
6      )
7      @task
8      def print_function(x):
9          print x
10     @task
11     def generate_number():
12         return random.randint(0, 10)
13
14     random_number = generate_number()
15     random_message = print_function(random_number)
16     t1 >> random_message
17 )
```

Listing 2.4: DAG example using TaskFlow API

2.3 Deployment

Airflow deployment is quite complex, so the preferred type of deployment is using Helm[5] charts to deploy in the Kubernetes cluster or managed installation, where the user pays someone else to host and maintain the installation in the SaaS model.

For development purposes, local deployment can be done via Docker-Compose or running Airflow in a Python environment.

[5] https://helm.sh/

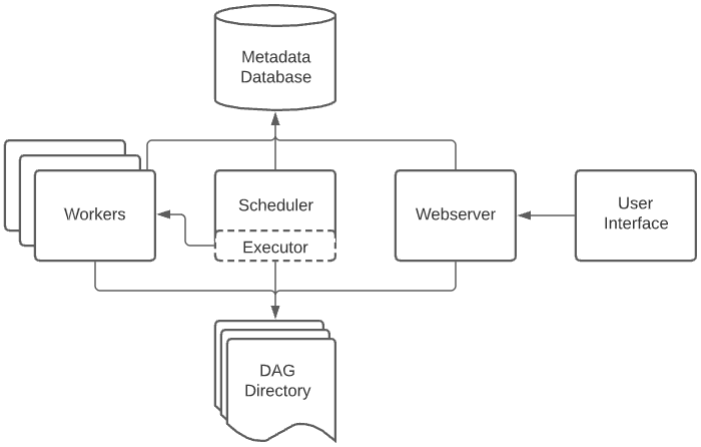


Figure 2.1: Airflow architecture schema
Source: `https://airflow.apache.org/docs/apache-airflow/stable/concepts/overview.html`

2.3.1 Executors

DAGs are processed by executors[6], which are processes running inside the Scheduler process and can be divided into local and remote types. Local executors keep the task execution under the scheduler process. Remote executors delegate the work to Dask[7], Celery[8] backend or Kubernetes. Executor choice is installation-wide and can be set in the Airflow configuration file.

Tasks from a single flow can, but don't necessarily need to be executed on the same machine.

Kubernetes executor

Kubernetes executor is a process running under the Scheduler process that has access to a Kubernetes cluster. When a DAG is invoked via Kubernetes Executor, the executor requests Kubernetes API for a worker

[6] `https://airflow.apache.org/docs/apache-airflow/stable/executor/index.html#executor`
[7] Dask is an open-source library for parallel computing - `https://dask.org/`
[8] `https://docs.celeryq.dev/en/latest/index.html#`

Pod where the task is executed and after execution and reporting result, the pod is dismissed.

2.4 Documentation and community

Airflow documentation is quite extensive since there are many concepts to be explained. Diagrams and code snippets are here to help with the understanding of individual concepts. However, some documentation sections are quite confusing and somewhat hard to understand.

One of the biggest advantages of Airflow is its massive community built over the last years. The community is also very active in contributing to the Airflow codebase[9] with hundreds of discussed issues on GitHub and also many open pull requests fixing bugs or suggesting improvements.

2.5 Summary

Apache Airflow is a mature, heavy-armed orchestrator with tradition and many users accumulated during the last years.

Users considering using Airflow should be aware of a steep learning curve and somewhat overwhelming installation process. This is mainly caused by many architectural changes over the years. In the near past, it feels like it's just trying to catch up on the competitors that started later than Airflow with fresh and better-designed architecture (taking advantage of Airflow design flows knowledge), rather than coming up with new ideas.

For those who can overcome the initial struggles, Airflow can deliver a reliable and robust service.

[9] `https://github.com/apache/airflow`

3 Prefect

Prefect is an open-source workflow management system. The main motivation for Prefect to rise was dissatisfaction with some of the Airflow's concepts (at that time the go-to workflow management system), like writing DAGs in an imperative way, weak DAG scheduling mechanism, or inability of passing big data chunks between tasks. Prefect chose Python as the language of choice for both, writing workflows and the platform itself.

Prefect allows user to define tasks and flows in two ways; the imperative and functional. The more elegant and more convenient is the functional way; it brings the advantage of having less boilerplate code and better readability. On the other hand, the imperative declaration offers more fine-grained control over tasks and flows. In later sections, only the functional task API will be considered.

3.1 Tasks

In Prefect, tasks are defined as Python functions annotated by *@task* decorator[1]. Task level properties like the number of retries or task identification are passed as arguments to the decorator.

Prefect comes with a decent library of mostly community-driven pre-defined tasks providing the ability to integrate with third-party services[2]. For reference, such tasks could execute a shell script, create a Jira issue or post a message via Slack.

[1] `https://docs.prefect.io/core/concepts/tasks`
[2] `https://docs.prefect.io/core/task_library`

3.2 Workflows

Flows[3] are Prefect's implementation of DAGs. Tasks are instantiated inside of a flow object, possibly using the return value of one task as input for other tasks, creating a natural dependency between these tasks. Dependencies are then automatically detected by Prefect.

The development and debugging can be done locally[4] on the user's machine without any available Prefect deployment.

In order to run a workflow on running deployment, it needs to be registered to the Prefect backend using the Prefect CLI[5].

```
1  @task(name="number generator")
2  def generate_number():
3      return randint(0,22)
4
5  @task(name="say task")
6  def say(x):
7      print(x)
8
9
10 with Flow('Simple Prefect flow') as flow:
11     number = generate_number()
12     another_number = generate_number()
13     say(number + another_number)
```

Listing 3.1: Prefect workflow example

3.3 Deployment

Prefect architecture is divided into two parts, backend, and agents. Backend stores metadata about registered flows, provides UI, and takes care of task scheduling. Flows are executed on agents, which can be deployed in several ways, depending on the use case and available infrastructure. Prefect comes with two implementations of backends, the cloud-managed called Prefect Cloud and the on-premise one called Prefect Core.

[3] https://docs.prefect.io/core/concepts/flows
[4] https://docs.prefect.io/core/getting_started/basic-core-flow.html
[5] https://docs.prefect.io/api/latest/cli/register

3.3.1 Prefect Cloud

Prefect Cloud encapsulates all the backend components into a cloud-managed solution. This means the user doesn't need to deploy in his infrastructure anything but one or more of the agent variants. Prefect Cloud is easy to set up and enables user to run his first workflow in a few moments. Cloud backend is hosted on Prefect side and there are several pricing options including a free one, although with some restrictions like a limited number of registered flows or a limited number of flow runs per month.

3.3.2 Prefect Core

The core version of Prefect is basically a lightweight on-premise version of Prefect Cloud. Depending on the desired scale, Prefect can be deployed on a local machine using *docker-compose* or scaling large in Kubernetes cluster using *helm charts*[6].

The *docker-compose* option is more suitable for evaluating and testing whether Prefect suits the user's needs rather than running in production. For production deployment, the Kubernetes deployment seems to be the only viable option.

Helm charts are provided by the Prefect team which makes the charts more trustworthy, documentation related to Prefect core Helm charts is well written and covers modifications to the default configuration.

[6] `https://github.com/PrefectHQ/server/tree/master/helm/prefect-server`

3.3.3 Prefect Cloud vs Server

The cloud solution provides many advantages compared to Server[7]:

- Authorization and Permissions – the ability to manage users authenticated through Auth0[8], their roles and team assignment
- Available to access from everywhere – as long as the user has a valid API key
- Better performance and scaling
- Effortless maintenance and upgrades
- Premium support

3.3.4 Agents

Agents are processes, where flows are being executed. All agents need to be registered either in Prefect Cloud or Prefect core backend. Registered agents periodically poll the backend for work scheduled to be done.

Agents can be deployed on user's local machine, executing flows as processes or Docker containers, but also in Kubernetes, where flows are executed as Kubernetes jobs or AWS ECS, where flows are executed as AWS ECS Tasks.

While flows are being executed on agents, the work can be delegated further to Dask using *Executors*[9].

3.4 Documentation and community

Prefect documentation is very well written, enriched by many code snippets, and easy to understand. Its content can be divided into two parts; one looking at the concepts with higher abstraction and the second explaining concepts in more detail. User can find code examples, video and written tutorials. Many companies wrote on their blogs[10]

[7] https://docs.prefect.io/orchestration/server/overview.html#prefect-server-vs-prefect-cloud-which-should-i-choose

[8] https://auth0.com/

[9] https://docs.prefect.io/api/latest/executors.html#executors

[10] https://www.prefect.io/why-prefect/case-studies/

about their experiences with Prefect and why they chose it or migrated from other tools.

Prefect has grown a big community reflecting in quick answers on Slack or contributions on GitHub.

3.5 Prefect Orion

On October 5th, 2021, Prefect announced the new generation of their product, Prefect Orion[3] along with technical alpha, expected to release in early 2022. The objective of Orion is to introduce dynamic, DAG-free workflows; a better development experience, and transparent and observable orchestration rules.

3.6 Summary

Prefect is a great workflow orchestration tool for users who can afford to use the Prefect Cloud license model and are familiar with Python programming. Especially, if the user can make use of Dask integration. The transition from pure Python code to Prefect workflow is intuitive and doesn't require many changes to the existing code.

The advantage of being able to run workflows without taking care of backend infrastructure is huge and unique in the field of orchestration tools. The setup process is clear and straightforward, the development process is user-friendly with the option to test and debug workflow code locally.

The concept documentation is brief, yet it tells the user everything he needs to know.

Prefect team is actively pushing forward the feature development giving the promise of future proof investment of time and effort in adopting Prefect as the go-to workflow orchestrator.

4 Snakemake

Snakemake[1][4][5] is a tool to create reproducible and scalable data analyses created by bioinformatics, but aiming to be understandable by everyone. The implementation of DAGs is highly influenced by the GNU Make[6] paradigm; workflow is a set of rules applied in a chain to transform inputs to outputs executed in isolated Conda[2] environments.

4.1 Tasks

Snakemake's implementation of tasks is called *rules*[3]. Rules are written in a special file named *Snakefile* using custom, Python-based language[4], aiming to be clear and human-readable without boilerplate code around.

The main rule components are name, input, output, and action. Some additional rule properties are available too, like specifying Conda environment, output log file location, or parameters.

