COSMOS – A FRAMEWORK FOR CONTAINERISED, DISTRIBUTED EXECUTION AND ANALYSIS OF HYDRAULIC WATER DISTRIBUTION SYSTEM MODELS

Georg Arbesser-Rastburg[1], David Camhy[2] and Daniela Fuchs-Hanusch[3]

[1,2,3] Institute of Urban Water Management and Landscape Water Engineering, Graz University of Technology, Graz, Austria

[1] *georg.arbesser-rastburg@tugraz.at*, [2] *camhy@tugraz.at*, [3] *fuchs-hanusch@tugraz.at*,

Abstract

Many scientific problems related to water distribution systems like optimization problems or sensitivity analysis require the creation and execution of a large number of hydraulic models. To reduce computation times, different approaches have been used in the past, often by employing multiple CPU cores to solve the hydraulic equations of a single model or to simulate multiple models in parallel on a single computer. However, these approaches often cannot make use of distributed computing. Furthermore, using these approaches in applications with a (web-based) graphical user interface (GUI) often requires the development of tailored software solutions and application programming interfaces (API) to link GUI and model execution backend.

To tackle these issues, we propose COSMOS (Containerised Model Simulator), a highly scalable Python-based framework which allows for the modification of hydraulic models using OOPNET, an API between Python and the hydraulic solver of the modelling software EPANET and can run model simulations. Simulation results can be then further analysed while all these tasks run encapsulated in containers in a cluster. It also allows to easily link the described functionality with other applications by providing a REST API.

A standard-based OpenAPI allows for passing hydraulic models and running scientific workflows via HTTP and generating clients based on the provided OpenAPI schemas, which simplifies the creation of web-based user interfaces. Python was chosen because of its growing spread in the scientific community, the availability of data processing and optimization packages and its high code readability.

Prefect, a data workflow orchestration framework, is employed to create workflows, starting with the transformation of hydraulic models into JSON representations for further use in web applications. The models can then be executed and simulated distributed over the available CPU cores (locally or in a cluster). Further tasks for doing analysis in the cluster can be easily added if necessary. Hydraulic models and simulation artifacts are stored on S3-compatible storage and can be easily retrieved.

A main advantage of this approach is the use of containers, which allows for reproducible workflows. Compared to other high performance computing approaches and container-based systems, Prefect has the advantage of being able to keep dedicated worker nodes available for use. This comes in handy especially when dealing with relatively short computation times where the start of a container might take longer than the actual simulation. Additionally, simulation data post-processing can be easily added to workflows in Prefect. Furthermore, as the structure of COSMOS is highly scalable, it can be used for different levels of problem complexity and simulation runtimes.

Keywords
Hydraulic modelling, EPANET, containerisation, model simulation backend.

1 INTRODUCTION

Hydraulic models have a wide variety of applications, from optimization problems like optimal design or optimal sensor placement, to simulating different kinds of operating conditions like system failures, to tasks like leak localization. Several application programming interfaces between the hydraulic modelling software EPANET [1] and different programming interfaces have been developed to simplify these tasks [2]–[5]. While there are several examples of web applications that use simulation results from water distribution system models [6]–[9], there appears to be only one framework that is targeted at linking web applications with simulating hydraulic models, epanet-js [3].

To link such a web interface with simulating hydraulic models, an interface to a hydraulic solver is necessary and the library epanet-js, written in JavaScript, provides such an interface. However, it is not does not provide a framework for running more complex workflows and that supports the fast simulation of many hydraulic simulations in parallel [3]. Speeding up the execution of hydraulic models in general has already been of interest to researchers in the past.

One possible way to speed up model execution is to employ several CPU cores in parallel to simulate a model. Wu and Elsayed developed a parallelization algorithm to concurrently run hydraulic and quality simulations [10]. Some employed parallelization to compute the individual models faster by distributing the computational load across several CPU cores [11], [12]. This approach's scalability however is limited, since the execution is constrained by the number of processor cores available at the used workstation. Additionally, Burger et al. found that they were not able to develop a solver that outperformed EPANET's solver when using real-world hydraulic models and even raised the question, if any solver will ever be faster than EPANET's original solver [11].

Another approach is to parallelize the computation of a set of hydraulic models. In this case, every processor core handles the computation of a single model. This approach however is again limited by the number of processor cores available at the used workstation [13].

Instead of doing calculations on a local workstation, computations can also be outsourced to a dedicated server infrastructure. Using distributed computing the computational load is spread across many computers and therefore the number of available CPU cores is increased. This approach has already been employed in the field of water distribution systems in the past. Alonso et al. used the Message Passing Interface (MPI) to distribute the calculation of hydraulic equations across several PCs with a custom hydraulic solver [14]. Wu and Zhu also used MPI to distribute the optimization of pump schedules [15]. Hu et al. developed a genetic algorithm for sensor placement that is based on the cluster computing framework Apache Spark [16]. Additionally, several frameworks written in the popular programming language Python have been developed for distributing calculations in a cluster in a simple manner. Examples for such frameworks are Celery, Apache Airflow, Prefect, and Dask.

Any results generated in a scientific context should be reproducible by others to validate conclusions or develop new methods based on existing research. Containers that package the environment and software required for running code can help facilitate reproducibility [17].

