water

MDPI

Article

Transforming Hydrology Python Packages into Web Application Programming Interfaces: A Comprehensive Workflow Using Modern Web Technologies

Sarva T. Pulla [1], Hakan Yasarer [1] and Lance D. Yarbrough [2,*]

[1] Department of Civil Engineering, The University of Mississippi, University, MS 38677, USA
[2] Department of Geology & Geological Engineering, The University of Mississippi, University, MS 38677, USA
* Correspondence: ldyarbro@olemiss.edu

Abstract: The accessibility and deployment of complex hydrological models remain significant challenges in water resource management and research. This study presents a comprehensive workflow for converting Python-based hydrological models into web APIs, addressing the need for more accessible and interoperable modeling tools. The workflow leverages modern web technologies and containerization to streamline the deployment process. The workflow was applied to three distinct models: a GRACE downscaling model, a synthetic time series generator, and a MODFLOW groundwater model. The implementation process for each model was completed in approximately 15 min with a reliable internet connection, demonstrating the efficiency of the approach. The resulting APIs provide standardized interfaces for model execution, progress tracking, and result retrieval, facilitating integration with various applications. This workflow significantly reduces barriers to model deployment and usage, potentially broadening the user base for sophisticated hydrological tools. The approach aligns hydrological modeling with contemporary software development practices, opening new avenues for collaboration and innovation. While challenges such as performance scaling and security considerations remain, this work provides a blueprint for making complex hydrological models more accessible and operational, paving the way for enhanced research and practical applications in hydrology.

Keywords: hydrological modeling; web-based hydrology tools; MODFLOW; GRACE; model interoperability

check for updates

Citation: Pulla, S.T.; Yasarer, H.; Yarbrough, L.D. Transforming Hydrology Python Packages into Web Application Programming Interfaces: A Comprehensive Workflow Using Modern Web Technologies. *Water* **2024**, *16*, 2609. https://doi.org/10.3390/w16182609

Academic Editor: Giuseppe Pezzinga

Received: 19 July 2024
Revised: 7 September 2024
Accepted: 13 September 2024
Published: 14 September 2024

1. Introduction

Hydrological modeling has been instrumental in simulating, forecasting, and managing water resources [1–3]. These models enable researchers to predict water availability, assess the impacts of land use and climate change, and devise strategies for sustainable water management. Popular hydrological models include the Soil and Water Assessment Tool (SWAT) [4], which evaluates the effects of land management practices on water, sediment, and agricultural chemical yields in large complex watersheds. Another widely used model is the Hydrological Simulation Program—FORTRAN (HSPF) [5], which simulates the hydrological processes of watershed systems to predict water quality and quantity. MODFLOW [6], developed by the US Geological Survey, is a key groundwater model that simulates groundwater flow and addresses issues such as water supply and contamination. These models are crucial for modern water resource management, enabling more effective and informed decision-making processes.

Advancements in remote sensing, geographic information systems (GISs), and machine learning have significantly transformed hydrological and groundwater modeling. Remote sensing technologies, such as satellite imagery and aerial sensors, provide high-resolution spatial and temporal data on critical variables like precipitation, land cover, soil moisture, and evapotranspiration [7]. These technologies enhance model accuracy by offering detailed and continuous observations over large and often inaccessible areas [8]. GIS technology further elevates hydrological modeling by integrating, analyzing, and visualizing spatial data, enabling researchers to develop comprehensive models that incorporate diverse datasets and complex spatial relationships [9].

The synergy of remote sensing and GIS technologies facilitates the creation of sophisticated hydrological models that are more robust and reliable [10]. These models can simulate various hydrological processes with greater precision, assess the impacts of land use and climate changes, and improve water resource management strategies. For instance, integrating remote sensing data into hydrological models has improved flood forecasting [11] and drought monitoring [12], enabling more timely and effective responses to these events. Similarly, GIS-based models have been crucial in mapping groundwater recharge areas and contamination risks, supporting sustainable groundwater management practices [13].

Machine learning has revolutionized hydrological and groundwater modeling by enabling the analysis of vast amounts of data and the identification of complex patterns that traditional methods might miss. Machine learning algorithms can improve model predictions by learning from historical data and adapting to new information [14]. For example, machine learning techniques have been employed to enhance flood prediction models [15], optimize water resource management, and assess the impacts of climate variability on water availability [16]. Integrating machine learning with remote sensing and GIS technologies results in more adaptive and accurate models, providing valuable insights for managing water resources in a changing environment [17].

The usefulness of advanced hydrological models depends on their successful deployment and integration into operational systems. The FastAPI-Celery-Redis-RabbitMQ (FCRR) stack provides a robust Python-based solution for managing the computational demands of these models. It efficiently handles a continuous stream of requests by passing them into a queue for processing. This architecture facilitates asynchronous task processing, allowing long-running tasks, such as complex simulations or extensive data processing, to be handled in the background. This allows web services to remain responsive and scalable even under heavy computational loads.

Asynchronous processing is essential across various domains. For example, in image processing, it enables resource-intensive tasks like rendering to proceed without affecting overall system responsiveness. Similarly, in data analysis, it supports the efficient management of large-scale information processing, ensuring that complex computations do not hinder the performance of other tasks.

This asynchronous process approach is relevant to the domain-specific cases discussed earlier. Whether integrating remote sensing data, managing machine learning algorithms, or using large-scale GIS data, the FCRR stack ensures seamless deployment and operation of these tools. This method enhances the accessibility and practical use of advanced hydrological models, improving their usefulness in water resource management.

1.1. Study Objective

Despite significant advancements, many open-source hydrology Python packages with robust modeling capabilities remain inaccessible as web-based applications. This study aims to develop a comprehensive workflow for converting hydrology Python packages into web APIs using modern web technologies. By enhancing the accessibility and usability of these tools for researchers and practitioners, this approach bridges the gap between advanced hydrological models and practical applications. This workflow ensures that these models can be easily deployed and utilized for effective water resource management. Furthermore, the workflow aspires to provide a playbook for the scientific community to deploy their workflows more broadly, democratizing access to advanced modeling and machine learning tools.

By transforming hydrology Python packages into user-friendly web application programming interfaces (APIs), this study not only opens new avenues for research but also provides practical solutions for water resource management. This integration facilitates broader access and more effective application of advanced hydrological models, ultimately supporting more informed decision-making in water resource management. This innovative approach enhances the practical utility of existing models, empowering a wider range of users to leverage cutting-edge technologies to address critical water management challenges. The significance of this study lies in its potential to transform how hydrological research is conducted and applied, paving the way for more efficient, accessible, and impactful water resource management practices.

1.2. Overview of Existing Web-Based Hydrology Frameworks

The development of web-based frameworks for hydrological modeling has become increasingly important due to the need for accessible and scalable tools. In this literature review, we examine several prominent web-based hydrology frameworks, focusing on their core functionalities, strengths, and limitations. This analysis will provide valuable insights into the current state of web-based hydrological tools and identify areas where improvements are necessary to make open-source hydrology Python packages more accessible and user-friendly as web APIs.

1.2.1. Tethys Platform

The Tethys Platform is an open-source framework designed for developing and deploying web-based environmental applications [18]. Built on the Django web framework, Tethys integrates hydrological models, GIS data, and various other data sources. It supports compute-intensive workflows using Dask [19] and HTCondor [20], facilitating parallel processing and distributed computing for large-scale hydrological simulations. While Dask has revolutionized distributed computing, its implementation within the Tethys production environment can be complex. This complexity of setting up and configuring HTCondor and Dask within Tethys limits its accessibility and usability for users lacking extensive technical expertise [21].

1.2.2. HydroShare

HydroShare, developed by the Consortium of Universities for the Advancement of Hydrologic Science, Inc. (CUAHSI), Arlington, MA, USA, is a collaborative platform for sharing hydrological data and models [22]. It integrates with the CUAHSI Hydrologic Information System (HIS), supporting a range of hydrological models. While HydroShare excels in data management and collaboration, its primary focus on data sharing rather than seamless deployment of modeling workflows as web APIs can limit its effectiveness for running hydrological models directly through a web interface.