```
1 rule NAME:
2     input: "path/to/inputfile"
3     output: "path/to/outputfile"
4     shell: "somecommand␣{input}␣{output}"
```

Listing 4.1: Snakemake rule template, executing shell command

The rule applies the action by taking the input and storing the output in specified locations. The action can be a shell command, Python script, or scripts in other languages like R, Julia, or Rust. Some users may appreciate the integration with Jupyter[5] notebooks too.

[1] https://snakemake.github.io/

[2] https://docs.conda.io/en/latest/

[3] https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html

[4] https://snakemake.readthedocs.io/en/stable/snakefiles/writing_snakefiles.html#grammar

[5] https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#jupyter-notebook-integration

Input and output files don't need to be on the local host machine, it could point to remote storages[6] like S3, FTP or HTTP endpoint. This remote storage is accessed via special functions called wrappers.

Wrappers[7] encapsulate general actions that can be reused. There is also a central wrapper repository where users can share their wrappers with others.

Inputs and outputs can use wildcards[8] and regular expressions to make rules generic e.g. by processing all files with common pattern.

Each rule can have its own isolated run environment provided by Conda or taking the isolation even further, each rule can be run in its own container.

4.2 Workflows

The user doesn't specify what the workflow looks like, he just specifies the target file and Snakemake determines rules that need to be applied to generate the output. Alternatively, the user can specify the rule to be executed. Invocation of workflows is done via CLI.

Snakemake workflows offer high customization in terms of resources that can be provided to rule execution. Such options include the number of threads, amount of memory, and disk usage.

4.2.1 Workflow execution

Workflow execution can be held locally on the user's machine or the execution could be delegated to the cloud or cluster. When executing remotely, inputs and outputs are stored on a shared filesystem or remote storage like S3.

Workflows also support caching to avoid repeated workflow evaluations on the same inputs. When executing in the cloud, these subresults are stored on external storage.

---

[6] `https://snakemake.readthedocs.io/en/stable/snakefiles/remote_files.html#remote-files`

[7] `https://snakemake.readthedocs.io/en/stable/tutorial/additional_features.html?highlight=wrapper#tool-wrappers`

[8] `https://snakemake.readthedocs.io/en/stable/snakefiles/rules.html#wildcards`

Workflow runs can be examined via generated HTML report that includes information like runtime statistics or visualization of the workflow topology.

4.3 Deployment

Snakemake doesn't require any kind of long-running backend. Workflow executions are invoked by the user on demand. The execution environment is specified as an option to the CLI invocation.

4.3.1 API

Since Snakemake doesn't offer any centralized server thus no API is exposed by default. However, there's an option to run workflows with special flag to send execution details to a Panoptes[9,10] server. However, this feature is still under development.

4.4 Documentation and community

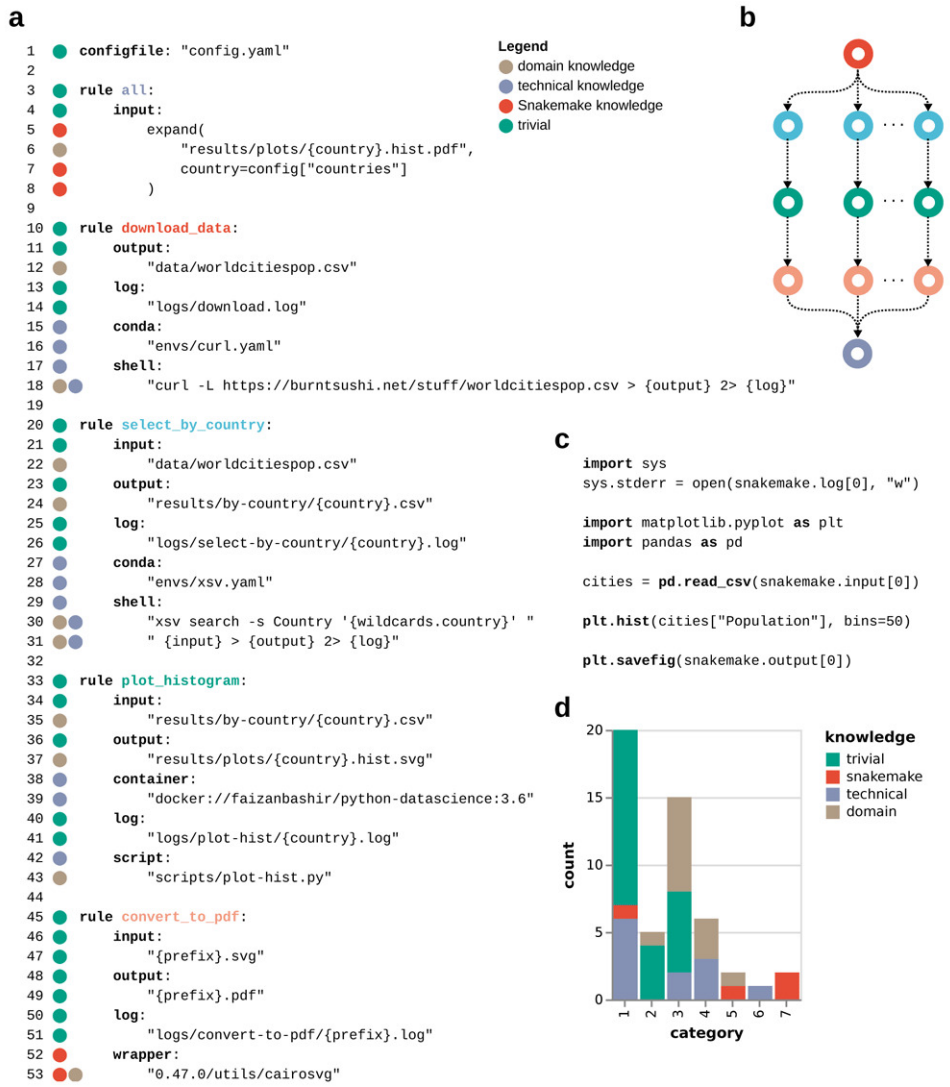
Snakemake's documentation[11] is detailed and well written. It includes tutorials with examples precisely explaining the workflow lifecycle.

The community is built mainly by bioinformatics and many scientific papers reference Snakemake as their choice for data processing tool used in the research. Questions about Snakemake are mainly discussed on StackOverflow, currently capturing more than a thousand threads.

[9] `https://getpanoptes.io/`

[10] `https://snakemake.readthedocs.io/en/stable/executing/monitoring.html`

[11] `https://snakemake.readthedocs.io/`




```
configfile: "config.yaml"

rule all:
    input:
        expand(
            "results/plots/{country}.hist.pdf",
            country=config["countries"]
        )

rule download_data:
    output:
        "data/worldcitiespop.csv"
    log:
        "logs/download.log"
    conda:
        "envs/curl.yaml"
    shell:
        "curl -L https://burntsushi.net/stuff/worldcitiespop.csv > {output} 2> {log}"

rule select_by_country:
    input:
        "data/worldcitiespop.csv"
    output:
        "results/by-country/{country}.csv"
    log:
        "logs/select-by-country/{country}.log"
    conda:
        "envs/xsv.yaml"
    shell:
        "xsv search -s Country '{wildcards.country}' "
        " {input} > {output} 2> {log}"

rule plot_histogram:
    input:
        "results/by-country/{country}.csv"
    output:
        "results/plots/{country}.hist.svg"
    container:
        "docker://faizanbashir/python-datascience:3.6"
    log:
        "logs/plot-hist/{country}.log"
    script:
        "scripts/plot-hist.py"

rule convert_to_pdf:
    input:
        "{prefix}.svg"
    output:
        "{prefix}.pdf"
    log:
        "logs/convert-to-pdf/{prefix}.log"
    wrapper:
        "0.47.0/utils/cairosvg"
```

```
import sys
sys.stderr = open(snakemake.log[0], "w")

import matplotlib.pyplot as plt
import pandas as pd

cities = pd.read_csv(snakemake.input[0])

plt.hist(cities["Population"], bins=50)

plt.savefig(snakemake.output[0])
```

Figure 4.1: Snakemake workflow example

(a) section shows how a workflow looks like, with each line labeled with color depending on user knowledge, (b) DAG of jobs, the color of nodes are corresponding with colors of rule names, (c) content of plot-hist.py script referenced in 'plot_histogram' rule, (d) knowledge requirements for readability by statement category.

Source: `https://f1000research.com/articles/10-33/v1#f3`

4.5 Summary

Snakemake has earned its place in mostly bioinformatic circles by ensuring the highly demanded reproducibility, robustness, but also scalability via integration with high-performance clusters.

At first, it might be challenging for the user to get his head around the rule definition grammar. Once the user gets hang of it, Snakemake becomes a powerful and effective orchestration tool with strong synergy with Jupyter notebooks. Adopting Snakemake might be a big step forward for analysts using plain Python or bash scripts as data processing pipelines.

Snakemake is aiming for the data science audience, rather than mainstream companies. This is reflected by the absence of central server and UI, rather focusing on effective task scheduling and individual task execution performance on computation clusters and grids.

5 Argo Workflows

Argo[1] is a Kubernetes oriented orchestration tool, standing out by defining workflows in a fully declarative way; extending Kubernetes Custom Resource Definition[2]. The design of the tool allows to run workflows in massive scale, roughly thousands of workflows per day.

Argo is developed under Cloud Native Computing Foundation[3] project. For anybody to use Argo effectively, one needs to be familiar with Docker and have a really good understanding of Kubernetes concepts.

5.1 Tasks

Argo tasks are represented as units called *templates*[4] and executed exclusively in Kubernetes pods. Templates are closely tied to container execution; running a container with arguments or having script wrapped up and executed inside of a container. In the latter case, script source code is stored in the YAML workflow definition file. Other template types allow performing cluster resource operations.

Container and script templates are defined as Kubernetes container specs with all related properties.

Tasks can share data using the combination of artifacts and Persistent Volume Claims that is also used for storing workflow results. Workflows and templates can be parametrized, also allowing to feed the output of one task into inputs to other tasks.

---

[1] `https://argoproj.github.io/`

[2] `https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/`

[3] `https://www.cncf.io/`

[4] `https://argoproj.github.io/argo-workflows/workflow-concepts/#template-types`

5.2 Workflows

Workflows are declared in yaml files as Workflow.spec[5], the Kubernetes specification of a workflow. The building blocks of a workflow are templates and an entrypoint; the template executed first.

Submission of workflows to the server is done via Argo CLI, UI, or REST API. Workflows are then stored as Kubernetes resources, in EtcD[6], by default. Alternatively, database storage can also be configured.

Workflows are represented by two types of *template invocators*[7]:

- *steps* executing task templates sequentially
- *DAG* executing tasks respecting the dependencies between them. Dependencies must be set manually.

Workflows also support caching to avoid redundant work and artifact producing/consuming mechanisms.

Steps in a workflow may or may not be executed based on conditions, like some exact value of a workflow parameter.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: hello-world
spec:
  entrypoint: whalesay
  templates:
  - name: whalesay
    container:
      image: docker/whalesay
      command: [cowsay]
      args: ["hello world"]
```

Listing 5.1: Argo workflow running the whalesay container

[5] https://argoproj.github.io/argo-workflows/fields/#workflowspec

[6] https://kubernetes.io/docs/tasks/administer-cluster/configure-upgrade-etcd/

[7] https://argoproj.github.io/argo-workflows/workflow-concepts/#template-invocators

5.3 Deployment

The official and Argo-supported installation and upgrade process is applying Kubernetes manifests located in the Argo GitHub repository. User can see what is being deployed and manifests can be customized if needed. The main functionality components are the workflow controller, which takes care of scheduling, and the Argo server, which runs the API server and UI.

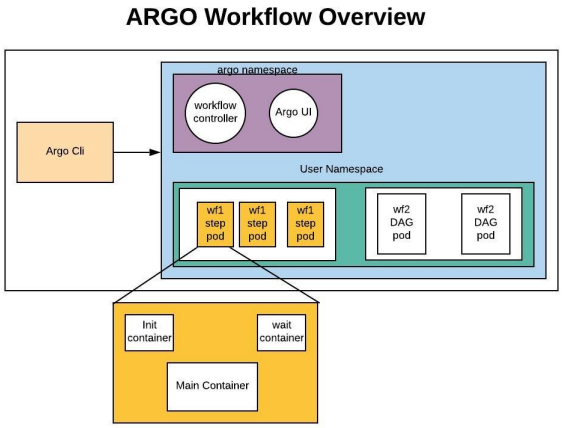


Figure 5.1: Argo architecture schema

Source: `https://argoproj.github.io/argo-workflows/architecture/`

5.4 API

Argo server comes with exposed REST API allowing user to communicate with the server via authenticated HTTP requests, which may come in handy to use with other automation tools.

There are also officially supported client libraries to wrap API calls for the user, written in Java, Python, and Golang.

5.5 Documentation and community

Documentation is in some places brief and some concepts could be described in more detail. The documentation is complemented by a really nice user interactive tutorial covering the most important concepts in the prepared Argo playground.

Argo GitHub repository contains many workflow examples and also a very informative tutorial by examples.

5.6 Summary

Argo Workflows is a great orchestrating tool for anyone who already invested in the Kubernetes environment and has workflow use-cases, where steps are more complex, containerized units in order to utilize Argo optimally. Workflow definitions might sometimes get tricky, especially when the user is not used to writing YAML files but in return, Argo's capabilities are quite unique.

6 Honorable mentions

This chapter is dedicated to workflow orchestration tools that earned their place in the workflow orchestration tool circles but didn't make it into the final comparison. The reason for this is that these tools were too far from ANALYZA use-case, or didn't provide any exceptional functionalities compared to selected, more popular tools.

6.1 Luigi

Developed by Spotify, Luigi[1] uses Python to orchestrate long-running batch jobs. Tasks are classes, where functions are used to define inputs, outputs, dependencies and action. Flows can be executed locally or using the central scheduler deployment. Flows are not defined explicitly, Luigi creates them based on task requirements.