Containers are similar in their functioning to virtual machines (VMs) as both concepts rely on virtualization. In contrast to VMs, containers however virtualize software while virtual machines also virtualize the underlying hardware. Containers in contrast to virtual machines share the host's kernel and offer almost the same performance as the host's operating system, decreased starting times and a reduced storage footprint on the host machine [14].

Containerizing model execution has several advantages. First, containers can help with analysis reproducibility. When using a suitable container image repository (e.g., Docker Hub or GitLab), versioned images can be kept as an archive and later be reused. Second, it leads to the possibility to quickly scale the number of available worker nodes by using a suitable orchestration software (e.g., Kubernetes or Docker Swarm). Third, containers can be easily deployed on workstations locally, to develop and test the containers. This can be further enhanced with continuous integration and continuous delivery (CI/CD) systems that automatically test analysis tasks for their correctness [15] and build the images for the containers. A container image is a blueprint for containers that include the entire environment and include all code necessary for running the code in the container.

In this paper we present COSMOS (Containerised Model Simulator), a framework for containerised hydraulic simulations that employs cloud computing and can be accessed by web-apps via a standard-based API. Section 2 describes the requirements that were determined while developing a frontend for hydraulic model simulations. In section 3 different available cloud computing frameworks are analysed regarding their suitability as web-app backends for scientific applications. COSMOS itself is described in section 4 and section 5 finally gives an outlook into further possible enhancements and use cases of COSMOS.

2 REQUIREMENTS FOR HYDRAULIC MODEL SIMULATION WEB-APP BACKENDS

Requirements for a hydraulic model simulation backend were derived during the development of an interactive web-based application that allows users to execute complex scientific workflows that are based on hydraulic simulations (e.g., sensitivity analysis or calibration).

The frontend should provide users with the possibility to manage stored models and their simulation and analysis results and run pre-defined algorithms or tasks via a REST API. Optionally, a graphical user interface (GUI) should provide a platform for easy execution of workflows.

First, the requirements for the backend were derived. They can be grouped into general, web application specific and scientific requirements. Below is a list and description of the requirements that were identified during an internal co-creation process:

Common requirements:

- Scalability

  The system should be able to cope with both very short tasks as well as longer-running and more complex tasks that require the simulation of many hydraulic models at the same time. This required a framework that allows for scaling from a small number of computing nodes to a large-scale computing cluster. The web application and the distributed computing framework should finally be deployed to a Kubernetes cluster.

- Python-based or existing Python client

  A well-established programming language was required to reach a wide audience. The choice fell on Python due to its simple syntax, its many existing libraries (by May 2022 the Python Package Index listed more than 375.000 projects) in general and especially the libraries tailored towards scientific usage like NumPy, SciPy or pandas.

- Usage of open-source and free software

  Open-Source provides a transparent view on the implementation of the underlying algorithms and delivers an easy way to communicate about issues in implementations or get helpful support from the community. Using free software in addition mitigates financial obstacles for reproducing results.

- Stability and support

  A mature and well supported framework was required to guarantee long-term support. Also, the documentation should be extensive and well written to enable new users to get into the framework more easily.
- General Data Protection Regulation (GDPR) conformity

  Scientific analysis is sometimes based on personal data that must be treated according to GDPR requirements. An on-premises solution was sought after to keep all data on internal servers in a controlled environment.
- Easy-to-use

  One of the most important requirements was the usability for users. The platform should provide convenient and easy-to-use entry points for both experienced developers via a standard-based API as well as for users without a dedicated IT background via a web GUI.
- Centralized and findable

  To help other scientist in getting insights in already processed research topics, a history of executed workflows and their metadata should be centrally stored together with used parameters and obtained results.
- Monitoring and alerting

  Users should be alerted about failed tasks and workflows via multiple channels (e.g., email or different messengers) and querying the current state of running workflows as well as their results should be possible. Keeping track of computing resources requires easy-to-use monitoring that allows for assessing the computing resources in use.

Web application specific requirements:

- Low-latency Execution

  Users should be able to interactively explore algorithms and their results in a responsive environment. Short running tasks should provide immediate feedback, which lead to the requirement of “low-latency” workflow executions. This means that when executing a relatively small number of models with a short runtime, the system should return results as quickly as possible. This requires a framework that adds little overhead to the executed analysis and simulation tasks.
- Easy API Access

  Integrating the platform into other web apps should be possible via an easily accessible API. Creating clients in different programming languages should facilitate the integration in other apps, for example by providing an OpenAPI or GraphQL API.
- Result and model storage

  Results and models should be stored centrally and in an easy-to-use fashion. Versioning of results and models should guarantee reproducibility. Hosting the storage on-premises should be possible as well as access via APIs in different programming languages.

Scientific requirements:

- Reproducible and repeatable complex workflows

  Main goals for research tasks and workflows are reproducibility and repeatability, so that others can evaluate and reproduce any generated results. This requires management of input and output data, the corresponding metadata and which programming code or workflow description was used to generate the results. Easy exchange and versioning of workflow descriptions also contributes to the openness of the methods used and makes it easier for other researchers to reproduce the results. This also includes any used software in the workflow run's environment, for instance EPANET if its hydraulic solver is being used.

Depending on the algorithms used (e.g., evolutionary algorithms) the need for complex workflows can arise, where one or many steps can be dependent on the previous ones. Therefore, only frameworks that already support separating tasks in terms of small units of code and task dependencies were considered.