1.2.3. HydroServer

HydroServer, also part of the CUAHSI HIS, is an open-source server designed to manage and share time series hydrological data [23]. It provides tools for storing, visualizing, and analyzing hydrological data, facilitating interoperability and data sharing among researchers and institutions. However, HydroServer lacks comprehensive workflow management and API deployment capabilities, making it more suited for data management rather than providing a complete solution for web-based hydrological modeling.

1.2.4. HydroLang

HydroLang is an open-source, web-based programming framework for hydrological sciences [24]. Implemented in JavaScript, it allows users to run hydrological simulations and analyses directly in web browsers. HydroLang integrates various hydrological models and data processing tools, providing an accessible platform for data retrieval, manipulation, and visualization. The primary advantage of HydroLang is its emphasis on usability, reducing the complexity of setting up and running hydrological models. It offers pre-built modules and an intuitive interface, enabling quick simulation setups and result analyses.

However, being JavaScript-based, it may be less familiar to hydrologists who typically use languages like Python or Fortran.

1.2.5. HydroDS

HydroDS is a web-based platform that offers comprehensive data services to support physically based distributed hydrological models [25]. The primary focus of HydroDS is to facilitate access to and processing of the extensive datasets required for hydrological modeling. By providing tools for data retrieval, preprocessing, and visualization, HydroDS streamlines the preparation of model input data, thereby reducing the time and effort needed for these tasks. HydroDS integrates a variety of hydrological and geospatial datasets, automating common preprocessing steps such as watershed delineation, terrain analysis, and climate data extraction. This automation ensures that data are readily available and formatted correctly for use in hydrological models. Since HydroDS itself does not execute hydrological models, it requires additional tooling and setup.

1.2.6. HydroCompute

HydroCompute is a high-performance computational library designed for web-based hydrological and environmental science applications [26]. Utilizing advanced web technologies, HydroCompute supports both sequential and parallel simulations, enhancing computational efficiency through multithreading with web workers. It leverages engines like WebGPU, Web Assembly, and native JavaScript and facilitates local data transfers via peer-to-peer communication with WebRTC. The open-source and flexible architecture allows users to incorporate their own code, promoting effective data management and decision-making. The complexity of setup and configuration can be a barrier for users without significant technical expertise. Its reliance on client-side computation limits its scalability for large-scale simulations, as performance may be constrained by the capabilities of the user's device. Additionally, the learning curve for integrating and utilizing its diverse features can be steep for those unfamiliar with web-based scientific computing.

1.2.7. PyWPS

PyWPS is an implementation of the Open Geospatial Consortium (OGC) Web Processing Service (WPS) standard, designed to facilitate geospatial processing services on the web [27]. PyWPS serves as an intermediary that allows users to define geospatial processes in Python and expose them as web services. The primary function of PyWPS is to expose the schema for inputs and outputs in the WPS format, effectively functioning as an empty shell without inherent processing capabilities. This means that PyWPS itself does not perform any geospatial processing but enables the integration of external processing services through a standardized interface. The flexibility and standards compliance offered by PyWPS make it a versatile tool for deploying geospatial processes in a web environment. However, the lack of built-in processing capabilities means that users must rely on external tools and services to perform the actual computations. This design necessitates significant technical expertise to configure and deploy effectively, which can be a barrier for users seeking a more straightforward and user-friendly solution.

To conclude this section, while several web-based hydrology frameworks offer robust and capable tools, they often present significant challenges in terms of deployment, maintenance, and operationalization. Platforms like Tethys and HydroCompute provide powerful features but require extensive technical expertise to set up and manage. Similarly, frameworks such as HydroShare and HydroServer excel in data management but lack comprehensive workflow management and API deployment capabilities. The complexity and technical demands of these existing tools highlight the need for a simple, lightweight workflow that can be easily adopted by users, including those with limited technical backgrounds. The objective is to create a system that enables even novices to quickly deploy and utilize hydrology Python packages as web APIs, thereby enhancing the accessibility and usability of advanced hydrological models.

2. Materials and Methods

2.1. Web Technologies and Frameworks

The development of a workflow to transform hydrology Python packages into web APIs requires the use of modern web technologies and frameworks. These technologies ensure the resulting web APIs are robust, scalable, and user-friendly. Given the complexity and computational demands of hydrological models, it is essential to employ tools that streamline the development and deployment process. Simplicity and ease of use are crucial, especially for users who may not have extensive technical expertise. Additionally, some newer frameworks offer significant advantages in performance and usability but have not yet been widely adopted in the scientific community. Utilizing open-source technologies is particularly advantageous as it promotes collaboration, transparency, and continuous improvement. This section outlines the key web technologies and frameworks employed in this study, highlighting their roles and the rationale for their selection (Figure 1).

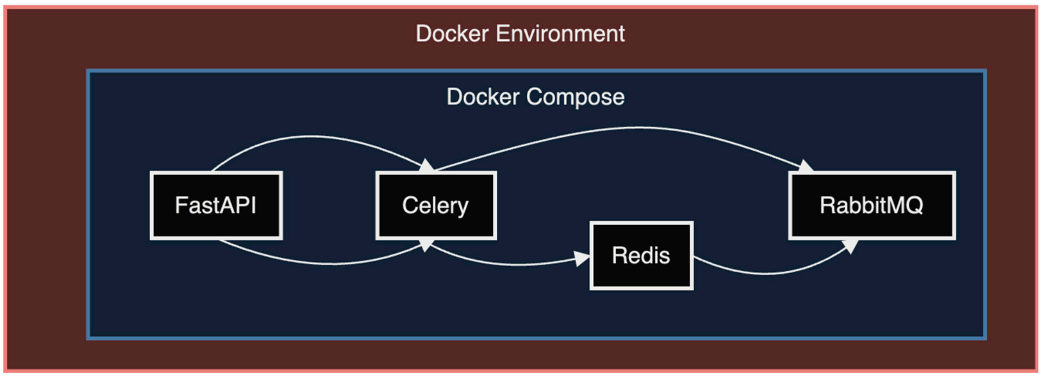


Figure 1. Architectural diagram of the workflow.

2.1.1. FastAPI

FastAPI was selected for its asynchronous capabilities, which are essential for handling the computationally intensive tasks associated with hydrological models [28]. Asynchronous processing, which is non-blocking, allows the application to handle multiple tasks concurrently, thereby improving efficiency and responsiveness. Built on standard Python type hints, FastAPI promotes efficient and intuitive coding practices. Leveraging the asyncio library, FastAPI can handle thousands of simultaneous connections, which significantly reduces latency and enhances throughput. This makes it particularly advantageous for real-time applications. Additionally, FastAPI's automatic generation of OpenAPI and JSON Schema documentation boosts developer productivity and improves API usability, which is critical for collaborative research environments.

2.1.2. Docker and Docker Compose

Docker ensures consistent environments across different stages of development and deployment, which is crucial for reproducibility in scientific research [29]. Docker containerizes applications by packaging the code along with all its dependencies, such as libraries and environment variables, into a single unit that can run reliably across different computing environments. This eliminates issues related to environmental discrepancies, ensuring consistent performance and interoperability across different systems and effectively addressing the common challenges associated with diverse localized testing environments. Docker Compose simplifies the orchestration of multi-container applications by allowing users to define and manage all containers and their interactions in a single YAML file [30]. This tool is particularly beneficial for setting up complex environments that involve multiple interconnected services, as it enables seamless management of dependencies and services with minimal manual intervention.

2.1.3. Celery and RabbitMQ

Celery was chosen for its robust task management capabilities, enabling asynchronous processing of long-running computations. Hydrological models often involve extensive data processing and computations that can be time-consuming. Celery allows these tasks to be executed in the background, ensuring that the main application remains responsive and available to handle new requests [31]. RabbitMQ acts as a reliable message broker, facilitating communication between different components by effectively queuing and distributing tasks [32]. This combination ensures that the application can efficiently manage and execute long-running processes without being bogged down, maintaining overall system performance and reliability.

2.1.4. Redis

Redis is employed for its high performance and scalability, making it ideal for real-time data storage, caching, and session management [33]. As an in-memory data store, Redis provides rapid access to data, which is crucial for applications requiring low-latency responses. Redis supports various data structures, such as strings, hashes, lists, and sets, enabling efficient handling of different types of data. This flexibility is particularly useful for session management, where quick access to user sessions is required, and for caching frequently accessed data, which can significantly reduce the load on primary databases and improve application performance. Redis's ability to handle high-throughput operations with minimal latency makes it a vital component in ensuring the responsiveness and efficiency of web APIs. Furthermore, Redis is used as the backend for Celery to store the state of tasks, track progress, and manage results, ensuring reliable task execution and monitoring.