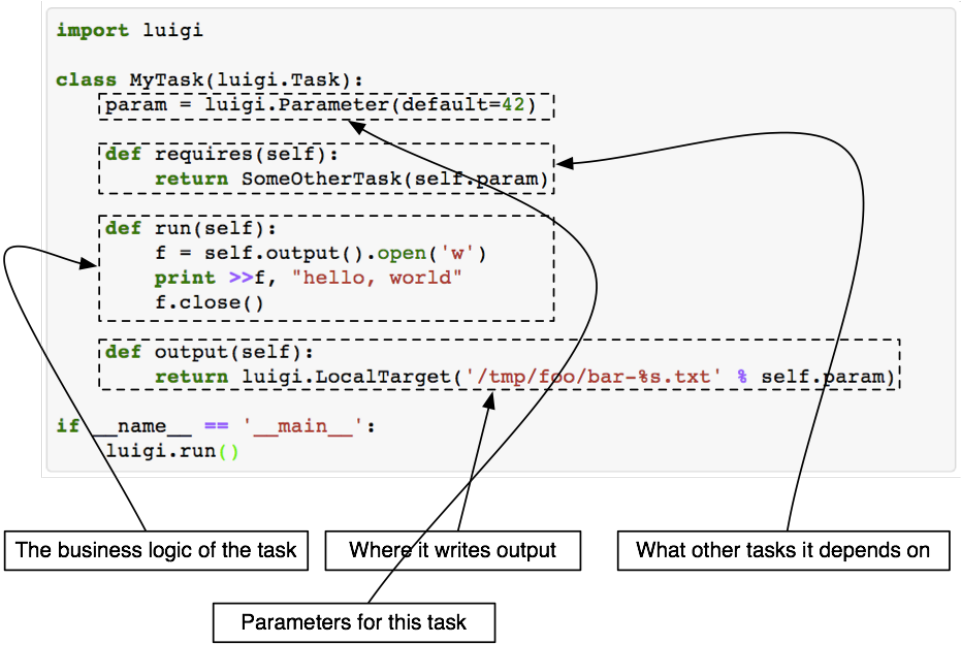


Figure 6.1: Luigi workflow example

Source: `https://luigi.readthedocs.io/en/stable/tasks.html`

[1] `https://luigi.readthedocs.io/en/stable/index.html`

6.2 Flyte

Flyte[2], a workflow orchestrator developed by Lyft, writing workflows in Python and defining workflows and tasks by using *@workflow* and *@task* decorators. Workflows are then executed in a Flyte cluster, deployed locally, or in Kubernetes.

```python
import typing
import pandas as pd
import numpy as np

from flytekit import task, workflow

@task
def generate_normal_df(n:int, mean: float, sigma: float) -> pd.DataFrame:
    return pd.DataFrame({"numbers": np.random.normal(mean, sigma,size=n)})

@task
def compute_stats(df: pd.DataFrame) -> typing.Tuple[float, float]:
    return float(df["numbers"].mean()), float(df["numbers"].std())

@workflow
def wf(n: int = 200, mean: float = 0.0, sigma: float = 1.0) -> typing.Tuple[float, float]:
    return compute_stats(df=generate_normal_df(n=n, mean=mean, sigma=sigma))
```

Figure 6.2: Flyte workflow example

Source: `https://docs.flyte.org/en/latest/getting_started/index.html`

[2] `https://flyte.org/`

6.3 Dagster

Dagster[3] is a Python orchestration tool combining computation steps, or *ops* as they call it, into *jobs* and further into *graphs* annotating Python functions by *@op*, *@graph* and *@job* operators. Dagster's workflow model is close to writing natural Python code, where nested function calls create implicit dependencies. The core, long-running Dagster deployment is composed of Daemon, Web Server, and Repositories. Workflows are run by *Executors*, e.g. Docker, Kubernetes, or Celery executors.

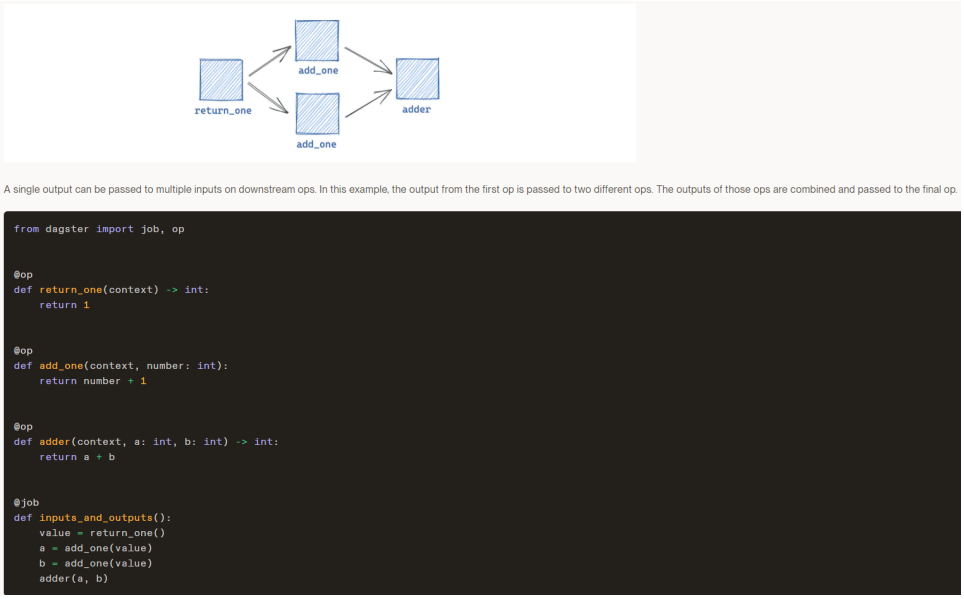


Figure 6.3: Dagster workflow example
Source: `https://docs.dagster.io/concepts/ops-jobs-graphs/jobs-graphs`

[3] `https://dagster.io/`

6.4 Nextflow

Nextflow[4], similarly to Snakemake, defines workflows using custom Domain Specific Language as an extension to the Groovy[5] programming language, which is a super-set of Java. Workflows are composed of *channels* and *processes*, where each process has inputs, outputs, and action. Workflows are executed locally or remotely via *Nextflow driver*, living in a Kubernetes cluster, which emits workflow jobs.

```
// Declare syntax version
nextflow.enable.dsl=2
// Script parameters
params.query = "/some/data/sample.fa"
params.db = "/some/path/pdb"

process blastSearch {
  input:
    path query
    path db
  output:
    path "top_hits.txt"

    """
    blastp -db $db -query $query -outfmt 6 > blast_result
    cat blast_result | head -n 10 | cut -f 2 > top_hits.txt
    """
}

process extractTopHits {
  input:
    path top_hits

  output:
    path "sequences.txt"

    """
    blastdbcmd -db $db -entry_batch $top_hits > sequences.txt
    """
}

workflow {
   def query_ch = Channel.fromPath(params.query)
   blastSearch(query_ch, params.db) | extractTopHits | view
}
```

Figure 6.4: Nexflow workflow example

Source: `https://www.nextflow.io/docs/latest/basic.html`

[4] `https://www.nextflow.io/`

[5] `https://groovy-lang.org/`

7 ANALYZA platform and new orchestrator service requirement analysis

Project ANALYZA[1] is a complex computational platform designed to handle the whole process of various large-scale data analyses, from storage, through computation to visualization. ANALYZA places emphasis on scalability, flexibility, and reliability.
The system architecture is composed of three core components:

- Data warehouse - where all the data to be analyzed are centralized on one place.
- Computation subsystem - set of data analyzing and transforming steps wrapped into data workflows and managed by the orchestrator service.
- Visualisation component - provides multiple views on results of data analysis and initiates analytical operation executions.

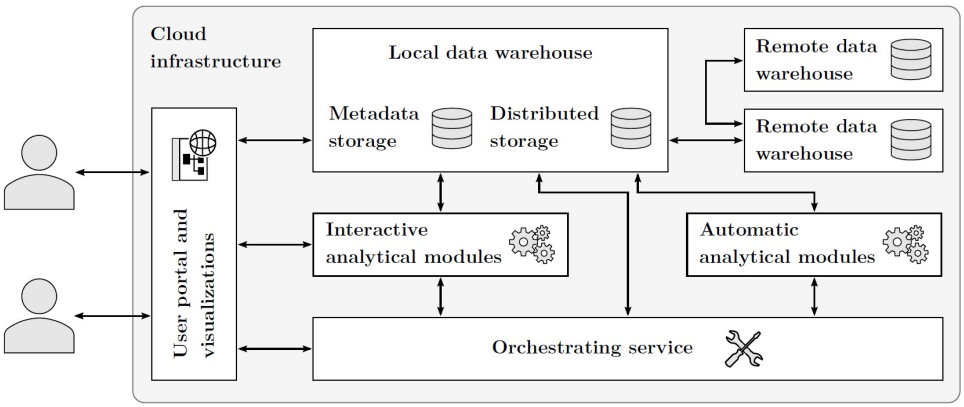


Figure 7.1: ANALYZA platform architecture

Source: [1]

This chapter is dedicated to the analysis of the Computation subsystem, especially the orchestrator service and its functionalities.

The computational subsystem is a Kubernetes service running on-demand analytical modules (tasks), composed by the orchestrator

service. Individual analytical modules are composed into analytical operations (workflows), transforming data for further data analysis. The orchestrator service runs module containers as Pods in the Kubernetes namespace in correct order with all needed inputs.

7.1 Orchestrator service

The orchestrator service was created as a part of the ANALYZA platform since at that time the field of data orchestration tools was lacking in tools fulfilling the project requirements. As time moved on, many new workflow orchestration tools emerged and there is now an option to swap from the current managed solution to a community-driven open-source service, lifting the burden of developing and maintaining the current orchestrator by the ANALYZA team.

7.1.1 Analytical modules

Tasks, represented by analytical modules, are Docker containers, further composed into analytical operations. Two types of analytical modules are available:

- Job – a short-running unit of work performing some kind of data processing action with a strictly bounded lifecycle. Jobs are executed as Kubernetes Jobs.
- Service – long-running module, whose execution is not blocking other modules and is running until the end of the entire analytical operation. Services are executed as Deployments and depending on the required internal/external ports, a Service of type ClusterIP/NodePort respectively.

The analytical module is defined by its type (Job or Service), Docker image, exposed container ports, parameters, and Kubernetes resource requirements. Property *IsBlocking* determines whether the module is running as a daemon[1].

[1] daemon – service running in the background

7.1.2 Analytical operations

Operation definitions are written in JSON format and stored in the definition database (PostgreSQL, MySQL, or SQLite). The operation contains a list of nodes and edges, representing dependencies between individual modules. Nodes are wrappers for analytical modules. Edges define dependencies between Nodes (modules) in a similar manner as in directed acyclic graphs.


```
{
   "modules": [
      {
         "name": "modul-test-1",
         "type": "service",
         "description": "testing service",
         "image": "test_1",
         "tag": "1.0",
         "isBlocking": true,
         "healthPort": 6060,
         "ports": [
            {
               "name": "port1",
               "number": 8080,
               "isPublic": true
            }
         ],
         "parameters": [
            {
               "name": "param1",
               "defaultValue": "value1",
               "description": "param1 description",
               "isStrict": true
            }
         ],
         "resources": {
            "cpus": 1,
            "ram": 250
         }
      }
   ]
}
```


Figure 7.2: Analytical operation definition example
Source: [1]

Orchestrator service comes with 2 CLI tools to operate on operations:

- Client CLI – provides communication with deployed orchestrator service with start/stop/get resource commands.
- Designer CLI – serves CRUD operations on definition database.

Additional to the client CLI, and the preferred way, to invoke operations is the REST API. The REST API exposes endpoints to manage all the orchestrator resources, like starting and stopping operations or listing running operations and modules.

Orchestrator service tries to optimize workflow execution by analyzing the operation to be run and if possible, reusing already running modules started by other operations.

7.1.3 Deployment

Orchestrator service deployment is simple containing:

- ConfigMap[2] – contains input parameters for the service like database credentials, data warehouse address or kubernetes config used further when executing operations (this is needed because orchestrator is managing Kubernetes resources).
- orchestrator deployment – specifies the orchestrator image and tag to be deployed. Serves the API and takes care of operation scheduling.
- orchestrator-svc service – orchestrator service is exposed as a single Kubernetes Service to expose REST API.