- Easy integration into existing scientific software packages

  Since Python and its accompanying scientific stack offer a variety of scientific tools, a solution that provides a similar interface was required. Users familiar with those tools should be able to transition seamlessly into the new distributed computing environment.

- Easy local development, testing and debugging

  Developing workflows locally should allow researchers to test and debug their code. Being able to run and test algorithms locally before running them in a computing cluster was deemed necessary to support the scientific workflow.

- Integration with already existing workflows

  While the execution of models is the main use case discussed in this paper, an integration of other already existing scientific tasks like machine learning or measurement data pipelines would provide a great benefit. The focus however lies on running hydraulic simulations.

3 DISTRIBUTED COMPUTING FRAMEWORKS

One of the most important aspects of the development of COSMOS was the evaluation of distributed computing frameworks with Python bindings.

Based on the requirements stated above, several frameworks were evaluated. Frameworks that did not fulfil all requirements but where the missing features could be implemented with low effort were also considered. Exchange and versioning of workflow descriptions if not already integrated into the framework can for instance be provided by git or other versioning systems. The evaluation was based on the framework documentations and small test runs to get to know the frameworks. Additionally, some of the software packages listed below have been in use at the Institute for several years so limitations and features were already clear.

All the frameworks were open-source and freely available. They all provided enough documentation and support to get to know the frameworks well enough to assess their features. Only frameworks where scalability according to the requirements listed above was given and which enable GDPR conformant workflows were considered. Support for containers should provide a reproducible environment and while the actual implementation between the frameworks is different, they all provide a way to use containers as execution layers.

Besides full-blown workflow scheduling solutions, GitLab as advocate for classical DevOp platforms and Jenkins as a more general automation platform were taken into account. Celery and Dask, while being more low-level in their abstractions, were also evaluated especially because the web-app approach requires a low-latency execution of certain workflows and tasks. Argo Workflows was considered because it provides support for running tasks in Kubernetes clusters as well. HTCondor was added to the mix as a more classical batch system which is often readily available on super-computers in scientific infrastructures. Prefect as a rather new competitor was considered because of its included abstractions and the good documentation. Finally, because of its widespread usage for cloud computing tasks, Apache Airflow was considered.

- Apache Airflow [18]

  Apache Airflow is a workflow scheduling and monitoring solution. It provides dynamic workflow descriptions written in Python, which helps users who already have experience in Python to generate more formalized workflow descriptions. Versioning of workflow descriptions is therefore very easy using git or other version control systems. It provides modular executors which allow scaling to different infrastructures like Kubernetes. It can act as layer over Celery and Dask which provides great flexibility. Integrated monitoring and logging as well as great expandability would make this a great solution for many of the requirements. However, Apache Airflow uses a central scheduling loop and jobs require a distinct execution date and time, which does not cover the use case of interactive web applications very well. First tests also showed that it did not behave according to the low-latency requirement when many workflows with small fast returning tasks are executed.

- Argo Workflows [19]

  Argo Workflows is a Kubernetes based and container native workflow-engine with good documentation and support for complex workflows. Workflows are created using Kubernetes manifests and can therefore easily be versioned and integrated in Kubernetes native GitOps frameworks like Argo CD. By using Argo Events the scheduling of workflows can be abstracted and there are many event-sources supported. Protocols like MQTT or NATS could then be used to send events that trigger a workflow execution.

  Being (only) Kubernetes based can be seen as a plus or minus depending on the use-cases stated in the requirements. While the main execution platform for the web-application is a Kubernetes cluster that can be easily scaled, reusing the workflow description in another infrastructure would not be possible.

  The main disadvantage however is the fact that Argo Workflows usually starts containers in the cluster only when needed and not permanently. While this provides great reproducibility and repeatability, it also adds significant overhead to the workflow execution and seems more suited to long running tasks. In our trials the container start time often exceeded the model execution time, especially when simulating small hydraulic models with run-times of less than a second. Therefore, the low-latency requirement is not fulfilled.

- Celery [20]

  Celery is a distributed task queue system with a large community of users. By building on a message broker like RabbitMQ or Redis and by deploying long-running workers, it adds very little. It provides a result backend abstraction which allows for keeping results connected to the task executions and therefore fulfils some of the centralization and findability requirements. In comparison to Apache Airflow and Prefect it appears to be a more low-level framework (for instance Apache Airflow has its dedicated Celery executor). Celery does not expose a standard based API for starting workflow runs, although the tool Flower provides API endpoints for monitoring Celery [21]. Triggering workflows via an API would therefore require the implementation of a custom API with a web framework like Flask, FastAPI or Django.

- Dask [22]

  Dask provides readily available larger-than-memory data structures built on common interfaces like NumPy, pandas or Python iterators making it well-suited for many scientific workflows. Switching between a local and a distributed scheduler is easily possible and does not impact the basic algorithms' design. The distributed scheduler adds very little overhead and seems similar in performance to Celery with its message broker approach. Also of interest are the multiple ways of deploying Dask clusters. It is possible to execute tasks on Kubernetes, via SSH and even on high performance computing (HPC) resources. This allows users to design scientific algorithms independent of the infrastructure it is running on. However, Dask does not include an abstraction layer for workflow definitions and task execution monitoring via an API.