2.2. Workflow Implementation

The process of transforming hydrology Python packages into web APIs involves a series of interconnected steps, each crucial for ensuring the resulting web application maintains the integrity and functionality of the original package while leveraging the advantages of web-based deployment. This workflow includes the following:

1. Analysis of package structure and dependencies.
2. API design and endpoint mapping.
3. Implementation of the FastAPI application.
4. Integration of asynchronous task processing.
5. Containerization and deployment configuration.
6. Comprehensive testing and validation.
7. Documentation generation.

This modular approach allows for flexibility in addressing the diverse requirements of various hydrological models while maintaining a consistent and reproducible conversion process. The following sections will explore each of these phases in detail.

The subsequent sections integrate conceptual explanations with practical examples to provide a comprehensive understanding of the framework. The linked GitHub repositories contain the code for the case studies, while key code snippets and configuration examples within the narrative demonstrate how theoretical principles directly inform implementation. This integrated approach is designed to cater to a diverse readership, ranging from those seeking conceptual understanding to those looking for practical guidance.

2.2.1. Analysis of Package Structure and Dependencies

The initial step in transforming a hydrology Python package into a web API involves a comprehensive analysis of the package structure and its dependencies. This analysis forms the foundation for all subsequent stages of the API development process.

1. Package Structure Analysis. The analysis begins with an in-depth examination of the overall structure of the Python package:

- Module Organization: The package is mapped out into its modules and submodules to understand the logical separation of functionalities within the package.
- Class and Function Hierarchy: The main classes and functions are identified, along with their relationships and dependencies, which informs the API design and helps decide which elements should be exposed as endpoints.
- Data Flow: The flow of data from input to output is traced, which is crucial for designing efficient API endpoints and determining where asynchronous processing might be beneficial.
- Configuration and Settings: Any configuration files or environment-specific settings the package relies on are identified, as these may need to be translated into API parameters or environment variables in the containerized setup.

2. Dependency Analysis. A thorough analysis of the package's dependencies is conducted as follows:
    - Direct Dependencies: The requirements.txt or setup.py file is examined to list all direct dependencies. Each dependency is evaluated for the following:
        - Compatibility with the target Python version.
        - Potential conflicts with other required libraries.
        - Availability of recent updates or known security issues.
    - Indirect Dependencies: Tools like pipdeptree are used to visualize the full dependency tree, including indirect dependencies, to identify potential conflicts or redundancies in the dependency chain.
    - System-Level Dependencies: Any system-level libraries or tools the package relies on are identified, and plans are made for their inclusion in the containerized environment.
    - Dependency Licensing: The licenses of all dependencies are reviewed to ensure compliance with the project's licensing requirements and to avoid any potential legal issues.
3. Computational Resource Assessment. The computational requirements of the package are assessed as follows:
    - CPU Usage: The package is profiled to understand its CPU intensity, identifying functions that may benefit from asynchronous processing or parallelization in the API.
    - Memory Usage: The memory footprint of typical operations is analyzed to inform server specifications and potential memory optimization in the API.
    - I/O Operations: The package's file I/O and database interactions are examined to design efficient data handling strategies in the API.
4. Identifying API Conversion Challenges. Based on the analysis, the following potential challenges in the API conversion process are identified:
    - Stateful Operations: Operations that maintain state between calls are noted, as these may require special handling in a stateless API environment.
    - Long-Running Processes: Processes that may exceed typical web request timeouts are identified, with plans for asynchronous processing solutions.
    - Data Volume: The typical volume of input and output data is assessed to inform choices for data transfer methods and storage solutions.
    - Package-Specific Quirks: Any unique behaviors or requirements of the package that may need special consideration in the API design are documented.
5. Documentation Review. A thorough review of the existing package documentation is conducted as follows:
    - User Guides: User guides are examined to understand the intended use cases and typical workflows of the package.
    - Examples and Tutorials: Examples and tutorials are collected, which will be valuable for creating sample API calls and usage guidelines.

By conducting this comprehensive analysis, a solid understanding of the package being converted is established. This knowledge directly informs API design choices, helps anticipate and mitigate potential issues, and ensures that the resulting web API faithfully represents the functionality of the original Python package.

2.2.2. API Design and Endpoint Mapping

The design of the API structure is a critical step in ensuring the usability and effectiveness of the converted hydrological models. Essentially, this process establishes a blueprint for user interaction with the hydrological models via the web.

A RESTful (Representational State Transfer) architecture was adopted for the API design. REST is a set of architectural principles that leverage web standards such as HTTP and URIs [34]. The key principles adhered to are as follows:

- Statelessness: Each client request to the server must contain all the necessary information to understand and process the request. The server does not store any client state between requests.
- Client–Server: The client and server are independent, allowing each to evolve separately.
- Uniform Interface: Standard HTTP methods (GET, POST, PUT, DELETE) are used for different operations:
  - GET: Retrieve data (e.g., obtain simulation results).
  - POST: Create new resources (e.g., start a new simulation).
  - PUT: Update existing resources (e.g., modify simulation parameters).
  - DELETE: Remove resources (e.g., cancel a running simulation).
- Resource-Based: The API is structured around resources such as models, simulations, datasets, and results.

When mapping Python functions to API endpoints, the following guidelines are followed:

- Core Model Functionalities: These are exposed as primary endpoints, such as the following:
  - /models/{model_id}/run: Initiates a model simulation.
  - /models/{model_id}/calibrate: Starts the model calibration process.
- Auxiliary Functions: These are mapped to secondary endpoints or incorporated as query parameters as follows:
  - /models/{model_id}/parameters: Retrieves or updates model parameters.
  - /simulations/{simulation_id}?include_metadata=true: Retrieves simulation results with an option to include metadata.
- Complex Workflows: These are decomposed into sequences of API calls. For instance, a complete modeling process might involve the following:
  - Uploading input data: POST/datasets.
  - Setting model parameters: PUT/models/{model_id}/parameters.
  - Running the simulation: POST/models/{model_id}/run.
  - Retrieving results: GET/simulations/{simulation_id}.

Handling input parameters and output formats is crucial for ensuring the API is both flexible and easy to use.

- Input Parameters: These are handled through a combination of the following:
  - Path variables: For identifying specific resources (e.g., {model_id} in/models /{model_id}/run).
  - Query parameters: For optional or filter-like parameters (e.g., ?start_date= 2023-01-01).
  - Request bodies: For complex or large inputs, typically sent as JSON.
- Output Formats: JSON is standardized format for structured data due to its widespread use and ease of parsing. For large datasets, binary formats may be used, and content negotiation is implemented to support multiple response formats.

API versioning is crucial for maintaining backward compatibility as the API evolves. Versioning is implemented in the URL path:/api/v1/models/{model_id}/run.

This allows for the introduction of breaking changes in new versions (e.g., /api/v2/. . .) while still supporting older versions.

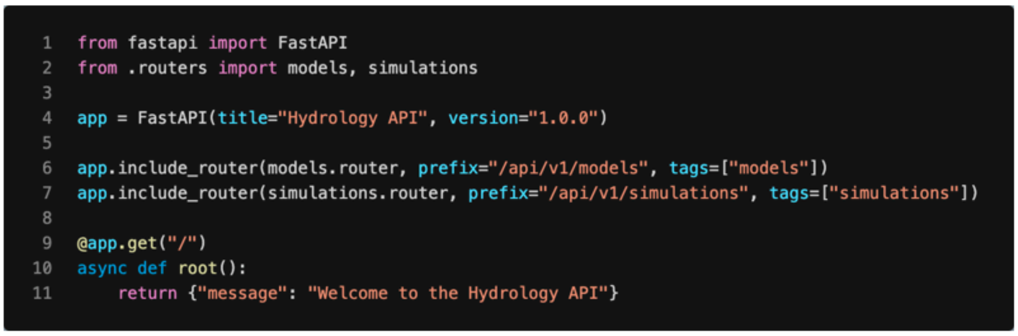
2.2.3. FastAPI Implementation

FastAPI is the chosen framework for implementing the API due to its high performance and use of standard Python type hints.