[2] https://kubernetes.io/docs/concepts/configuration/configmap/

7.2 Requirement analysis

Based on the orchestrator analysis, requirements for its successor can be formulated:

- Kubernetes support – individual analytical modules are encapsulated in virtualized containers for easier reusability and ensuring scalability and resource management.
- parametrized workflows – analysis steps may be experimental and is required to run workflows with different input parameters to achieve the most accurate outputs.
- dependencies between tasks - analysis may be a simple chain of steps, but also a complex computation tree with some steps strongly dependent on outputs of previous steps. The orchestrator service must be able to respect such dependencies.
- inter-task communication – dependencies between modules can be of many types, one such can be a step requiring normalized data stored somewhere in the data warehouse by the previous workflow step, another may be just using computed parameter from previous step. Orchestrator service should provide wide range of possibilities to ensure such communication.
- publicly available services – it is a common use case in ANALYZA project to run various types of analysis on large datasets where, as a result of the analysis, is a service endpoint exposed to the internet; outside of the Kubernetes cluster the analysis was run on. This use case introduces a non-trivial configuration that chosen workflow engine must fulfill.
- isolation to secure data – subject data to be analyzed may be of a sensitive character and it may be vulnerable to sending data to some cloud storage. It is required that the orchestrator service is able to run in isolated environments that may even be disconnected from the internet.
- security – not only processed data should remain secure, but also the service itself and also communication with the service

- open-source – the open-source character of the solution can bring many advantages, like the ability to view the source code, suggest improvements and bug fixes, or being free of charge.
- actively developed - when introducing any new technology into the existing project stack, it is appropriate to have a vision, that the chosen technology has a strong and stable developer base (especially in open-source projects) and will be supported in the future to prevent the need to replace the tool with new solution in near future.
- scalability – new orchestration solution should be able to scale well while maintaining high availability.
- API – analytical operations are in most use-cases invoked via REST API. new orchestration solution should keep this contract.

7.3 New orchestration tool choice

From the list of researched tools, most of them are to some extent meeting the requirements. One of them, Argo, has exceptional support for Docker and Kubernetes, while the rest of reviewed tools offer Docker/Kubernetes support just as one of the execution options; that means taking workflow source code and wrapping it in a container and running it in the Kubernetes cluster. On the other side, Argo is fully focusing on Kubernetes, and workflow definition is close to the current orchestrator. Taking these facts into account, Argo seems to be the best candidate for the new orchestrator role.

8 Argo Workflows fundamentals

Argo was introduced on a high level in chapter 5. This chapter explains Argo Workflows fundamentals in detail in order to explain Argo's application in the ANALYZA orchestrator service. From now on, Argo Workflows will be shortened to Argo, to avoid confusion with other projects from Argo portfolio[1].

8.1 Workflows

Workflows, and other workflow derivatives like WorkflowTemplate (see section 8.1.4), CronWorkflows[2], etc., are implemented as Kubernetes Custom Resources[3], thus their definitions follow Kubernetes object manifest format.

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow                  # new type of k8s spec
metadata:
  generateName: hello-world-    # name of the workflow
    spec
spec:
  entrypoint: whalesay          # invoke the whalesay
    template
  templates:
  - name: whalesay              # name of the template
    container:
      image: docker/whalesay
      command: [cowsay]
      args: ["hello␣world"]
```

Listing 8.1: Argo workflow example

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

[2] https://argoproj.github.io/argo-workflows/cron-workflows/#cron-workflows

[3] https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/

The *spec* part is where Argo-specific configuration takes place. The core structure of spec is defined as templates and an entrypoint.

Entrypoint is the root of the workflow, and there can be multiple entrypoints of one workflow definition (in the same manner as DAGs can have multiple roots, i.e. multiple vertices with in-degree equal to zero). Default entrypoint can then be overridden on submission. Every step of a workflow is a *template* and there are several types of templates, further divided into two groups:

1. Template Definitions – steps from which the workflow is composed:
   (a) container[4] (A.1) – the most common template type. It specifies the Docker container to be run as a workflow step. Container input arguments, image, and others are configured here.
   (b) script[5] (A.2) – extends the *container* template by injecting script source code to the container to be executed. This may come in handy for simple tasks, where it's not necessary to build a complete Docker image to run a few lines of code. The source code is stored directly in the definition YAML file, so for the sake of readability, the script should stay short and clear.
   (c) resource[6] (A.3) – manages Kubernetes objects directly. This is basically reduced *kubectl*[7] tool functionality built directly into the workflow. Supported operations are to *get*, *create*, *apply*, *delete*, *replace*, or *patch* cluster resources. Kubernetes manifest to be applied is specified directly as one of the template configuration fields.
   (d) suspend[8] (A.4) – pauses the workflow execution for a specified amount of time or indefinitely until user input is pro-

[4] https://argoproj.github.io/argo-workflows/fields/#container
[5] https://argoproj.github.io/argo-workflows/fields/#scripttemplate
[6] https://argoproj.github.io/argo-workflows/fields/#resourcetemplate
[7] https://kubernetes.io/docs/reference/kubectl/
[8] https://argoproj.github.io/argo-workflows/fields/#suspendtemplate

vided. The workflow can then be resumed via the CLI, API call, or UI.

2. Template Invocators – compose Template Definitions into workflows based on the complexity of workflow dependencies:

   (a) Steps(A.5) – execute a serie of tasks in the order they are defined. Steps can run sequentially or in parallel.

   (b) DAG(A.6) – the more complex invocator. Composes tasks into Directed Acyclic Graph, respecting all the dependencies.

Templates can be nested together, it is even possible to run a workflow as a part of another workflow.

8.1.1 Input and Outputs

In most cases, templates will have specified inputs and outputs (these can be chained together; one task can take the output of the previous task as its own input).

There are two types of arguments:

- Parameters(A.7) – string/integer values.
- Artifacts(A.8) – arbitrary resources like files or archives loaded from remote storage like S3, and then mounted on the specified location of a container.

When talking about input arguments, it's important to distinguish between workflow-level and template-level arguments.

- workflow arguments – usually the user input to the workflow invocation, defined at the root of the Argo definition spec; on the same level as *entrypoint*
- template arguments – defined for each template (step). These arguments can be referencing values of workflow-level arguments or other workflow variables.

Workflow level parameters, as one of the workflow argument options, offer an option to alter workflow behavior without changing the workflow source. Parameters have their default values hardcoded in the workflow source and can be overridden when submitting workflow. Parameters, also as other *workflow variables*[9] can be referenced anywhere in the YAML workflow source file, using special double curly bracket syntax, e.g. {{*workflow.parameters.size*}} when *size* is defined as workflow parameter.

8.1.2 Workflow management

Argo resources can be controlled in several ways:

- Argo CLI – requires access to the Kubernetes cluster Argo is running on.
- REST API – connects to Argo server exposed API. Most suitable for automation or integration with other tools.
- UI – together with workflow management, it gives the user insight into the state of workflows or view workflow visualization. Best suitable for workflow development and debugging.
- kubectl – since Argo resources are Kubernetes CRD, they can be implicitly managed by *kubectl*. However, this option is not recommended since is doesn't provide syntax validation the other options do.

8.1.3 Workflow execution

Naming

Each workflow step is executed in a separate *Pod* whose name follows specific format:

<workflow_name>-<workflow_id>-<pod_id>

[9] `https://argoproj.github.io/argo-workflows/variables/#workflow-variables`

Namespace

If not specified otherwise, workflows are executed in *the default* Kubernetes namespace. This setting can be overridden in the workflow definition file, specifying the *namespace* field under *metadata*, in the same manner as the rest of the Kubernetes resources. Another way to choose the target namespace is to set it as an invocation parameter in UI, or as a command line parameter when using Argo CLI.

Service account

By default, all workflow Pods run under the default service account in the namespace where workflow is executed. However, for security reasons, it's recommended to override this setting and create service accounts with respective roles for each type of workflow, depending on the scope and requirements. There is a defined minimum set of permissions required for all executing service accounts:

```yaml
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: executor
rules:
  - apiGroups:
      - argoproj.io
    resources:
      - workflowtaskresult
    verbs:
      - create
      - patch
```

Listing 8.2: Minimum Role persmissions required for Service Account to run workflows

A good idea on how to divide service accounts is by provided secrets. For example, the user may want to create a service account 's3-executor' with secrets needed to authenticate to S3[10]. This account would then be used for all workflows that need to connect to S3.

[10] `https://aws.amazon.com/s3/`

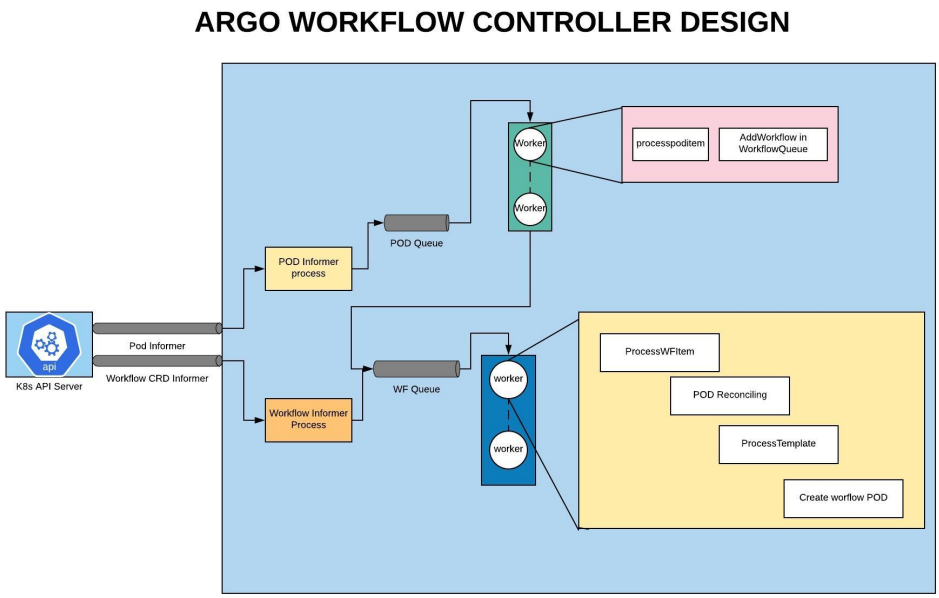


Figure 8.1: Argo workflow controller architecture
Source: `https://argoproj.github.io/argo-workflows/architecture/`

8.1.4 Workflow templates

Workflow templates are a vital part of the effective usage of Argo Workflows. By default, the workflow's lifecycle is consisting of submitting and running the workflow on demand. For better reusability, workflow templates were introduced and they allow the user to upload and store workflow definitions directly on the Argo server, removing the need to send the whole definition on each submit. It's even possible to reference workflow templates from other workflows.

Workflow templates will play important role in ANALYZA orchestrator requirement implementation.

It's crucial to understand the difference between template and WorkflowTemplate. WorkflowTemplate is the definition of the whole workflow stored in the cluster, but a template is a step of a workflow of which the whole workflow is composed. This naming collision is quite unfortunate and the user needs to get used to it.

8.2 UI

Argo UI serves most of the operations managed by the user, like submitting workflows, updating workflow definitions, or viewing past workflow runs. Workflow run visualization comes in handy when debugging new workflows or investigating failed runs. Logs from individual workflow steps, together with inputs/outputs are available to view too. There are also tabs managing integration with other Argo projects (when applicable).

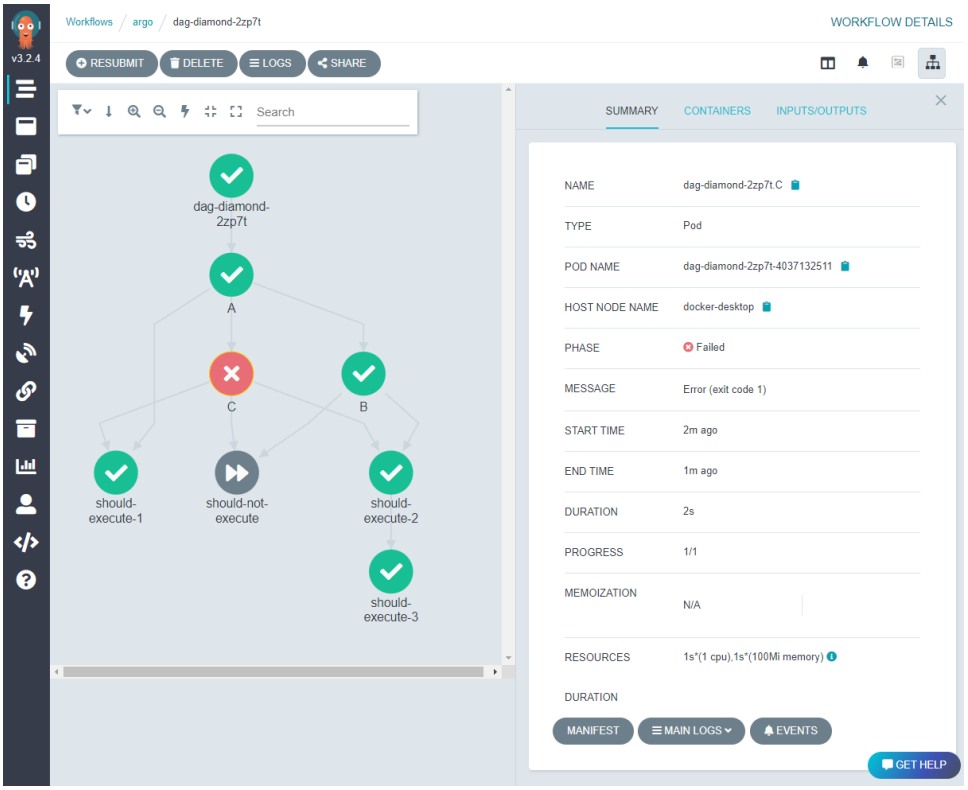


Figure 8.2: Argo UI displaying the state of a workflow run

8.3 Auth

Argo server configuration offers three authentication modes:

- server – no authentication required; anyone who can access UI or API, can operate with workflows
- client – in order to access the server, the user needs to provide his service account bearer token associated with the role he is supposed to have
- SSO – having configured the OIDC provider correctly (this includes having set Kubernetes secrets to the provider and referenced in the 'workflow-controller' ConfigMap together with the OIDC provider address), the user is able to authenticate using the OAuth2 protocol. RBAC is then managed using groups[11].

Argo server supports encrypted communication via TLS, enabled by the server passing a security flag to the server initialization command.

8.4 User management

When the Argo server is set to run in client auth mode, the user needs to provide an authentication token in order to make any kind of action. This token is the Kubernetes bearer token of a service account. This means each user needs to have a service account created. The role associated with the given service account determines what the user can do in the system. Since all Argo resources are implemented as Kubernetes resources, the usual Kubernetes access restriction[12] applies. User can have restricted access just to some kinds of Argo resources, like Workflows or WorkflowTemplates, or even allowed resources can be specified by concrete names.

Users can also be restricted to only reduced set of actions, for example, user can be restricted just to view workflows, not create new ones.

[11] `https://argoproj.github.io/argo-workflows/argo-server-sso/#sso-rbac`

[12] `https://kubernetes.io/docs/reference/access-authn-authz/authentication/`