- HTCondor [23]

  HTCondor is a batch software which was already used extensively at our Institute for large scale model execution and well documented. Being a more classical batch system, the tool DAGMan adds support for workflow definitions. An advantage of HTCondor is the possibility to use free computing resources from user workstations when they are not in use. It is very well suited for an extremely large number of models and monitoring can be performed over the command line. Preliminary tests showed that the low-latency execution of models was slower than the other approaches. Also there seems to be no already available monitoring solution that can be integrated easily in a web app.

- GitLab [24]

  GitLab as a representative for git implementations with support for continuous integration and continuous delivery (CI/CD) pipelines was considered as well. Some of the scaling requirements are fulfilled by the concept of GitLab runners and GitLab also provides an easy-to-use API for querying pipeline runs and their status. However, it seems not to be very well suited for the low-latency requirement and first tests showed a considerable delay between starting, scheduling and running pipeline executions.

- Jenkins [25]

  Jenkins is an open-source automation server and is used for CI/CD pipelines. While being mainly built for CI/CD pipelines, Jenkins can also be used for more general automation tasks. Using Jenkins pipelines, reproducibility and repeatability requirements can be fulfilled by using a version control system. Running pipeline steps either in a Kubernetes cluster, via SSH on Linux worker nodes and even Windows workstation, the scalability requirement was met as well. However, as with GitLab, low-latency execution of tasks could not be achieved.

- Prefect [26]

  Prefect is a data pipeline orchestration and runtime system that can use Dask for executing complex workflows. It offers abstractions for tasks, workflows, storage and executors. Among the supported storages are Docker images, Git, AWS Simple Storage Service (S3) and Bitbucket. Workflow definitions and result storages can be defined for workflows and tasks individually. A web interface can be used to trigger workflow runs. Workflow runs are then started by Agents. Agents provide the environment that is needed to start the workflow, i.e., they contain all the necessary code and dependencies and keep track of the workflow run's status. KubernetesAgents for instance start Kubernetes jobs that first pull the newest workflow definition from the designated storage and then start the workflow run. A GraphQL API supports triggering workflow runs, monitoring their status, and reading a task's result location in the used storage. Furthermore, since Dask can be used as an executor, Dask's API for larger-than-memory objects is available for usage as well. Prefect can be either used in its free and open-source version called Prefect server, or as the paid service Prefect Cloud. Prefect Cloud includes further functionalities like user authentication and a secret store. Prefect 2.0 Orion is currently under development, which will include an OpenAPI instead of a GraphQL API.

After the first evaluation, Celery and Prefect seemed to fulfil the requirements better than the other frameworks. To choose between them, the two frameworks were further evaluated in terms of the required service infrastructure to assess their integration in scientific workflows. Figure 1 shows Celery's service structure, while Figure 2 shows a simplified version of Prefect's structure.

Celery itself does not include an API that enables starting workflow runs via HTTP requests. A REST API would have to be implemented using a Python web framework like FastAPI, Flask or Django ("Producer"). This API would then send a task to a task queue system. Celery offers support for RabbitMQ, Redis, Amazon SQS and Zookeeper.

The tasks in the task queue are then scattered across Celery workers running in a cluster. These workers are responsible for executing the tasks ("Consumers") and need all the dependencies required for the task's execution in their environment. The task results are then sent to a central result backend. Out of the box, Celery supports Redis, RabbitMQ and SQLAlchemy as backend.

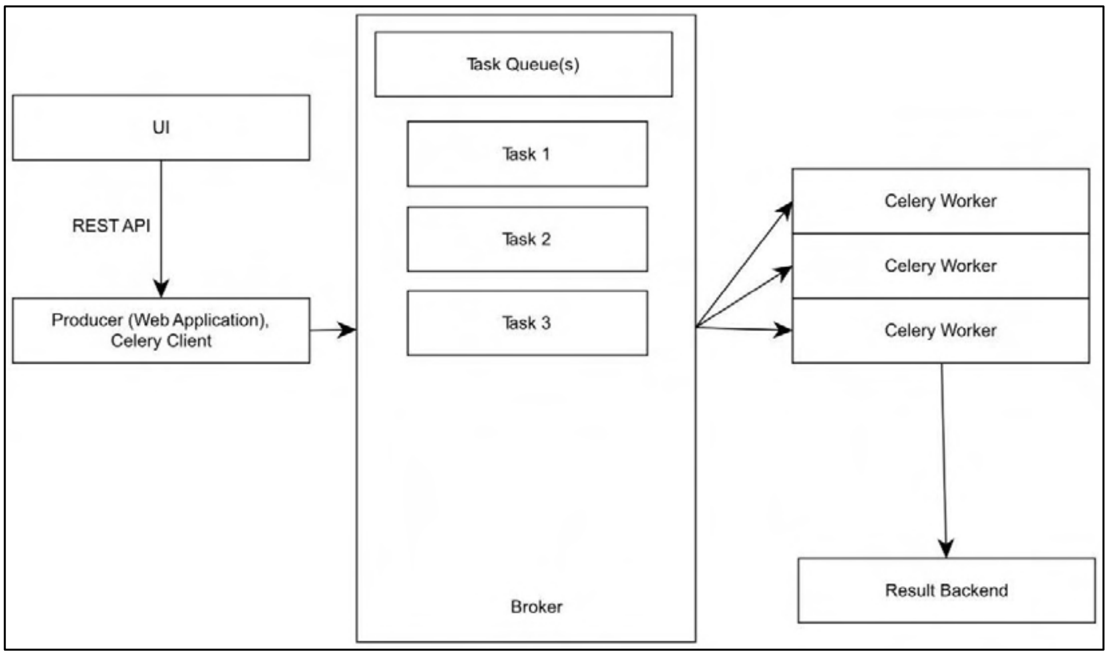

Figure 1. Celery service structure.