Application Structure. The FastAPI application is structured into modular components as follows:

- **main.py**: The entry point of the application, where the FastAPI instance is created and routers are included.
- **routers/**: A directory containing route handlers for different parts of the API (e.g., models.py, simulations.py).
- **models/**: Pydantic models defining the structure of request and response data.
- **services/**: Business logic and interactions with the underlying hydrological models.
- **utils/**: Utility functions and helpers.

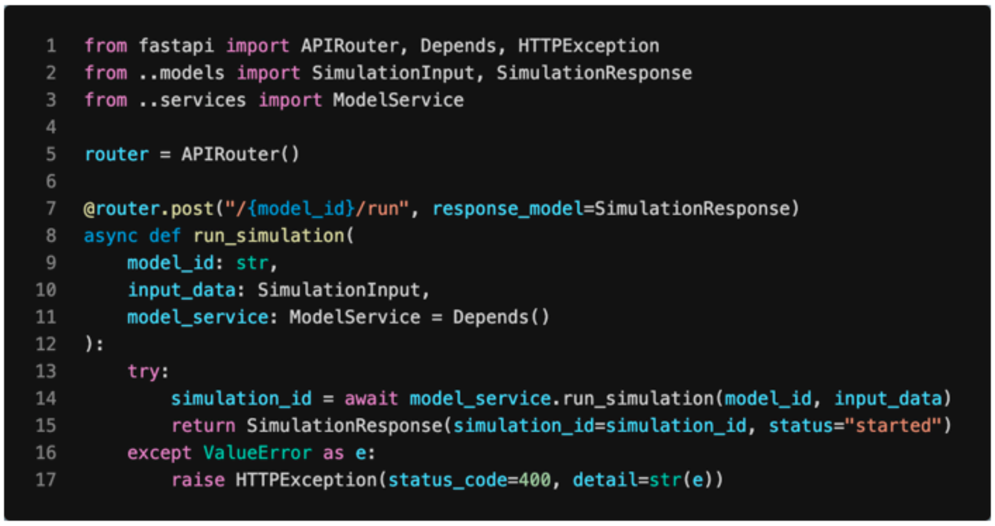
A simplified example of main.py is shown in Figure 2:



```python
from fastapi import FastAPI
from .routers import models, simulations

app = FastAPI(title="Hydrology API", version="1.0.0")

app.include_router(models.router, prefix="/api/v1/models", tags=["models"])
app.include_router(simulations.router, prefix="/api/v1/simulations", tags=["simulations"])

@app.get("/")
async def root():
    return {"message": "Welcome to the Hydrology API"}
```

Figure 2. A sample main.py file setup.

Route Handlers: Route handlers are the core of the API implementation. Route handlers define the API's response to various HTTP requests. Figure 3 demonstrates a route handler for running a model simulation:

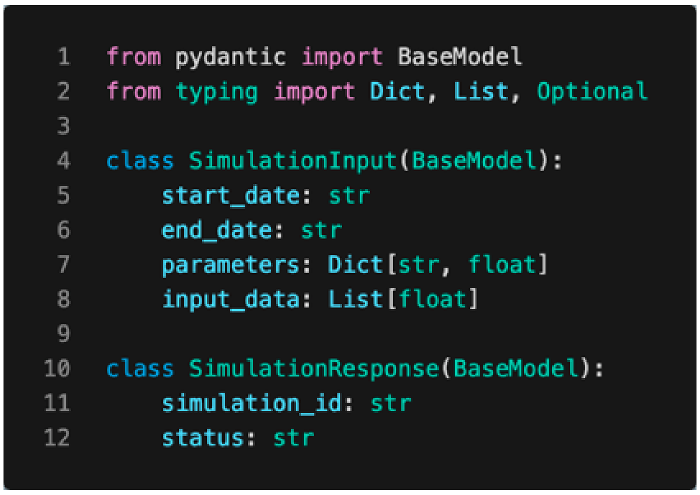


```python
from fastapi import APIRouter, Depends, HTTPException
from ..models import SimulationInput, SimulationResponse
from ..services import ModelService

router = APIRouter()

@router.post("/{model_id}/run", response_model=SimulationResponse)
async def run_simulation(
    model_id: str,
    input_data: SimulationInput,
    model_service: ModelService = Depends()
):
    try:
        simulation_id = await model_service.run_simulation(model_id, input_data)
        return SimulationResponse(simulation_id=simulation_id, status="started")
    except ValueError as e:
        raise HTTPException(status_code=400, detail=str(e))
```

Figure 3. A sample route handler for running a model simulation.

This handler receives a model ID and simulation input data, uses a ModelService to run the simulation, and returns a response with the simulation ID. Errors are managed by raising HTTP exceptions.

Data Models: Pydantic models define the structure of request and response data, providing automatic validation and clear documentation (Figure 4).



```python
from pydantic import BaseModel
from typing import Dict, List, Optional

class SimulationInput(BaseModel):
    start_date: str
    end_date: str
    parameters: Dict[str, float]
    input_data: List[float]

class SimulationResponse(BaseModel):
    simulation_id: str
    status: str
```

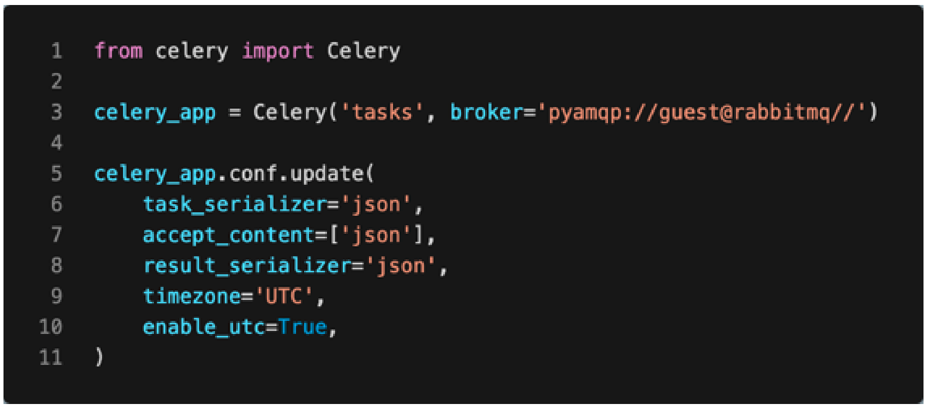
Figure 4. A pydantic data model to set up input and output format classes.

2.2.4. Asynchronous Task Processing

Hydrological simulations can be computationally intensive and time-consuming. To prevent these long-running tasks from blocking API responses and user interfaces, asynchronous task processing was implemented using Celery, a distributed task queue that is integrated with FastAPI and RabbitMQ as the message broker.

RabbitMQ serves as the message broker, facilitating communication between the FastAPI application and Celery workers. It ensures reliable message delivery and helps in distributing tasks across multiple worker processes.

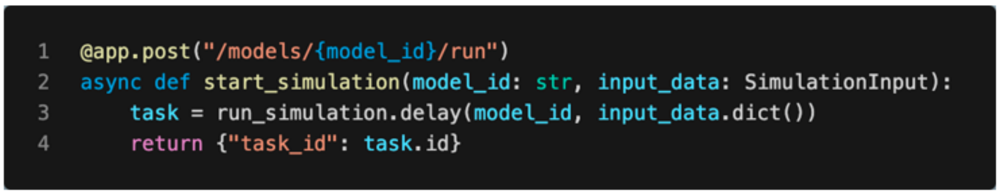
The core of this implementation involves configuring Celery with RabbitMQ and defining tasks for time-consuming operations (Figure 5).



```python
from celery import Celery

celery_app = Celery('tasks', broker='pyamqp://guest@rabbitmq//')

celery_app.conf.update(
    task_serializer='json',
    accept_content=['json'],
    result_serializer='json',
    timezone='UTC',
    enable_utc=True,
)
```

Figure 5. Setting up Celery and RabbitMQ to handle asynchronous tasks.