```yaml
# The restriction could be tightened further by using `resourceNames`
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: submit-workflow-template
rules:
  - apiGroups:
      - argoproj.io
    resources:
      - workfloweventbindings
    verbs:
      - list
  - apiGroups:
      - argoproj.io
    resources:
      - workflowtemplates
    verbs:
      - get
  - apiGroups:
      - argoproj.io
    resources:
      - workflows
    verbs:
      - create
```

Listing 8.3: Role definition example with enough permissions to submit a workflow template.

8.5 Argo API

Argo API can be controlled in two ways:

- REST API – Argo server exposes rich RESTful API allowing the user to fully control Argo workflows using HTTP requests. When the Argo server is run in *client* mode, requests need to have filled the Authorization header with the respective bearer token.
- client libraries – Argo is also controllable using client libraries, officially supported by the Argo team. Supported languages are Go, Java and Python. When managing Argo via code, it's arguably more comfortable to use the client library instead of REST API, since it makes the request construction, response parsing, and authentication much easier. Client libraries are basically a wrapper around GRPC communication between the client and Argo server. Argo CLI is under the hood using the Go client library.

8.6 Argo deployment

Argo is deployed to the cluster by applying a set of Kubernetes manifests provided by the Argo team as a part of each release. Manifests can be found in the Argo GitHub repository in the 'Releases' section. These manifests define service accounts and related roles required to run the server, the workflow controller, and some custom resource definitions.

Server configurations are stored in ConfigMap passed to the workflow controller Deployment. Available configuration options[13] need to be set in deployment manifests before applying to the cluster.

[13] `https://argoproj.github.io/argo-workflows/workflow-controller-configmap.yaml`

The deployment consists of two main services, each is required to run under a service account with sufficient permission role:

- argo-server – hosting UI and API
- workflow-controller – scheduling workflows and communicating with Kubernetes API

Currently, there are three variants of deployment, depending on the scope Argo operates on:

- cluster – Argo manages workflows on all cluster namespaces
- namespaced – Argo manages workflows only in the namespace Argo is deployed in
- managed namespace – Argo manages workflows in the namespace Argo is deployed in and those specified in workflow-controller ConfigMap

It's common practice to have multiple Argo deployments across Kubernetes namespaces, for performance, security, or use-case distribution.

The bigger the scope Argo operates on, the more permissions must be elevated for workflow-controller.

For the UI to be available publicly, the *argo-server* Service needs to be patched to be of type NodePort to use the node's address or *LoadBalancer* to get an IP address from the cluster load balancer provider. This can be patched on runtime, or even better to specify this in deployment manifests before Argo is installed.

8.6.1 Scaling

Argo can be scaled up in several ways:

- increase *QPS* value – the bigger QueriesPerSecond, the more requests can Argo send to the Kubernetes API.
- increase *burst* value – allows to temporarily exceed QPS value.
- increase number of workers – event workers are processing scheduled workflows/pods. The more workers, the faster is the scheduler queue being processed.

- increase CPU/memory for controller – increasing the number of event workers may result in higher CPU/Memory demands.
- run multiple workflow controllers – when running a single workflow controller replica, in case of failure, service disruption may occur until a new Pod is created. When running multiple replicas, when the main replica fails, it can be replaced by backup replicas.
- run multiple Argo servers – in case of really high API/UI request volume, some requests may get lost. To prevent this, run multiple Argo server replicas.
- utilizing Pod disruption budget[14] on workflow controller.

[14] `https://kubernetes.io/docs/concepts/workloads/pods/disruptions/#pod-disruption-budgets`

8.7 Secrets

Argo works with secrets via standard Kubernetes secret mechanism[15]. Each workflow runs under the specified service account (or default, if not set explicitly) and each service account has secrets associated with it. The workflow template can then reference these secrets in the workflow definition and secrets are then mounted as environment variables to the container.

```
1  ...
2    templates:
3    - name: print-secret
4      container:
5        image: alpine:3.7
6        command: [sh, -c]
7        args: ['echo "secret from env: PASSWORD";']
8        env:
9          - name: PASSWORD
10           valueFrom:
11             secretKeyRef:
12               name: my-secret
13               key: mypassword
```

Listing 8.4: Template snippet, demonstrating secret referencing from template definition

[15] https://kubernetes.io/docs/concepts/configuration/secret/

9 Requirements implementation in Argo

Most of the requirements are fulfilled trivially just by using Argo's built-in functionalities described in chapter 8. However, some are more complex and need some more smart compositions of Argo features.

9.1 Public services

The most complex requirement to implement is to expose the service endpoint as a part of a workflow step to the internet. This requires correct Kubernetes configuration together with keeping this configuration simple and easy to reuse. Before the demonstration of how this can be achieved, some Kubernetes theory needs to be reminded.

9.1.1 Kubernetes resources revised

Pod

The most basic Kubernetes unit is a Pod, which is in fact an encapsulation of one or more containers, mostly of Docker runtime. When multiple containers are running in a Pod, they share a common set of Linux namespaces, groups, and other facets of isolation. A lifecycle of a pod is tightly tied to the state of the containers running within.

Nodes

A node is a worker machine that runs Kubernetes workloads. It can be a physical (bare metal) machine or a virtual machine (VM)

Workloads

Although it is possible to define manifests for Pods, it's more common to manage Pods using Workloads like Deployments, StatefulSets, or Jobs. These workloads manage Pods to meet the desired state defined in the Workload definition. Such state can consist of a number of available Pods running containers in the latest versions, managing stateful and stateless applications, or running Jobs.

Service

Pods are logically grouped in namespaces[1] are part of an internal network using internal DNS. From the Kubernetes documentation:

"In Kubernetes, a Service is an abstraction that defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). The set of Pods targeted by a Service is usually determined by a selector."

9.1.2 Create service in Argo

Now, with the knowledge of how networking in a Kubernetes cluster works and what needs to be done, the goal is to find a way how to create a proper Kubernetes service as a part of a workflow. Fortunately, Argo comes with an easy way of managing Kubernetes resources, a *resource template*. An example of such template is shown in listing 9.1.

[1] `https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/`

```yaml
- name: pi-tmpl
  resource:
    action: create
    manifest: |
      apiVersion: v1
      kind: Pod
      metadata:
        name: nginx
        labels:
          app.kubernetes.io/name: proxy
      spec:
        containers:
        - name: nginx
          image: nginx:11.14.2
          ports:
            - containerPort: 80
              name: http-web-svc
      ---
      apiVersion: v1
      kind: Service
      metadata:
        name: nginx-service
      spec:
        type: NodePort
        selector:
          app.kubernetes.io/name: proxy
        ports:
        - name: name-of-service-port
          protocol: TCP
          port: 80
          targetPort: http-web-svc
          nodePort: 30007
```

Listing 9.1: Kubernetes Service creation as a workflow step

Specifying service in each workflow like shown above would work, but it makes the template less readable and brings complexity to team members writing workflows, but not that familiar with the rest of Kubernetes mechanisms. Fortunately, Argo's *WorkflowTemplate* feature can easily solve this problem.

Template creating new service can be turned into parametrized WorkflowTemplate. Usable parameters to this template might be opened ports, container to run in the deployment, or resource labels.

Create service template breakdown

A service in Kubernetes is composed of Deployment (or other workload) and Service.

The parametrized template example to create a Service is shown in listing 9.2

```
1       - name: external-svc
2         inputs:
3           parameters:
4             - name: app
5             - name: port
6         resource:
7           action: create
8           setOwnerReference: true
9           manifest: |
10            apiVersion: v1
11            kind: Service
12            metadata:
13              name: service-external-{{inputs.parameters.
                    app}}
14            spec:
15              type: NodePort
16              selector:
17                name: {{inputs.parameters.app}}
18              ports:
19                - port: {{inputs.parameters.port}}
```

Listing 9.2: A reusable, parametrized template (step) that creates new Service of type NodePort

The template definition is pretty straightforward, parameters are referenced using double curly braces syntax as has been shown in 8.1.1.

Kubernetes Service needs Pods to handle service calls and this can be managed e.g. by Deployment. The Deployment creation template is defined in a similar manner as the Service template. The *selector* field in the Service definition needs to match the *name* label in the *metadata* section of the Deployment manifest. An example of such template is shown in listing 9.3.

```yaml
- name: deployment
  inputs:
    parameters:
      - name: app
      - name: image
      - name: tag
      - name: env
      - name: port
      - name: args
  resource:
    action: create
    setOwnerReference: true
    manifest: |
      apiVersion: apps/v1
      kind: Deployment
      metadata:
        name: {{inputs.parameters.app}}
      spec:
        selector:
          matchLabels:
            name: {{inputs.parameters.app}}
        template:
          metadata:
            labels:
              name: {{inputs.parameters.app}}
          spec:
            containers:
              - name: {{inputs.parameters.app}}
                image: {{inputs.parameters.image}}:{{inputs.parameters.tag}}
                ports:
                  - containerPort: {{inputs.parameters.port}}
                args: ["{{inputs.parameters.args}}"]
                envFrom:
                - configMapRef:
                    name: {{inputs.parameters.env}}
```

Listing 9.3: A reusable template that creates new Deployment

Using two templates defined above is enough to create a Service available from public internet. However, notice the Service only exposes a single port, which may not fit all the use cases for Services. An easy fix for this problem is to use another template, which applies patch[2] to the existing Service opening even more ports, if possible.

[2] https://kubernetes.io/docs/tasks/manage-kubernetes-objects/update-api-object-kubectl-patch/

```
1       - name: patch
2         inputs:
3           parameters:
4             - name: names
5             - name: app
6             - name: portValue
7             - name: portName
8         resource:
9           action: patch
10          manifest: |
11            apiVersion: v1
12            kind: Service
13            metadata:
14              name: service-external-{{inputs.parameters.
                    names}}
15            spec:
16              type: NodePort
17              ports:
18                - port: {{inputs.parameters.portValue}}
19                  targetPort: {{inputs.parameters.
                        portValue}}
20                  name: {{inputs.parameters.portName}}
21              selector:
22                name: {{inputs.parameters.app}}
```

Listing 9.4: A reusable template that patches existing service

The number of ports can be variable so some kind of iteration through the parameter list may come in handy and Argo allows that using the *WithParam* clause.

Invoking the *patch* template defined above with variable number of parameters is shown on the code shown in listing 9.5:

```
steps:
...
    - - name: patch
        template: patch
        arguments:
          parameters:
            - name: app
              value: '{{inputs.parameters.app}}'
            - name: portValue
              value: '{{item.value}}'
            - name: portName
              value: '{{item.name}}'
        withParam: '{{inputs.parameters.ports}}'
```

Listing 9.5: An invocation of reusable template that patches existing service

Notice the usage of *WithParam* field. It iterates through given input parameter and individual configurations can be then referenced as *items*. Such invocation of complex input parameters can be defined as follows:

```
...
arguments:
  parameters:
    - name: ports
      value: |
        [
          { "name": "kafka", "value": 9092 },
          { "name": "https", "value": 443 }
        ]
```

Listing 9.6: Example of passing structured items as parameters to template

Parameters then behave as key-value pairs assigned to each *item*.

A complete example of WorkflowTemplate creating Service can be found in the attachment. The beauty of these templated solutions is in a variety of modifications. Such modifications could be adjustments

to naming, labeling, or changing the garbage collection strategies, depending on the conventions established in the Kubernetes cluster where workflow executions will be held.

9.1.3 Garbage collection

When the Service is up and running, it may run for some period of time that may not be known prior to definition/execution. Such situation may be solved by a few different approaches. One may be to leave the service running and then run some periodic jobs in Kubernetes cluster that would clear Services that are no longer used. This would require defining rules, on when to delete the Service.

Another approach may be taken utilizing Argo functionalities:

- utilizing workflow property *onExit*, which specifies template executed at the very end of the workflow (similar to the *defer* mechanism used in Go[3]). Each workflow then could define a cleanup template of type *resource* that would remove all resources created during the workflow execution. The catch here is to keep the workflow running for some time, so the cleanup template is not executed too early. This can be achieved by setting template of type to *suspend* as the last template executed.
- utilizing *setOwnerReference* property of *resource* template. Setting this value to *true*, Argo instruments Kubernetes garbage collection to destroy created resource, when the whole workflow ends and is being garbage collected.

9.2 Custom API

Although Argo provides REST API by default with installation, it's convenient to create another layer of API between Argo and the user. The main reason for this is that requests and responses of Argo are quite verbose and might be hard to navigate in. An API with a reduced set of requests/responses was created to better fit the needs of the ANALYZA project. This means the set of available requests has been cut down to basic operations only with mandatory information

[3] `https://go.dev/tour/flowcontrol/12`

and actions. These actions include mainly workflow management like creating, stopping, or viewing workflows. The API contract is shown in the table 9.1.

Apart from reduction of complexity, the additional API layer also creates an abstraction of the orchestration tool currently used. This means, that in case of a decision to move into another workflow orchestrator tool happens, the contract between API and consumer will be able to stay the same. This is ensured because API hides the implementation details of the used orchestration tool and its contract between API and consumer is reduced to the minimum set of functions with parameters always present in each orchestration tool.

Method	URL	Description
Get	/v2/workflow/{name}	Gets workflow status
Get	/v2/workflow/running	Gets all running workflows
Get	/v2/workflow/{definitions}	Gets all runnable workflow definitions
Get	/v2/workflow/definition/{name}	Gets single runnable workflow definition
Post	/v2/workflow/start	Invokes workflow
Delete	/v2/workflow/{name}	Stops workflow execution

Table 9.1: Custom Argo API Contract

9.2.1 Changes to the API

Some API contract changes were required when compared to the original ANALYZA orchestrator API. The reason for changes is that the old API[1] was designed to reflect the old orchestrator implementation and its details (e.g. concept of nodes, modules,..). Even though the API has changed, its functionality remained mostly unchanged. The only breaking change is the loss of control of workflows on the module/step level. However, it's possible to adjust the custom API to parse this information from Argo, if the future shows that this functionality is really required.

9.2.2 Implementation

API is implemented in Go and the project structure follows standard Go project layout[4]. The connection between API and Argo server is ensured using Argo's Go client library.

Code is compiled as a part of a Docker multi-stage build, where source code is compiled in an intermediate image, and result binaries are then copied to the final image. The base of the result image is *gcr.io/distroless/base-debian11*; which is an image with almost no binaries in it. The used base image size has roughly *2 MB* in size. Apart from size, another advantage is reduced attack surface in case of a hack attempt.

[4] `https://github.com/golang-standards/project-layout`

10 Argo features that might be used in future

This chapter describes Argo features that are not necessary for writing workflows but might be beneficial.

10.1 Workflow definition synchronization

Over time, workflow definitions may change and new ones can be created. As has already been shown, it's convenient to store commonly used workflows as WorkflowTemplates. However, it's not user-friendly to go through the manual submission process over and over again, so a little bit of automation can step in. Since workflow definitions are text files, it's convenient to store them in Git. This introduces better control over workflow definitions, versioning, and history. Synchronization between Git and Argo needs to be established using some kind of pipeline, e.g. Github Actions[1] when using Github.
An example of such pipeline using Github Actions may look like this:

0. Prerequisite is to have Argo API address and Argo access token stored in Github secrets.
1. User pushes new workflow template to a Git branch.
2. An automated on-push action is triggered. A Docker container from *argoproj/argocli* image is created. In the container, *argo template lint* command is executed, performing syntax check of committed workflow definition, to prevent merging invalid files into the master branch.
3. After the syntax lint check and pull request review from peer developers, the branch is merged to master.

[1] `https://docs.github.com/en/actions`

4. Another automated action is triggered, again creating the *argoproj/argocli* container with Argo credentials pulled. Currently, Argo doesn't support updating a WorkflowTemplate using Argo CLI, but there are workarounds:
   - Using Argo CLI: Deleting WorkflowTemplate and then recreating with the same name.
   - Using *kubectl*: Since WorkflowTemplate is just a Kubernetes CustomResourceDefinition, *kubectl apply* could be used to update the template. Beware, this option doesn't include syntax validation

10.2 Usage of volume claims

Currently, ANALYZA workflows are designed to share data through the data warehouse. This may create unnecessary traffic to the warehouse just to pass data between steps of a workflow. An optimization of this may be to use Kubernetes persistent volume claims as temporary data storage in the context of a workflow run.

10.3 Workflow notifications

User may want to be notified about workflow events as workflow failed, or workflow succeeded. This can be achieved by editing the default workflow spec[2], and running workflow using *onExit* property. This workflow can be sending results of the workflow, or doing any kind of cleanup.

Alternatively, user can collect Kubernetes events about workflows emitted by Argo. However, Kubernetes events live in the cluster for only limited amount of time, so some kind of collector needs to be set up to collect these events and send them to some persistent storage. An example of such a tool can be *Kubewatch*[3], which is unfortunately no longer maintained by original authors but is externally maintained by the community.

---

[2] `https://argoproj.github.io/argo-workflows/default-workflow-specs/#default-workflow-spec`

[3] `https://github.com/vmware-archive/kubewatch`

10.4 Prometheus metrics

When executing workflows, Argo can emit metrics[4] collectable by *Prometheus*[5], quite popular tool used to watch Kubernetes metrics. Such metrics could be average workflow execution time, failure or success rate. On top of Prometheus, user can build a Grafana[6] dashboards to visualize these metrics.

[4] https://argoproj.github.io/argo-workflows/metrics/#prometheus-metrics

[5] https://prometheus.io/

[6] https://grafana.com/

11 Demonstration and evaluation of results

The following chapter demonstrates Argo's capabilities on real use-case, describes testing environment and demonstration scenario. Individual scenario steps are linked to Appendix, where lie actual template definitions used in the demonstration. The final section evaluates observed results.

11.1 Demonstration environment

All the tools evaluation was executed on desktop PC with Intel Core i5-9600K CPU and 32 GB RAM running Windows 11 and Ubuntu WSL[1].

Two separate options of Kubernetes deployment were used in the testing process:

- *kind*[2] (**k**ubernetes **in d**ocker) tool
- Docker Desktop's built-in Kubernetes feature[3]

Both tools, when enabled, automatically create a *kubeconfig*[4] and context[5] configuration, that is used by *kubectl* tool to communicate with the cluster.

To communicate with the Argo server via either UI or REST API, the port-forwarding[6] on Argo deployment was applied. Port-forwarding is quite common practice when developing Kubernetes application; it brings the comfort of accessing Pods from the local machine without the need to set up the ingress[7]. Port-forwarding can also be applied to access all Pods, even those not exposed as a Service.

---

[1] `https://ubuntu.com/wsl`

[2] `https://kind.sigs.k8s.io/`

[3] `https://docs.docker.com/desktop/kubernetes/`

[4] `https://kubernetes.io/docs/tasks/access-application-cluster/access-cluster/`

[5] `https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/`

[6] `https://kubernetes.io/docs/tasks/access-application-cluster/port-forward-access-application-cluster/`

[7] `https://kubernetes.io/docs/concepts/services-networking/ingress/`

11.2 Demonstration scenario

Argo capabilities were tested on scenario based on real data analysis use-case. Workflow consists of following steps:

1. A.9.1 – Spin up MySQL database using Deployment, Service of type NodePort (to allow access outside of cluster), and Service of type ClusterIP (to allow access from inside of cluster). All these steps are encapsulated into a reusable WorkflowTemplate. The Deployment loads environment variables from ConfigMap supplied to the service creation WorkflowTemplate. This step demonstrated the usage of *resource* templates executed in parallel, parameter passing, and usage of WorkflowTemplates.

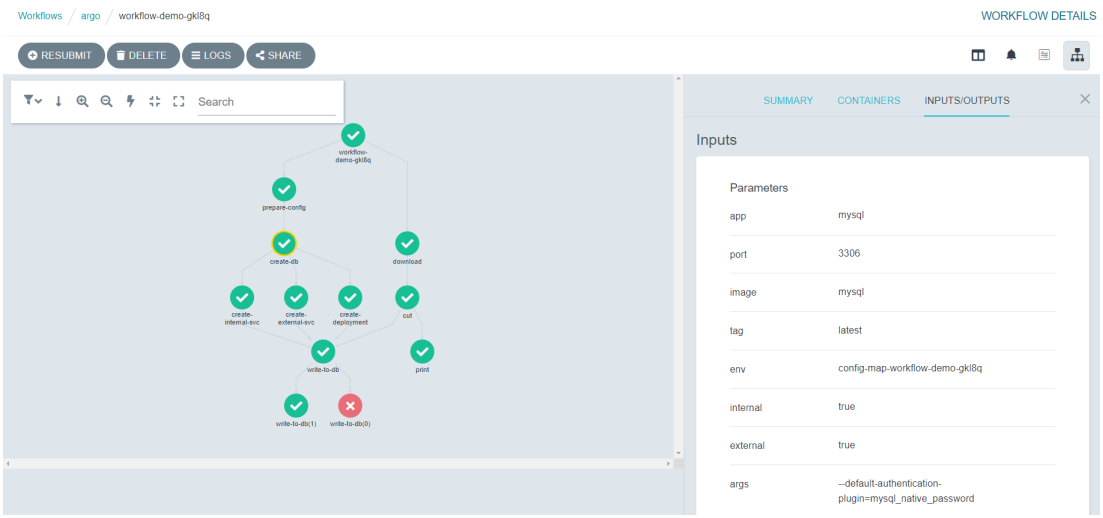


Figure 11.1: MySQL demo step

2. A.9.2 – Download CSV dataset using *curl* command executed in *container* template and save it to persistent volume shared between steps. This step demonstrated container execution with arguments and usage of persistent volume claims.

3. A.9.3 – Load CSV dataset from a persistent volume, cut the content to first 5 lines of the dataset and write back to persistent volume. This step demonstrated the usage of artifacts, where CSV processing Go code was downloaded as git artifact from a public repository and executed by *Golang* Docker image.

4. A.9.4 – After the database is up and the CSV dataset is preprocessed (this is enforced by using DAG template and dependencies), the *source* template is invoked, running Python script connecting to deployed MySQL and loading CSV dataset from persistent volume. Then it fills the database with the dataset. The database schema is created dynamically from the dataset structure. If the script attempts to write to the database before it's fully initialized (remember that Service/Deployment creation success doesn't imply it's fully initialized. However, it's possible to create guards to ensure this property), the execution fails and retry policy steps in to try again. This step demonstrated the usage of inline script used in the workflow definition file, retry policies, and dependency enforcement.

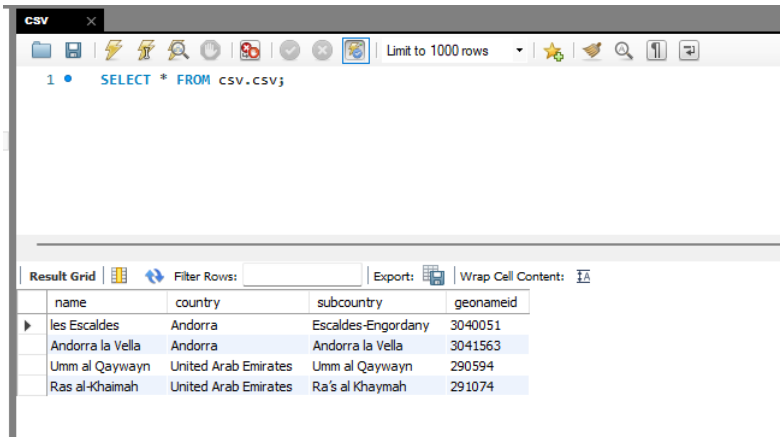


Figure 11.2: MySQL DB demo step

5. A.9.5 – In parallel with databse writing step, there's a step that reads pre-processed dataset from persistent volume and writes it into standard output.

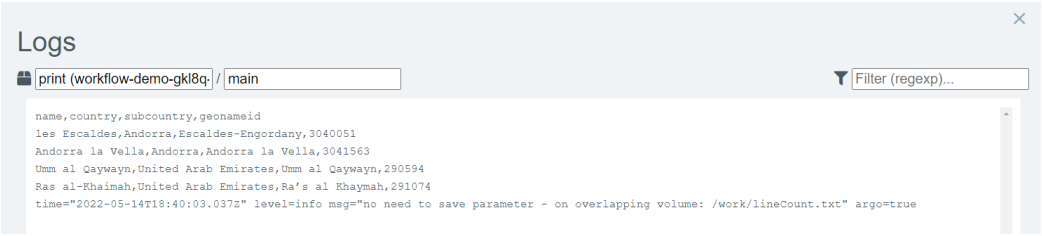


Figure 11.3: Print step demo logs

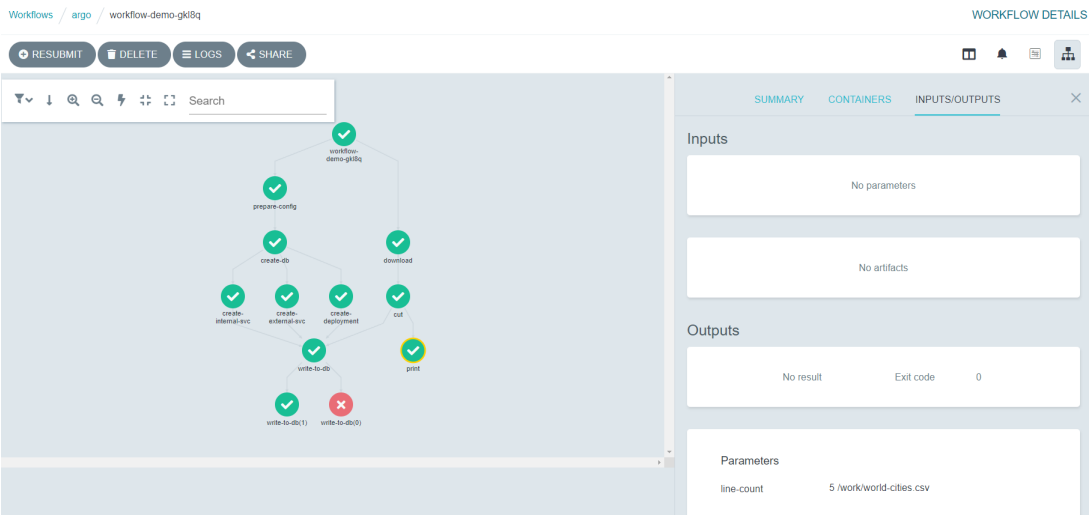


Figure 11.4: Print demo step outputs

The result of the demonstration workflow is MySQL database accessible from outside of Kubernetes cluster (ensured by Service of type NodePort). User with access to the network where Kubernetes Node takes place, could then query the database. Since all resources (Services and Deployments) have configured *setOwnerReference* to *true*, all of these will be destroyed when garbage collection will destroy workflow pods. This behavior can be adjusted by adding the final workflow step of type *suspend* that would keep the workflow running until user input to resume workflow is provided.

Complete test scenario definitions can be found in the thesis attachment.

11.3 Evaluation

After the Argo Workflows' capabilities theoretical review and practical demonstration, it's appropriate to evaluate Argo Workflows as full-featured orchestration engine suitable to succeed the ANALYZA's orchestration tool. Argo proved its exceptional Kubernetes support, task dependencies, parameters and abilities to manage Kubernetes resources to fulfill all ANALYZA platform requirements.

There is one function that Argo didn't manage to fulfill, and that is reusing of running modules the way the ANALYZA orchestrator do. However, this is by Argo's design, where workflow steps are isolated units. However, Argo is able to mimic this behavior with some advanced usages of workflow properties, but there's no one-line solution for this case as it is in ANALYZA orchestrator.

12 Conclusion

The goal of this thesis was to review the current market of workflow orchestration tools, mostly those operating on Kubernetes, then pick one of them to succeed the in-house built workflow orchestration solution in the ANALYZA project.

The research part of the thesis introduced four workflow orchestration tools, described workflow definition syntax, deployment, and execution model, and discussed the documentation and community around the tool.

The next part analyzed the current ANALYZA orchestration tool, the architecture, and workflow definition style. Based on this analysis, requirements for the orchestration tool successor were formalized. Based on these requirements, Argo Workflows was selected as the tool of choice.

The final part demonstrated Argo Workflow's capabilities from the user point of view, from workflow definition syntax, through deployment and execution to implementation of more complex requirements defined in the requirements. Each of the most important workflow properties was demonstrated in examples. Some advanced workflow properties Argo offers are suggested to use in ANALYZA workflows that may improve performance and usability together with integrations with 3 rd party tools.

Further improvements could be applied to the custom API, depending on the feedback from production experiences. There are a lot of features Argo API provides and are not yet included in the custom API on purpose, to keep things simple and add functionality when it's really needed. Even though the API complies with the required contract, the request and response payload can be adjusted based on the production needs. Since the API is running in the Kubernetes cluster, it can help supervise Kubernetes resources created by workflows. For example, it could have the ability to list all publicly exposed services, or close the no longer used ones.

Another area of improvements is to build on top of service creation WorkflowTemplate used in the demonstration scenario, and extend it to be more flexible in terms of service and deployment properties, or labelling strategy.

A Workflow Examples

A.1 Container

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: container-demo-
spec:
  entrypoint: whalesay
  templates:
    - name: whalesay
      container:
        image: docker/whalesay:latest
        command: [cowsay]
        args: ["hello world"]
```