Prefect follows a slightly different approach by implementing its own task scheduling system and adding additional layers of abstraction. It also requires more dedicated services to be running. Prefect uses Apollo and Hasura to host a GraphQL API for many functionalities like starting workflow runs and querying workflow states. Prefect 2.0 however will implement an OpenAPI. Using the GraphQL API, users can trigger workflow runs by sending a request to the Prefect Server or Prefect Cloud, depending on whether Prefect's cloud service is being used or if Prefect is being hosted on-premises.

Prefect Agents query Prefect Server/Cloud for any scheduled workflow runs and are responsible for providing an environment that can execute the workflow (i.e., all dependencies and requirements are fulfilled in the environment). There are several different Agents available, that rely on different technologies for providing the environment (e.g., a DockerAgent that starts a workflow run from within docker containers or a KubernetesAgent that runs flows as Kubernetes Jobs).

How the Agent accesses the workflow definition can be controlled by choosing a suitable storage and including it in the workflow definition. Options range from Python modules accessible within the Agent's environment, to GitStorage that pulls the newest workflow definition via git or DockerStorage which pulls a container image that includes the workflow definition. If a storage solution like DockerStorage or GitStorage is chosen, the Prefect Agent pulls the newest workflow definition before executing it.

How a workflow is then executed, is part of the workflow definition. A LocalExecutor executes the workflow in the Agent's local environment, while a DaskExecutor can execute tasks in a Dask cluster.

Finally, results are stored in a Storage as well and the path to the result can be queried via the GraphQL API.

In addition to the services shown in Figure 2, Prefect also relies on additional services for stopping tasks that no longer communicate with the API, scheduling new tasks and maintenance routines.

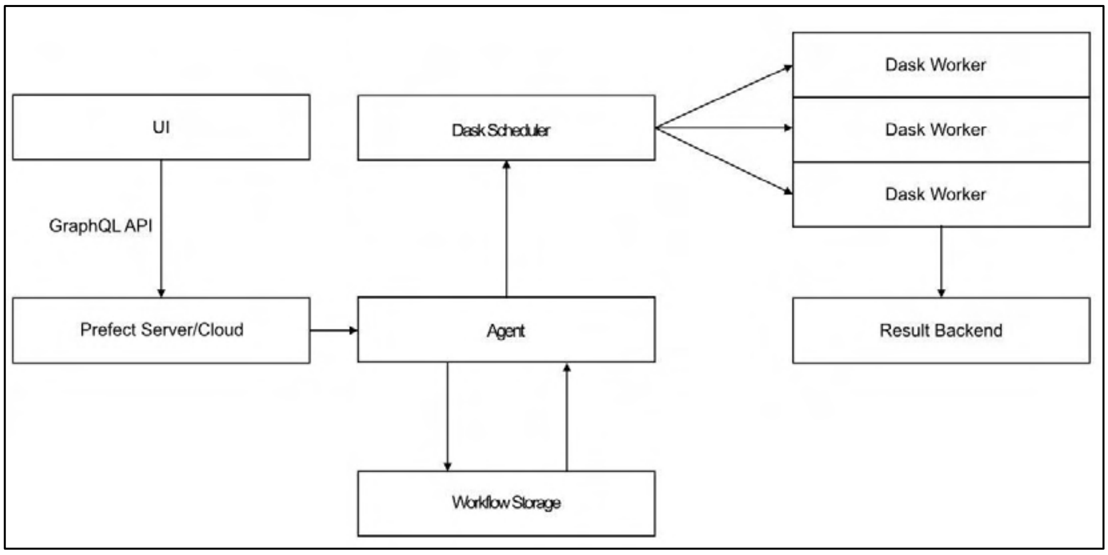

Figure 2. Prefect service structure.

Compared to Celery, Prefect includes more abstractions that can be used to control how artifacts and workflows are stored or how workflows are executed. It also provides a GUI for starting and monitoring workflow runs, whereas Celery does not provide any GUI. Different Agents allow for using different technologies to encapsulate a workflow depending on the available infrastructure and needs.

Native support for Kubernetes and Docker, a good documentation, the easy syntax that uses regular Python constructs for defining task dependencies and the additional functionality provided by Dask in the end lead to Prefect being chosen as basis for COSMOS.

4 COSMOS

COSMOS was created with the intention to provide a platform for executing a variety of different modelling tasks, from simulating simple hydraulic models to more complex analysis workflows if necessary. COSMOS is built upon several existing services, frameworks and standards that allow for an open structure that can be easily enhanced and is focused on reproducibility, easy usage, a central model and result storage and low-latency task execution.