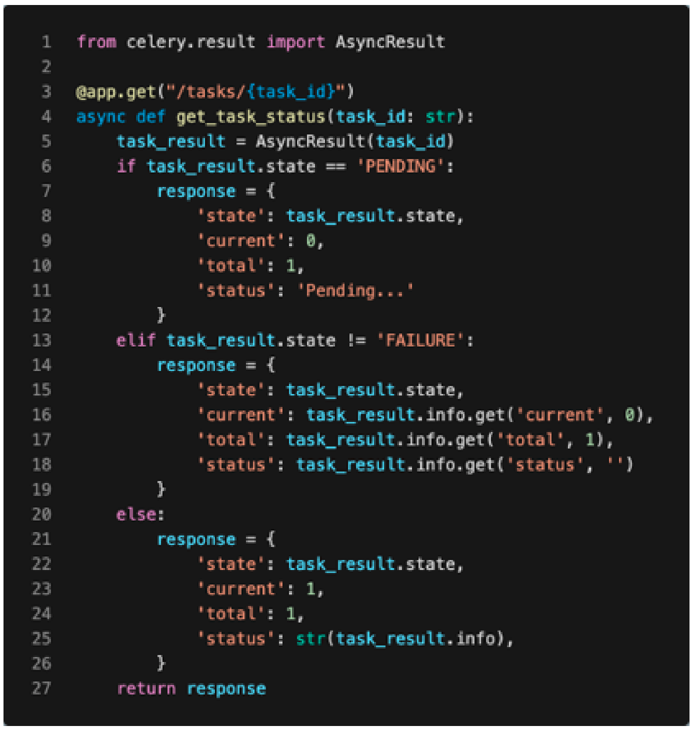
These tasks are then initiated from API endpoints, returning task IDs for subsequent status checks (Figure 6).



```python
@app.post("/models/{model_id}/run")
async def start_simulation(model_id: str, input_data: SimulationInput):
    task = run_simulation.delay(model_id, input_data.dict())
    return {"task_id": task.id}
```

Figure 6. Function for retrieving model run results.

Progress monitoring and result retrieval are implemented through additional endpoints that query the task status and fetch completed results from the Celery backend (Figure 7).



```python
from celery.result import AsyncResult

@app.get("/tasks/{task_id}")
async def get_task_status(task_id: str):
    task_result = AsyncResult(task_id)
    if task_result.state == 'PENDING':
        response = {
            'state': task_result.state,
            'current': 0,
            'total': 1,
            'status': 'Pending...'
        }
    elif task_result.state != 'FAILURE':
        response = {
            'state': task_result.state,
            'current': task_result.info.get('current', 0),
            'total': task_result.info.get('total', 1),
            'status': task_result.info.get('status', '')
        }
    else:
        response = {
            'state': task_result.state,
            'current': 1,
            'total': 1,
            'status': str(task_result.info),
        }
    return response
```

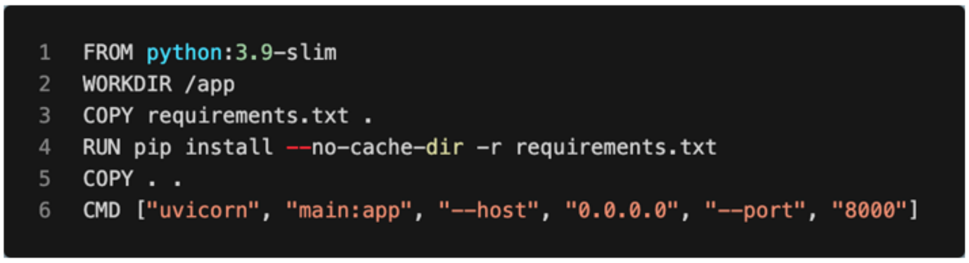
Figure 7. Tracking and retrieving results from celery backend.

This architecture, combining FastAPI, Celery, and RabbitMQ, allows the API to handle multiple simultaneous requests efficiently, improving overall system responsiveness and scalability for complex hydrological simulations. It also provides a mechanism for monitoring the progress of long-running tasks, enhancing the user experience for time-intensive operations.

2.2.5. Containerization and Deployment

The containerization and deployment phase ensures consistent environments across different stages of development and deployment, which is crucial for reproducibility in scientific research. This phase employs Docker for containerization, packaging the application and its dependencies into containers that run reliably across different computing environments.

The process begins with the creation of a Dockerfile, which defines how the application should be containerized (Figure 8).

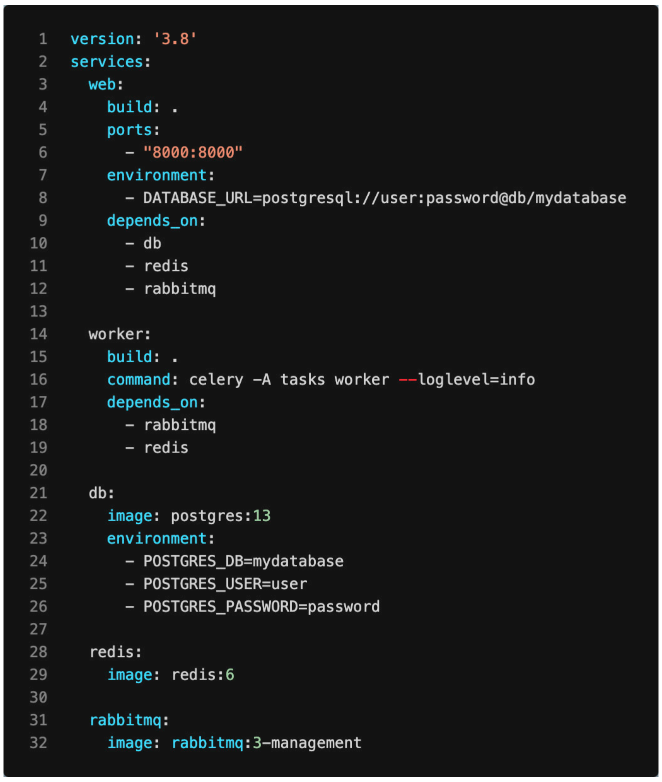


```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
COPY . .
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

Figure 8. A sample Dockerfile setup.

This Dockerfile specifies a base Python image, sets up the working directory, installs dependencies, copies the application code, and defines the command to run the FastAPI application using the Uvicorn server.

For orchestrating multiple containers, Docker Compose is utilized. A typical Docker Compose configuration for this workflow is shown in Figure 9.



```
version: '3.8'
services:
  web:
    build: .
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=postgresql://user:password@db/mydatabase
    depends_on:
      - db
      - redis
      - rabbitmq

  worker:
    build: .
    command: celery -A tasks worker --loglevel=info
    depends_on:
      - rabbitmq
      - redis

  db:
    image: postgres:13
    environment:
      - POSTGRES_DB=mydatabase
      - POSTGRES_USER=user
      - POSTGRES_PASSWORD=password

  redis:
    image: redis:6

  rabbitmq:
    image: rabbitmq:3-management
```

Figure 9. Setting up Docker Compose to handle multiple services.

This configuration defines services for the web API, Celery worker, PostgreSQL database, Redis (for caching), and RabbitMQ (as a message broker for Celery).

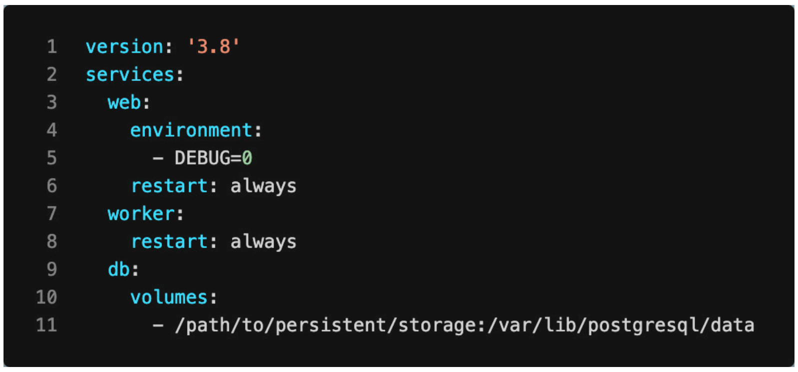
- Web Service: Builds the FastAPI application, exposes port 8000, and sets environment variables for the database connection. It depends on the database, Redis, and RabbitMQ services.
- Worker Service: Builds the application, runs the Celery worker with appropriate logging, and depends on RabbitMQ and Redis.
- Database Service: Uses a PostgreSQL image with specified environmental variables for the database name, user, and password.
- Redis Service: Utilizes a Redis image for caching.
- RabbitMQ Service: Uses a RabbitMQ management image for message brokering.

The deployment process involves building the Docker image and starting the services defined in the Docker Compose file. This can be achieved with the following commands:

- docker-compose build.
- docker-compose up-d.

These commands build the necessary images and start all services in detached mode, allowing them to run in the background.

Environment-specific configurations are managed through environment variables and Docker Compose override files. For instance, a production-specific configuration might be defined in a docker-compose.prod.yml file (Figure 10).