Listing A.1: Argo workflow example running the whalesay container

A.2 Script

```
1  apiVersion: argoproj.io/v1alpha1
2  kind: Workflow
3  metadata:
4    generateName: script-demo-
5  spec:
6    entrypoint: generate
7    templates:
8      - name: generate
9        script:
10         image: python:alpine3.6
11         command: [python]
12         source: |
13           import random
14           i = random.randint(1, 100)
15           print(i)
```

Listing A.2: Argo workflow example running Python script

A.3 Resource

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: resource-demo-
spec:
  entrypoint: prepare-config
  templates:
    - name: prepare-config
      resource:
        action: create
        setOwnerReference: true
        manifest: |
          apiVersion: v1
          kind: ConfigMap
          metadata:
            name: config-map-mysql
          data:
            MYSQL_ROOT_PASSWORD: root
```

Listing A.3: Workflow example running the executing resource manifest

A.4 Suspend

```yaml
kind: Workflow
metadata:
  generateName: suspend-demo-
spec:
  entrypoint: suspend
  templates:
  - name: suspend
    steps:
    - - name: suspend-temporary
        template: suspend-20
    - - name: suspend-forever
        template: suspend-forever

  - name: suspend-forever
    suspend: {}

  - name: suspend-20
    suspend:
      duration: "20"
```

Listing A.4: Workflow example running the suspend template

A.5 Steps

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: steps-demo-
spec:
  entrypoint: generate
  templates:
    - name: generate
      script:
        image: python:alpine3.6
        command: [python]
        source: |
          import random
          i = random.randint(1, 100)
          print(i)
```

Listing A.5: Workflow example running steps template

A.6 DAG

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: dag-demo-
spec:
  entrypoint: diamond
  templates:
    - name: diamond
      dag:
        tasks:
          - name: A
            template: echo
            arguments:
              parameters:
                - name: message
                  value: "I'm running first"
          - name: B
            depends: "A"
            template: echo
            arguments:
              parameters:
                - name: message
                  value: "I'm running after A"
          - name: C
            depends: "A"
            template: echo
            arguments:
              parameters:
                - name: message
                  value: "I'm running after A"
          - name: D
            depends: "B && C"
            template: echo
            arguments:
              parameters:
                - name: message
                  value: "I'm running after B and C"

    - name: echo
      inputs:
        parameters:
          - name: message
      container:
        image: alpine:3.7
        command: [echo, "{{inputs.parameters.message}}"]
```

Listing A.6: Workflow example running DAG template

A.7 Parameters

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: parameters-demo-
spec:
  serviceAccountName: argo
  entrypoint: hi
  arguments:
    parameters:
      - name: message # workflow-level parameter configurable at submission
        value: "workflow input message"
  templates:
    - name: hi
      inputs:
        parameters:
          - name: message # referencing workflow-level parameter
      steps:
        - - name: hello1
            template: whalesayToFile
            arguments:
              parameters:
                - name: message
                  value: "{{inputs.parameters.message}}"  # referencing 'message' parameter
                         value of template 'hi'
        - - name: hello2
            template: whalesay
            arguments:
              parameters:
              - name: message
                value: "{{steps.hello1.outputs.parameters.hello}}" # referencing 'hello' output
                       parameter value of template 'hello1'

    - name: whalesay
      inputs:
        parameters:
          - name: message
      container:
        image: alpine:latest
        command: [echo]
        args:
          - "{{inputs.parameters.message}}"
    - name: whalesayToFile
      inputs:
        parameters:
          - name: message
      container:
        image: docker/whalesay:latest
        command: [sh, -c]
        args: ["cowsay {{inputs.parameters.message}} > /tmp/hello_world.txt"]
      outputs:
        parameters:
          - name: hello
            valueFrom:
              path: /tmp/hello_world.txt
```

Listing A.7: Workflow example running parameters passing demo

A.8 Artifacts

```yaml
#this demo requires configured artifact storage
apiVersion: argoproj.io/v1alpha1
kind: Workflow
metadata:
  generateName: artifact-demo-
spec:
  entrypoint: artifact-example
  templates:
  - name: artifact-example
    steps:
    - - name: generate-artifact
        template: whalesay
    - - name: consume-artifact
        template: print-message
        arguments:
          artifacts:
          - name: message
            from: "{{steps.generate-artifact.outputs.artifacts.hello-art}}"

  - name: whalesay
    container:
      image: docker/whalesay:latest
      command: [sh, -c]
      args: ["sleep 1; cowsay hello world | tee /tmp/hello_world.txt"]
    outputs:
      artifacts:
      - name: hello-art
        path: /tmp/hello_world.txt

  - name: print-message
    inputs:
      artifacts:
      - name: message
        path: /tmp/message
    container:
      image: alpine:latest
      command: [sh, -c]
      args: ["cat /tmp/message"]
```

Listing A.8: Workflow example running artifact producer/consumer demo

A.9 Test scenario

A.9.1 Service creation

```yaml
apiVersion: argoproj.io/v1alpha1
kind: WorkflowTemplate
metadata:
  name: create-service
  namespace: argo
spec:
  entrypoint: main
  templates:
    - name: main
      inputs:
        parameters:
          - name: app
          - name: port
          - name: image
          - name: tag
          - name: env
          - name: internal
          - name: external
          - name: args
      dag:
        tasks:
          - name: create-deployment
            template: deployment
            arguments:
              parameters:
                - name: app
                  value: "{{inputs.parameters.app}}"
                - name: image
                  value: "{{inputs.parameters.image}}"
                - name: tag
                  value: "{{inputs.parameters.tag}}"
                - name: env
                  value: "{{inputs.parameters.env}}"
                - name: port
                  value: "{{inputs.parameters.port}}"
                - name: args
                  value: "{{inputs.parameters.args}}"
          - name: create-internal-svc
            when: '{{inputs.parameters.internal}} == true'
            template: internal-svc
            arguments:
              parameters:
                - name: app
                  value: "{{inputs.parameters.app}}"
                - name: port
                  value: "{{inputs.parameters.port}}"
          - name: create-external-svc
            when: '{{inputs.parameters.external}} == true'
            template: external-svc
            arguments:
              parameters:
                - name: app
                  value: "{{inputs.parameters.app}}"
                - name: port
                  value: "{{inputs.parameters.port}}"