To develop an API for webapps that is based on Python, one of two hydraulic modelling APIs can be used: the Water Network Tool for Resilience (WNTR) [5] and OOPNET [2]. Both provide basic functionalities like parsing EPANET input files, manipulating hydraulic models and simulating hydraulic models with EPANET. OOPNET was chosen as basis for COSMOS, since it has been in use at the Institute for several years and therefore many algorithms have already been implemented using it. This leads to an increased available codebase for future applications.

4.1 Service structure and functionality

COSMOS employs several services to satisfy the requirements specified in section 2:

- Prefect serves as workflow management and scheduling system
- Dask acts as execution backend
- The Amazon Simple Storage Solution (S3) compatible storage MinIO is used as model and simulation result storage
- FastAPI provides a RESTful interface following OpenAPI specifications and authorization
- Elasticsearch for finding models and simulation results in the storage backend

To use COSMOS, users first have to upload their hydraulic models via the provided REST API by sending a POST request to the FastAPI backend. This REST API wraps Prefect's GraphQL API. This was done to add a layer of abstraction so that the execution backend can be easily exchanged in the future, if Prefect 2.0 shows significant advantages after it has reached a stable state. As part of the upload request, users are able to add tags that can later be used for querying stored models, and a description. The hydraulic models are then converted into GeoJSON files, which are more widely compatible with web applications compared to EPANET input files.

GeoJSON is a file format based on JSON (JavaScript Object Notation) but includes geographical features. It implements different "Geometry" types (e.g., Points, LineStrings or Polygons), that include one or more "Positions" which themselves are an array of coordinates. Geometries are then combined with properties to form "Features". A collection of Features can finally be represented as a "FeatureCollection".

In COSMOS, features are equivalent with physical model components. Nodes (junctions, tanks and reservoirs) are modelled as Points and links (pipes, pumps and valves) as LineStrings. The model itself is represented as a FeatureCollection. Curves, patterns, and model settings are converted into regular JSON objects and also added to the GeoJSON model as so called "foreign members". Foreign members extend the GeoJSON specification with additional key-value pairs.

The conversion is handled by a Prefect task that employs pydantic, a package for creating data models based on the native type hints integrated in Python 3. Different validators can be used to for examples validate data types and value ranges in an intuitive manner. Additional custom validators can be added to the models as well (e.g., validating the IDs of model components corresponding to EPANET's ID length requirement or ensuring that a tank's initial tank level is between the tank's minimum and maximum water levels). A basic check of model validity can therefore be optionally run open model upload.

Pydantic can also write model instances into regular JSON files but does not include base models for GeoJSON objects. This functionality is provided by another package: geojson-pydantic. geojson-pydantic provides additional models that follow the GeoJSON specifications for the data types described above (e.g., Points and LineStrings but also Properties and FeatureCollections).

After a model has been uploaded and converted into a FeatureCollection object along with all settings and model components, this new model representation is stored on an S3 storage. The example deployment that comes with COSMOS includes MinIO, a S3 compatible object storage that can be hosted on-premises. Alternatively, Amazon S3 could be used as storage as well. S3 compatible storage was chosen for several reasons:

1. S3 storage is an object storage.

   COSMOS can simply create a GeoJSON object and store it on S3. Any web application can then load the GeoJSON object and parse it. Furthermore, other files can easily be stored on S3 without more complex preparatory steps like writing database models.

2. S3 provides versioning for objects.

   When models or simulation results are updated, the original file is not lost but can still be recovered afterwards.

3. Files on S3 can be encrypted.

   S3 provides the possibility to encrypt data very easily. While this was not a requirement for COSMOS, this might be beneficial for users who want to store sensitive data.

Upon model upload on S3, an event notification pushes the model JSON and any passed metadata to the search engine Elasticsearch. Elasticsearch is a search engine written in Java that stores data as JSONs. Clients can afterwards run queries using a RESTful API to find stored models.

After a model has been stored on S3, it is available for use in COSMOS. Three additional Prefect tasks are available in COSMOS:

- One task is responsible for querying a hydraulic model stored on S3 by model name and tags, loading the corresponding GeoJSON file and converting it back into an OOPNET model object.
- Alternatively, another Prefect task can be used to load a GeoJSON model from S3. This task simply takes the model's path on S3 as argument, loads the model's GeoJSON representation from the provided path from S3 and converts the model back into an EPANET model. This task comes in handy when a model has been uploaded to an S3 storage by another application and stored for use in COSMOS.
- Finally, the third Prefect task handles the simulation of an EPANET model. The task only takes one argument: the hydraulic model to simulate. OOPNET is used for simulating the model and creating a SimulationReport object. This object serves as a container for simulation results like node pressure and pipe flow rates in OOPNET. For further use in other applications, this object is again transformed into a JSON object via pydantic and stored on S3. The path to the result file is automatically stored in Prefect Server/Cloud and can be queried using either Prefect's GraphQL API or the REST API included in COSMOS.

While the tasks are executed, the COSMOS' FastAPI can be used to query their status via a dedicated API endpoint to check if a task has been executed successfully. Finally, a route allows users to query the results for a specific workflow execution.