```
version: '3.8'
services:
  web:
    environment:
      - DEBUG=0
    restart: always
  worker:
    restart: always
  db:
    volumes:
      - /path/to/persistent/storage:/var/lib/postgresql/data
```

Figure 10. A docker-compose file configured for production setup.

This production configuration disables debug mode, sets services to restart automatically, and configures persistent storage for the database.

Security considerations in the containerization and deployment phase include the following:

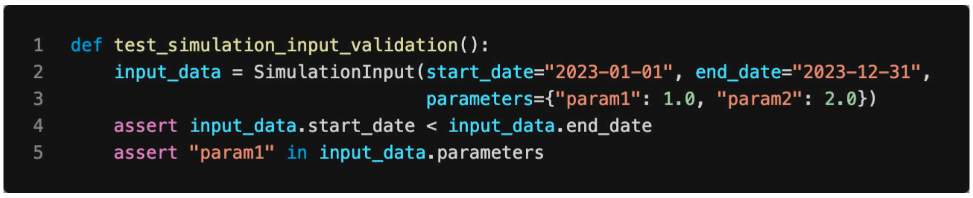
1. Using official base images to reduce the risk of vulnerabilities.
2. Minimizing the attack surface by only installing necessary packages.
3. Implementing proper secret management for sensitive data like database passwords.
4. Regularly updating all images and dependencies to patch known vulnerabilities.
5. Utilizing Docker's network features to isolate services that do not need to communicate with each other.

By following these containerization and deployment practices, the converted hydrology Python packages can be reliably and securely deployed in various environments, from local development setups to large-scale cloud deployments. This approach not only simplifies the deployment process but also enhances the reproducibility and scalability of hydrological modeling workflows.

2.2.6. Testing and Validation

The testing and validation phase is crucial for ensuring the reliability, accuracy, and performance of the converted web APIs. This phase encompasses functional testing, performance testing, and validation against the original Python package.

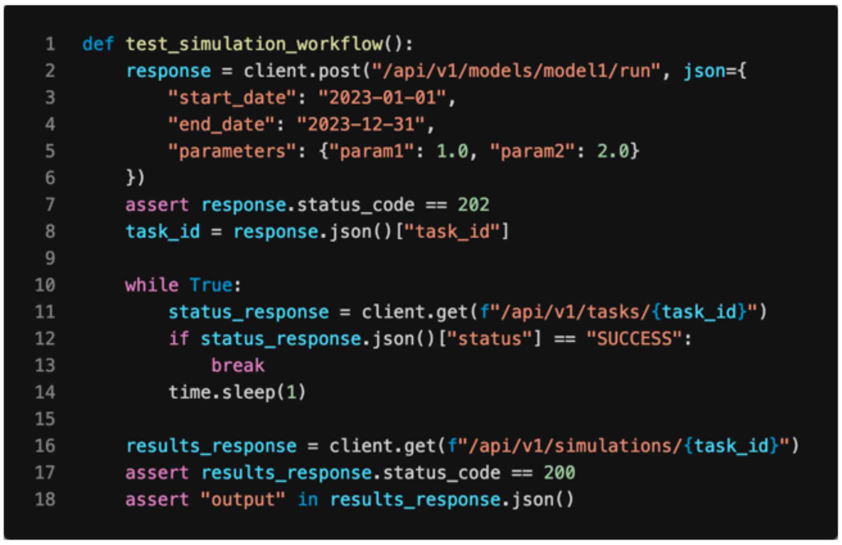
Functional testing verifies that the API behaves correctly and produces expected results. This includes unit tests for individual components and integration tests for system-wide functionality. Unit tests focus on validating specific functions or classes in isolation. For instance, a unit test might verify the correct parsing of simulation input parameters (Figure 11).



```python
def test_simulation_input_validation():
    input_data = SimulationInput(start_date="2023-01-01", end_date="2023-12-31",
                                 parameters={"param1": 1.0, "param2": 2.0})
    assert input_data.start_date < input_data.end_date
    assert "param1" in input_data.parameters
```

Figure 11. An example of a unit test to validate a function is working as intended.

Integration tests, on the other hand, ensure that different components of the system interact correctly. These tests typically cover entire workflows, from initiating a simulation to retrieving results (Figure 12).

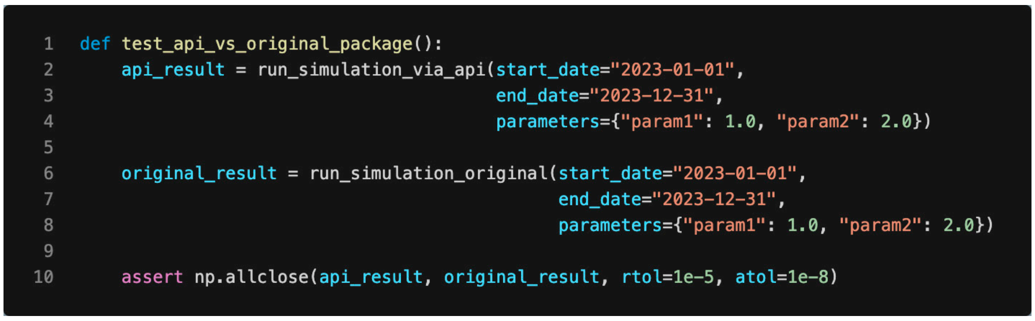


```python
def test_simulation_workflow():
    response = client.post("/api/v1/models/model1/run", json={
        "start_date": "2023-01-01",
        "end_date": "2023-12-31",
        "parameters": {"param1": 1.0, "param2": 2.0}
    })
    assert response.status_code == 202
    task_id = response.json()["task_id"]

    while True:
        status_response = client.get(f"/api/v1/tasks/{task_id}")
        if status_response.json()["status"] == "SUCCESS":
            break
        time.sleep(1)

    results_response = client.get(f"/api/v1/simulations/{task_id}")
    assert results_response.status_code == 200
    assert "output" in results_response.json()
```

Figure 12. An example of an integration test running the workflow from start to end.

Performance testing is essential to ensure the API can handle expected loads and respond within acceptable timeframes. This includes load testing, which simulates multiple concurrent users, and response time testing for various types of requests. Tools such as Locust or Apache JMeter are employed for these tests.

A critical aspect of the testing phase was the validation of the web API against the original Python package. This ensured that the conversion process did not introduce errors or inconsistencies. The validation tests compared the results obtained from the web API with those from the original package (Figure 13).



```python
def test_api_vs_original_package():
    api_result = run_simulation_via_api(start_date="2023-01-01",
                                        end_date="2023-12-31",
                                        parameters={"param1": 1.0, "param2": 2.0})

    original_result = run_simulation_original(start_date="2023-01-01",
                                              end_date="2023-12-31",
                                              parameters={"param1": 1.0, "param2": 2.0})

    assert np.allclose(api_result, original_result, rtol=1e-5, atol=1e-8)
```

Figure 13. An example validation test to compare the original workflow against the new functions.

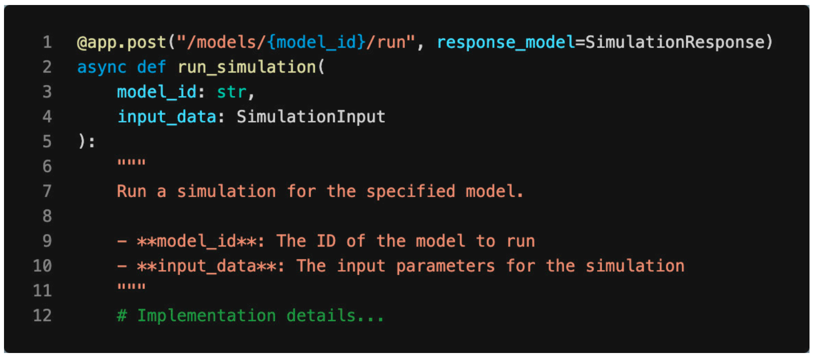
These validation tests ensured that the results from the web API match those from the original Python package within an acceptable margin of error, typically using numpy's allclose function for numerical comparisons.

The testing and validation phase was iterative, with tests run continuously throughout the development process. This approach allows for early detection and correction of issues, ensuring the final web API is robust, reliable, and faithful to the original Python package's functionality.

2.2.7. Documentation Generation

The final phase of the workflow implementation leverages FastAPI's built-in documentation capabilities to generate comprehensive API documentation. FastAPI automatically creates interactive API documentation using the OpenAPI (formerly Swagger) and ReDoc standards.

This documentation is generated from the Python code and type hints, requiring minimal additional effort (Figure 14).