```

```
57      - name: deployment
58        inputs:
59          parameters:
60            - name: app
61            - name: image
62            - name: tag
63            - name: env
64            - name: port
65            - name: args
66        resource:
67          action: create
68          setOwnerReference: true
69          manifest: |
70            apiVersion: apps/v1
71            kind: Deployment
72            metadata:
73              name: {{inputs.parameters.app}}
74            spec:
75              selector:
76                matchLabels:
77                  name: {{inputs.parameters.app}}
78              template:
79                metadata:
80                  labels:
81                    name: {{inputs.parameters.app}}
82                spec:
83                  containers:
84                    - name: {{inputs.parameters.app}}
85                      image: {{inputs.parameters.image}}:{{inputs.parameters.tag}}
86                      ports:
87                        - containerPort: {{inputs.parameters.port}}
88                      args: ["{{inputs.parameters.args}}"]
89                      envFrom:
90                      - configMapRef:
91                          name: {{inputs.parameters.env}}
92      - name: internal-svc
93        inputs:
94          parameters:
95            - name: app
96            - name: port
97        resource:
98          action: create
99          setOwnerReference: true
100         manifest: |
101           apiVersion: v1
102           kind: Service
103           metadata:
104             name: service-internal-{{inputs.parameters.app}}
105           spec:
106             type: ClusterIP
107             selector:
108               name: {{inputs.parameters.app}}
109             ports:
110               - port: {{inputs.parameters.port}}
111
```

```
112       - name: external-svc
113         inputs:
114           parameters:
115             - name: app
116             - name: port
117         resource:
118           action: create
119           setOwnerReference: true
120           manifest: |
121             apiVersion: v1
122             kind: Service
123             metadata:
124               name: service-external-{{inputs.parameters.app}}
125             spec:
126               type: NodePort
127               selector:
128                 name: {{inputs.parameters.app}}
129               ports:
130                 - port: {{inputs.parameters.port}}
```

Listing A.9: WorkflowTemplate creating Kubernetes Services and Deployment

A.9.2 Download csv

```
1       - name: download-csv
2         container:
3           image: alpine/curl:latest
4           volumeMounts:
5             - name: workdir
6               mountPath: /work
7           command: [sh, -c]
8           args: ["curl {{workflow.parameters.url}} > /
                work/{{workflow.parameters.file-name}}"]
```

Listing A.10: Workflow template downloading CSV and storing into persistent volume

A.9.3 Shorten csv

```
1       - name: shorten-csv
2         inputs:
3           artifacts:
4             - name: code
5               path: /go/src/github.com/445455rk/argo-
                    workflows-demo
6               git:
7                 repo: https://github.com/445455rk/argo-
                      workflows-demo
8         container:
9           image: golang:1.18
10          command: [ sh, -c ]
11          args: ["cd␣/go/src/github.com/445455rk/argo-
                  workflows-demo␣&&␣go␣run␣.␣/work/{{
                  workflow.parameters.file-name}}"]
12          volumeMounts:
13            - name: workdir
14              mountPath: /work
```

Listing A.11: Workflow template trimming CSV content

A.9.4 MySQL write

```
- name: write-to-mysql
  retryStrategy:
    limit: "10"
  inputs:
    parameters:
      - name: dbip
      - name: user
      - name: password
      - name: database-name
      - name: filename
  script:
    volumeMounts:
      - name: workdir
        mountPath: /work
    image: 445455/mysql-helper:1.0
    command: [ python ]
    source: |
      import pymysql
      import csv

      db = pymysql.connect(host="{{inputs.parameters.dbip}}",port=3306, user="{{inputs.
          parameters.user}}", password="{{inputs.parameters.password}}")
      cursor = db.cursor()
      cursor.execute('create␣database␣if␣not␣exists␣{{inputs.parameters.database-name}}')
      db.select_db("{{inputs.parameters.database-name}}")
      with open('/work/{{inputs.parameters.filename}}', encoding="utf8") as csv_file:
          csv_data = csv.reader(csv_file)
          columns = next(csv_data)
          columns_with_types = [str(col) + "␣varchar(255)" for col in columns]
          create_table_query = 'create␣table␣if␣not␣exists␣{{inputs.parameters.database-
              name}}␣({0});'.format(',',join(columns_with_types))
          cursor.execute(create_table_query)
          db.commit()
          query = 'insert␣into␣{{inputs.parameters.database-name}}␣({0})␣values␣({1})'
          query = query.format(',',join(columns), ',',join(["%s"] * len(columns)))
          for data in csv_data:
              cursor.execute(query=query, args=data)
      db.commit()
      cursor.close()
```

Listing A.12: Workflow template reading CSV content and storing into MySQL DB

A.9.5 Print csv

```
- name: cat-csv
  container:
    image: alpine/curl:latest
    volumeMounts:
      - name: workdir
        mountPath: /work
    command: [ sh, -c ]
    args: [ "cat /work/{{workflow.parameters.file-
      name}} && wc -l /work/{{workflow.parameters
      .file-name}} > /work/lineCount.txt" ]
  outputs:
    parameters:
      - name: line-count
        valueFrom:
          path: /work/lineCount.txt
```

Listing A.13: Workflow template printing CSV content to stdout

Bibliography

[1] Tomáš Rebok et al. *ANALYZA – Výpočetní a orchestrační subsystém*. cze. Masarykova univerzita, 2020. URL: https://is.muni.cz/publication/1736351.

[2] Rizos Sakellariou et al. "Scheduling Workflows with Budget Constraints". In: *Integrated Research in GRID Computing: CoreGRID Integration Workshop 2005 (Selected Papers) November 28–30, Pisa, Italy*. Ed. by Sergei Gorlatch and Marco Danelutto. Boston, MA: Springer US, 2007, pp. 189–202. ISBN: 978-0-387-47658-2. DOI: 10.1007/978-0-387-47658-2_14. URL: https://doi.org/10.1007/978-0-387-47658-2_14.

[3] Jeremiah Lowin. Oct. 2021. URL: https://www.prefect.io/blog/announcing-prefect-orion/.

[4] F Mlder et al. "Sustainable data analysis with Snakemake [version 2; peer review: 2 approved]". In: *F1000Research* 10.33 (2021). DOI: 10.12688/f1000research.29032.2.

[5] Johannes Köster and Sven Rahmann. "Snakemake—a scalable bioinformatics workflow engine". In: *Bioinformatics* 28.19 (Aug. 2012), pp. 2520–2522. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/bts480. eprint: https://academic.oup.com/bioinformatics/article-pdf/28/19/2520/819790/bts480.pdf. URL: https://doi.org/10.1093/bioinformatics/bts480.

[6] Richard M Stallman, Roland McGrath, and Paul D Smith. *GNU make.*, 2001.

[7] B.P. Harenslak and J. de Ruiter. *Data Pipelines with Apache Airflow*. Manning, 2021. ISBN: 9781617296901. URL: https://books.google.cz/books?id=8EwnEAAAQBAJ.

[8] Brendan Burns, Joe Beda, and Kelsey Hightower. *Kubernetes*. Dpunkt Heidelberg, Germany, 2018.

[9] Rabi Padhy and Manas Patra. "Evolution of Cloud Computing and Enabling Technologies". In: *International Journal of Cloud Computing and Services Science (IJ-CLOSER)* 1 (Oct. 2012). DOI: 10.11591/closer.v1i4.1216.

[10] Ibrahim Abaker Targio Hashem et al. "The rise of "big data" on cloud computing: Review and open research issues". In: *Information Systems* 47 (2015), pp. 98–115. ISSN: 0306-4379. DOI: https:

//doi.org/10.1016/j.is.2014.07.006. URL: https://www.sciencedirect.com/science/article/pii/S0306437914001288.

[11] Michael Jackson, Kostas Kavoussanakis, and Edward W. J. Wallace. "Using prototyping to choose a bioinformatics workflow management system". In: *PLOS Computational Biology* 17.2 (Feb. 2021), pp. 1–13. DOI: 10.1371/journal.pcbi.1008622. URL: https://doi.org/10.1371/journal.pcbi.1008622.

[12] Pramod Singh. "Airflow". In: *Learn PySpark: Build Python-based Machine Learning and Deep Learning Models*. Berkeley, CA: Apress, 2019, pp. 67–84. ISBN: 978-1-4842-4961-1. DOI: 10.1007/978-1-4842-4961-1_4. URL: https://doi.org/10.1007/978-1-4842-4961-1_4.

[13] Scott Haines. "Workflow Orchestration with Apache Airflow". In: *Modern Data Engineering with Apache Spark: A Hands-On Guide for Building Mission-Critical Streaming Applications*. Berkeley, CA: Apress, 2022, pp. 255–295. ISBN: 978-1-4842-7452-1. DOI: 10.1007/978-1-4842-7452-1_8. URL: https://doi.org/10.1007/978-1-4842-7452-1_8.

[14] Paolo Di Tommaso et al. "Nextflow enables reproducible computational workflows". In: *Nature Biotechnology* 35.4 (Apr. 2017), pp. 316–319. ISSN: 1546-1696. DOI: 10.1038/nbt.3820. URL: https://doi.org/10.1038/nbt.3820.

[15] Yared Dejene Dessalk et al. "Scalable Execution of Big Data Workflows Using Software Containers". In: *Proceedings of the 12th International Conference on Management of Digital EcoSystems*. MEDES '20. Virtual Event, United Arab Emirates: Association for Computing Machinery, 2020, pp. 76–83. ISBN: 9781450381154. DOI: 10.1145/3415958.3433082. URL: https://doi.org/10.1145/3415958.3433082.

[16] Shyam BV. *Airflow vs. prefect-workflow management for Data Projects*. Oct. 2021. URL: https://towardsdatascience.com/airflow-vs-prefect-workflow-management-for-data-projects-5d1a0c80f2e3.

[17] Shubhnoor Gill on Apache Airflow, Osheen Jain on Data Ingestion, and Abhinav Chola on Data Ingestion. *7 best airflow alternatives for 2022*. Mar. 2022. URL: https://hevodata.com/learn/airflow-alternatives/.

[18] Pedram Navid. *Airflow, Prefect, and Dagster: An inside look*. Jan. 2022. URL: https://towardsdatascience.com/airflow-prefect-and-dagster-an-inside-look-6074781c9b77.

[19] Markus Schmitt. *Airflow vs Luigi vs Argo vs Kubeflow vs mlflow*. URL: https://www.datarevenue.com/en-blog/airflow-vs-luigi-vs-argo-vs-mlflow-vs-kubeflow.

[20] Samadrita Ghosh. *Prefect vs. airflow: Censius blogs*. URL: https://censius.ai/blogs/prefect-vs-airflow.

[21] JORDAN SEGALL. *The Rise of Workflow Orchestration Tools*. URL: https://assets-global.website-files.com/5e46eb90c58e17cafba804e9/5f8f885195ca2b64eb6d462c_200027%5C%20ON%5C%20UV%5C%20Workflow%5C%20Orchestration%5C%20White%5C%20Paper.pdf.

[22] Ian McGraw. *Picking a kubernetes orchestrator: Airflow, Argo, and prefect*. Dec. 2020. URL: https://medium.com/arthur-engineering/picking-a-kubernetes-orchestrator-airflow-argo-and-prefect-83539ecc69b.

[23] Aleksey Bilogur. *Orchestrating spell model pipelines using prefect*. Sept. 2021. URL: https://spell.ml/blog/orchestrating-spell-model-pipelines-using-prefect-YU3rsBEAACEAmRxp.

[24] Kelsey Taylor. *The best data orchestration tools that businesses should be aware of*. July 2021. URL: https://www.hitechnectar.com/blogs/the-best-data-orchestration-tools-that-businesses-should-be-aware-of/.

[25] Mihhail Matskin et al. "A Survey of Big Data Pipeline Orchestration Tools from the Perspective of the DataCloud Project *". In: Dec. 2021.

[26] Tom Yedwab. *Why we switched to airflow for Pipeline Orchestration | Khan Academy blog*. URL: https://blog.khanacademy.org/why-we-switched-to-airflow-for-pipeline-orchestration/.

[27] Xingwei Wang, Hong Zhao, and Jiakeng Zhu. "GRPC: A Communication Cooperation Mechanism in Distributed Systems". In: *SIGOPS Oper. Syst. Rev.* 27.3 (July 1993), pp. 75–86. ISSN: 0163-5980. DOI: 10.1145/155870.155881. URL: https://doi.org/10.1145/155870.155881.