4.2 Continuous integration

To keep the basis of COSMOS reliable, allow for linking COSMOS versions with simulation results for increased reproducibility and to make releases easier, a continuous integration pipeline has been implemented. The pipeline runs through several stages:

- Testing
- Version number generation
- Python package and Docker image building
- Creating a new release of COSMOS

The testing stage runs several hydraulic simulations (both single period analysis and extended period simulations) in COSMOS and compares the results with pre-calculated results that are included in COSMOS' testing module. This in combination with other unit-tests assures the reliable functioning and reproducibility of results generated with COSMOS. This however also requires an S3 storage available in the testing environment. If any test fails, the CI pipeline itself fails as well and no new COSMOS version is released.

Next, the version of the next COSMOS release is determined. The version numbers follow semantic versioning and is automatically generated based on the git commit messages that haven been submitted since the last release. This

After the new version number has been derived, first a Python package is built and uploaded to a Python package registry. Afterwards, a Docker image based on a Docker image provided by Dask itself (*daskdev/dask*) is built. In addition to the services and packages required for running a Dask worker node, EPANET, OOPNET and COSMOS are added to the Docker image.

In a final step, a new release is created that uses the previously derived version number, the corresponding Python package and Docker image and a changelog based on the Angular git commit messages.

Users are then able to install the framework on their local workstations and develop workflows for usage in COSMOS. If additional dependencies have to be available in a workflow run's execution environment, users can build their own Docker images if needed using the COSMOS image as base image.

4.3 Low-latency execution vs. flexibility during development

For low-latency execution tasks like running a small number of hydraulic simulations, the deployment of Prefect has to be optimized. This includes choosing a suitable Agent and Storage type for the workflow definition.

The Agent type used is an important aspect regarding execution speed on the one hand and flexibility during development on the other. A DockerAgent is able to pull a Docker image to provide the environment necessary for a workflow's execution every time a workflow is executed. While this approach is very flexible since the Agent requires hardly any further setup, it also leads to longer workflow execution times. A LocalAgent however is meant to be running in an environment that already fulfils the requirements and dependencies of the workflow. An easy way for keeping the time from workflow run submission to workflow execution low is running a LocalAgent in a dedicated container that already contains all the dependencies needed. However, this requires users to take care of running an Agent based on the most recent execution environment Docker image. A CI/CD pipeline can be used to simplify this process.

In addition to Agents, different Storage types are available in Prefect as well. Storage types like DockerStorage or GitLabStorage provide users with the possibility to load the most recent workflow definition from a central storage. This is well suited for tasks that don't require a low-latency execution since it also adds overhead to the execution. Pulling the latest workflow version can be skipped if the most recent version of the flow is already available in the Prefect Agent. This can be achieved by packaging all dependencies and workflows in a Docker image and using a ModuleStorage in the workflow definition. A ModuleStorage points to a workflow already available in a local Python module. Similar to the LocalAgent, this means that the most recent workflow definition has to be available in the execution environment image.

4.4 Extending COSMOS

Since COSMOS mainly provides additional features in Prefect, new Prefect tasks can be created and added to the existing workflows using the Prefect syntax. Hydraulic modelling tasks can be implemented using OOPNET's syntax, while simulation results are available as pandas DataFrames. Pandas is a powerful library for data manipulation and analysis which leads to high flexibility regarding result analysis in dedicated analysis tasks.

Due to being based on containerization, any newly added tasks and workflows have to be included in the used container images. This can be done by using a CI pipeline similar to the one used in COSMOS itself, or manually by using the Docker command line interface to build a new container image.

5 CONCLUSIONS

COSMOS is already being actively used while developing a scientific task execution platform. It provides an OpenAPI based RESTful API that can be used in any web application that requires more complex workflows or the execution of several hydraulic models at once. Hydraulic models and simulation results are both stored in conformity with the JSON and GeoJSON standards which makes working with them in a web context easy.

JSON files however tend to be verbose compared to EPANET input files and therefore increase in size rather quickly. This should be mitigated in the future by e.g., using compression or other file size reducing approaches. Also use cases outside of web development might benefit from more concise file formats.

Concerning storage, alternatives to S3 might be added in the future to support storing models and results in a database. Databases like PostGIS would add further usability options to software like the geographic information system QGIS and would enable the modelling of relationships between, for example, measurement data and hydraulic models.

The current version only implements basic functionalities regarding the handling of simulations, hydraulic models and simulation artifacts. Future releases it will be extended to include various algorithms and methods related to hydraulic modelling. Work on migrating already existing algorithms (e.g., roughness calibration and water distribution system sectorization) have already begun. While right now COSMOS is not publicly available yet, it will be hosted on a code sharing platform like GitHub to reach a wider scientific audience and gather a user community that can add new algorithms.

In addition to the platform for running scientific analysis with COSMOS, another useful feature would be a graphical user interface for managing hydraulic models and linked model simulations. There are already plans to implement such an interface for the task execution platform.

Currently, a new version of Prefect is being developed. The new version promises to be easier in usage, has a slimmer structure that requires less services to be running. Furthermore, the integration of Prefect flows and tasks into native Python code is claimed to be improved which would be beneficial for using COSMOS as a general model simulation backend in scripts.

COSMOS' approach could also be used for other types of models such as EPA SWMM using one of the Python to SWMM APIs available. This would shift COSMOS from being focused on water distribution system models to being a more flexible model execution backend.

Since COSMOS was developed while building a scientific platform for modelling and result analysis workflows, it can be used in a wide variety of ways. It supports long running tasks but also rather short running simulation tasks while providing a standard-based interface that allows for the easy creation of clients in many different programming languages.