```python
@app.post("/models/{model_id}/run", response_model=SimulationResponse)
async def run_simulation(
    model_id: str,
    input_data: SimulationInput
):
    """
    Run a simulation for the specified model.

    - **model_id**: The ID of the model to run
    - **input_data**: The input parameters for the simulation
    """
    # Implementation details...
```

Figure 14. An example of documentation strings within a function.

In the code snippet, FastAPI uses the function's docstring, parameter types, and response model to generate detailed API documentation. The resulting documentation includes endpoint descriptions, request/response schemas, and example usage, ensuring that it remains synchronized with the actual API implementation.

The generated documentation provides a user-friendly interface for exploring and testing the API, facilitating easier adoption and usage of the converted hydrological models.

3. Results

The workflow for converting hydrology Python packages into web APIs was applied to two machine learning-based models: a GRACE downscaling model [35] and a synthetic groundwater time series generation model [36]. This section presents the outcomes of implementing the workflow, focusing on the resulting user interfaces, deployment processes, and practical applications. Additionally, a second case study involving the conversion of a simple MODFLOW model into a web API is discussed as well, demonstrating the workflow's versatility across different types of models.

3.1. Case Study 1: Web APIs for GRACE Downscaling and Synthetic Time Series Models

The workflow was successfully applied to two machine learning-based models: a GRACE downscaling model and a synthetic hydrological time series generation model. Both models were converted into web APIs with minimal modifications to their core functionalities, demonstrating the workflow's adaptability to different types of machine learning models in hydrology.

3.1.1. Implementation and Deployment

The deployment process, following the workflow's containerization guidelines, proved efficient and straightforward:

1. Both machine learning models were containerized using a single Dockerfile, simplifying the deployment process.

2. With a reliable internet connection, the entire process of provisioning an instance and deploying the models was accomplished in less than 15 min.
3. The rapid deployment showcases the workflow's efficiency in making complex machine learning-based hydrological models accessible as web services.

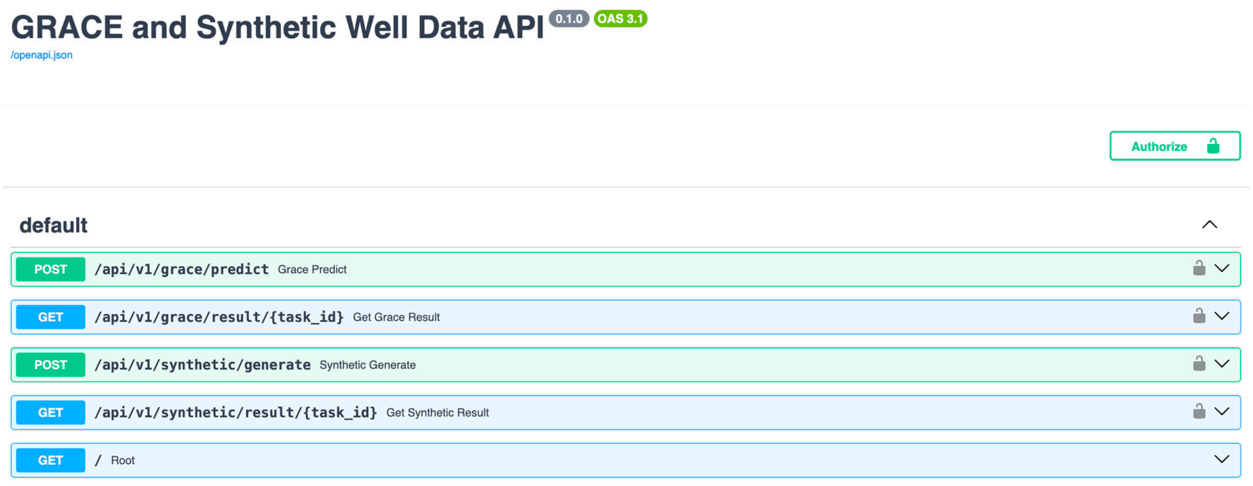
This quick turnaround from local development to deployed web service demonstrates the workflow's potential to significantly reduce the time and effort required to make hydrological models operational.

3.1.2. User Interface and Functionality

The resulting web interface, generated automatically by FastAPI, provides intuitive access to both machine learning models. Key features include the following:

- Interactive API documentation with OpenAPI (Swagger) UI.
- Clear separation of endpoints for each model.
- User-friendly forms for input parameter submission.

Figure 15 shows the landing page of the API documentation. Users can easily input parameters, upload necessary files, and receive results through this interface.



GRACE and Synthetic Well Data API 0.1.0 OAS 3.1

/openapi.json

Authorize

default

POST /api/v1/grace/predict Grace Predict

GET /api/v1/grace/result/{task_id} Get Grace Result

POST /api/v1/synthetic/generate Synthetic Generate

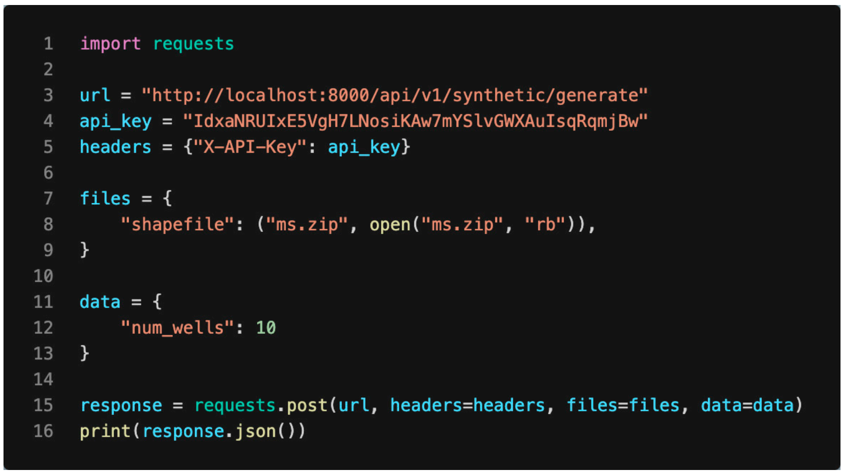
GET /api/v1/synthetic/result/{task_id} Get Synthetic Result

GET / Root

Figure 15. Screenshot of the API documentation landing page.

3.1.3. API Integration Examples

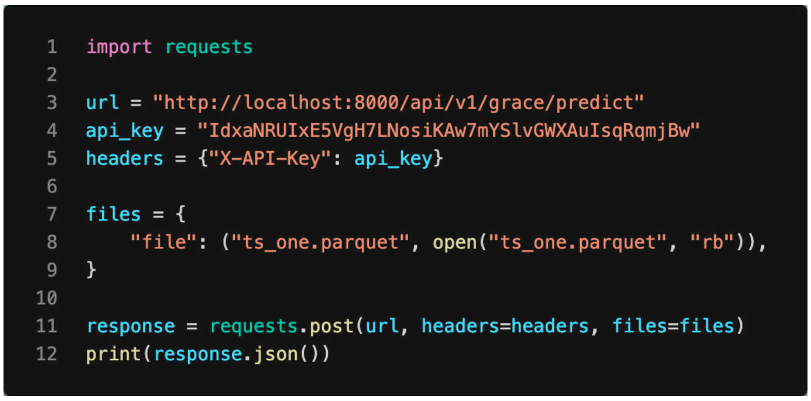
Figure 16 demonstrates how to interact with the deployed machine learning model APIs.