Exemplary other use cases are more complex online EPANET editors that take factors into account that lead to an increase in necessary model execution runs like different operating conditions or Monte Carlo simulations. Furthermore, COSMOS could be integrated into analysis scripts written in Python to easily parallelize the execution of many hydraulic models at once while also making use of the reproducibility features of Prefect and COSMOS.

6 REFERENCES

[1] L. Rossman, H. Woo, M. Tryby, F. Shang, R. Janke, and T. Haxton, EPANET 2.2 User Manual. Washington, D.C: U.S. Environmental Protection Agency, 2020.

[2] D. B. Steffelbauer and D. Fuchs-Hanusch, “OOPNET: An object-oriented EPANET in Python,” Procedia Engineering, vol. 119, 2015, pp. 710–718.

[3] L. Butler, Á. Bautista, and C. Dzuwa, “EPANET-JS”, https://github.com/modelcreate/epanet-js, 2022.

[4] E. Arandia and B. J. Eck, “An R package for EPANET simulations,” Environmental Modelling & Software, vol. 107, Sep. 2018, pp. 59–63.

[5] K. A. Klisel, R. Murray, and T. Haxton, “An Overview of the Water Network Tool for Resilience (WNTR),” in Proceedings of the 1st International WDSA/CCWI Joint Conference, Kingston, Ontario, Canada, 2018, p. 8.

[6] D. B. Laucelli, L. Berardi, and A. Simone, “A Teaching Experiment Using a Serious Game for WDNs Sizing,” 13th International Conference on Hydroinformatics, vol. 3, 2018, pp. 1104–1112.

[7] D. Savic, M. Morley, and M. Khoury, “Serious Gaming for Water Systems Planning and Management,” Water, vol. 8, no. 10, Art. no. 10, Oct. 2016.

[8] G. Arbesser-Rastburg and D. Fuchs-Hanusch, “Serious Sensor Placement—Optimal Sensor Placement as a Serious Game,” Water, vol. 12, no. 1, Dec. 2019, p. 68.

[9] T. Bayer, D. P. Ames, and T. G. Cleveland, “Design and development of a web-based EPANET model catalogue and execution environment,” Annals of GIS, vol. 27, no. 3, Jul. 2021, pp. 247–260.

[10] Z. Y. Wu and S. M. Elsayed, “Parallelized Hydraulic and Water Quality Simulations for Water Distribution System Analysis,” in World Environmental and Water Resources Congress 2013, Cincinnati, Ohio, May 2013, pp. 892–902.

[11] G. Burger, R. Sitzenfrei, M. Kleidorfer, and W. Rauch, “Quest for a New Solver for EPANET 2,” Journal of Water Resources Planning and Management, vol. 142, no. 3, Mar. 2016, p. 04015065.

[12] O. Piller, M. Le Fichant, and J. E. Van Zyl, “Lessons learned from restructuring a hydraulic solver for parallel computing,” in WDSA 2012: 14th water distribution systems analysis conference, Adelaide, South Africa, Sep. 2012, pp. 398–406.

[13] S. Artina, C. Bragalli, G. Erbacci, A. Marchi, and M. Rivi, “Contribution of parallel NSGA-II in optimal design of water distribution networks,” Journal of Hydroinformatics, vol. 14, no. 2, Apr. 2012, pp. 310–323.

[14] J. M. Alonso et al., “Parallel Computing in Water Network Analysis and Leakage Minimization,” Journal of Water Resources Planning and Management, vol. 126, no. 4, Jul. 2000, pp. 251–260.

[15] Z. Y. Wu and Q. Zhu, “Scalable Parallel Computing Framework for Pump Scheduling Optimization,” in World Environmental and Water Resources Congress 2009, Kansas City, Missouri, United States, May 2009, pp. 1–11.

[16] C. Hu, G. Ren, C. Liu, M. Li, and W. Jie, “A Spark-based genetic algorithm for sensor placement in large scale drinking water distribution systems,” Cluster Computing, vol. 20, no. 2, Jun. 2017, pp. 1089–1099.

[17] B. M. Zaragozí, S. Trilles, and J. T. Navarro-Carrión, “Leveraging Container Technologies in a GIScience Project: A Perspective from Open Reproducible Research,” ISPRS International Journal of Geo-Information, vol. 9, no. 3, Art. no. 3, Mar. 2020.

[18] M. Beauchemin, “Apache Airflow”, https://github.com/apache/airflow, 2014.

[19] Argo Project Authors, “Argo Workflows”, https://argoproj.github.io/argo-workflows/, 2022.

[20] A. Solem, “Celery”, https://docs.celeryproject.org, 2021.

[21] M. Movsisyan, “Flower”, https://flower.readthedocs.io/en/latest/, 2012.

[22] Dask core developers, “Dask”, https://dask.org/, 2019.

[23] Center for High Throughput Computing, University of Wisconsin–Madison, “HTCondor”, https://htcondor.readthedocs.io/en/latest/, 2022.

[24] GitLab Inc, “GitLab”, https://about.gitlab.com/, 2022.

[25] Jenkins, ”Jenkins”, https://jenkins.io, 2018.

[26] Prefect Technologies, Inc., “Prefect”, https://docs.prefect.io/core/, 2022.