```python
import requests

url = "http://localhost:8000/api/v1/synthetic/generate"
api_key = "IdxaNRUIxE5VgH7LNosiKAw7mYSlvGWXAuIsqRqmjBw"
headers = {"X-API-Key": api_key}

files = {
    "shapefile": ("ms.zip", open("ms.zip", "rb")),
}

data = {
    "num_wells": 10
}

response = requests.post(url, headers=headers, files=files, data=data)
print(response.json())
```

Figure 16. A sample Python script to query time series using the synthetic time series model.

Figure 17 demonstrates how to generate synthetic well data using a machine learning model with a provided shapefile.



```python
import requests

url = "http://localhost:8000/api/v1/grace/predict"
api_key = "IdxaNRUIxE5VgH7LNosiKAw7mYSlvGWXAuIsqRqmjBw"
headers = {"X-API-Key": api_key}

files = {
    "file": ("ts_one.parquet", open("ts_one.parquet", "rb")),
}

response = requests.post(url, headers=headers, files=files)
print(response.json())
```

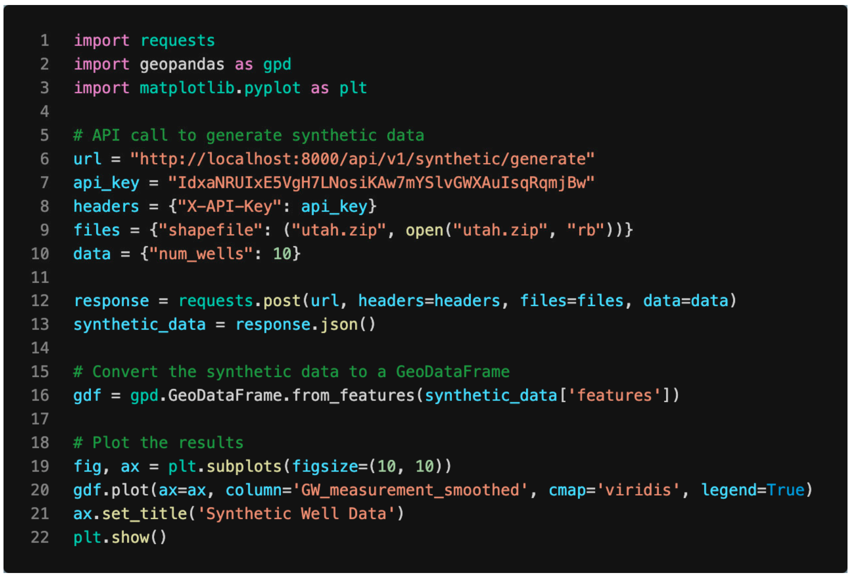
Figure 17. A sample Python script to generate downscaled GRACE values.

The script shows how to make predictions using the GRACE downscaling machine learning model with a provided parquet file.

3.1.4. Integration with Developer Applications

The web APIs created through this workflow provide a flexible foundation for developers to incorporate advanced hydrological modeling capabilities into their own applications. Developers can leverage the API endpoints to retrieve model predictions, generate synthetic data, and build web or mobile applications that offer on-demand access to hydrological insights.

For example, a developer could create a web application that allows users to upload shapefiles and generate synthetic well data, then visualize the results on an interactive map. Here is a basic example of how this might be implemented using Python and a plotting library (Figure 18).



```python
import requests
import geopandas as gpd
import matplotlib.pyplot as plt

# API call to generate synthetic data
url = "http://localhost:8000/api/v1/synthetic/generate"
api_key = "IdxaNRUIxE5VgH7LNosiKAw7mYSlvGWXAuIsqRqmjBw"
headers = {"X-API-Key": api_key}
files = {"shapefile": ("utah.zip", open("utah.zip", "rb"))}
data = {"num_wells": 10}

response = requests.post(url, headers=headers, files=files, data=data)
synthetic_data = response.json()

# Convert the synthetic data to a GeoDataFrame
gdf = gpd.GeoDataFrame.from_features(synthetic_data['features'])

# Plot the results
fig, ax = plt.subplots(figsize=(10, 10))
gdf.plot(ax=ax, column='GW_measurement_smoothed', cmap='viridis', legend=True)
ax.set_title('Synthetic Well Data')
plt.show()
```

Figure 18. Sample code for generating plots from synthetic time series model results.

This example demonstrates how developers can easily incorporate the API's functionality into their own data processing and visualization workflows. By providing these APIs, the workflow enables developers to create custom applications that leverage sophisticated hydrological models without needing to implement the underlying machine learning algorithms themselves.

For a more comprehensive view of the implementation and additional examples, readers are encouraged to visit the GitHub repository at https://github.com/IGWM/streamML (accessed on 13 September 2024).

3.2. *Case Study 2: Web API for MODFLOW Groundwater Model*

The workflow was successfully applied to convert a simple MODFLOW groundwater model into a web API, demonstrating its adaptability to physics-based numerical models alongside machine learning applications.

3.2.1. Implementation and Deployment

The deployment process for the MODFLOW model was efficient:

1. The MODFLOW model was containerized using a single Dockerfile, with FloPy [37,38] handling interactions with MODFLOW, streamlining the setup process.
2. Deployment time was comparable to the machine learning models in Case Study 1, taking approximately 15 min.
3. This rapid deployment showcases the workflow's effectiveness in making complex numerical hydrological models accessible as web services.

One challenge encountered during implementation was ensuring proper error handling for various MODFLOW input scenarios, which was addressed through comprehensive input validation in the API layer.

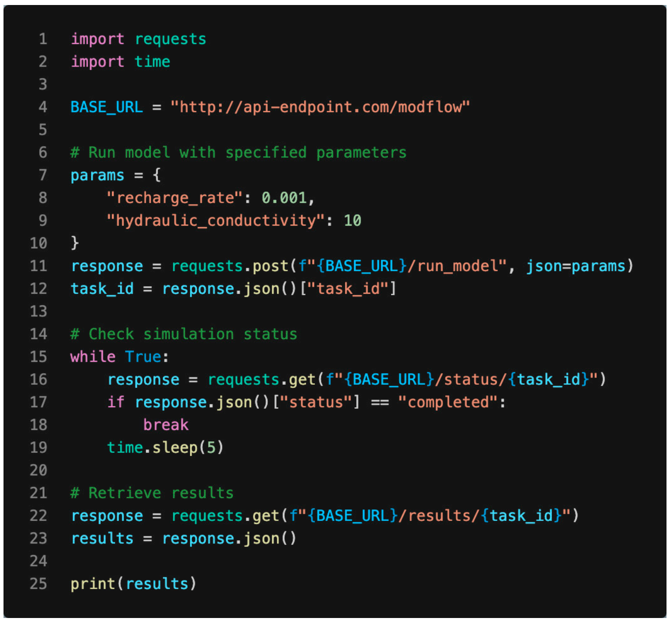
3.2.2. API Structure and Functionality

The MODFLOW API provides a streamlined interface for groundwater simulations:

- A run_model endpoint that accepts parameters such as recharge rate and hydraulic conductivity.
- An endpoint to track the progress of model runs.
- A results retrieval endpoint for accessing simulation outputs.

The API documentation landing page is like that from Case Study 1, providing clear, interactive documentation for all endpoints.

Figure 19 demonstrates the interaction with the MODFLOW model API using a sample Python code snippet. It includes running a MODFLOW simulation and retrieving results through the API.



```python
import requests
import time

BASE_URL = "http://api-endpoint.com/modflow"

# Run model with specified parameters
params = {
    "recharge_rate": 0.001,
    "hydraulic_conductivity": 10
}
response = requests.post(f"{BASE_URL}/run_model", json=params)
task_id = response.json()["task_id"]

# Check simulation status
while True:
    response = requests.get(f"{BASE_URL}/status/{task_id}")
    if response.json()["status"] == "completed":
        break
    time.sleep(5)

# Retrieve results
response = requests.get(f"{BASE_URL}/results/{task_id}")
results = response.json()

print(results)
```

Figure 19. Sample code for running mudflow simulation and retrieving results.

3.2.3. Integration with Developer Applications

The MODFLOW API provides a flexible foundation for developers to incorporate groundwater modeling capabilities into their own applications. Developers can leverage the API endpoints to run simulations, track progress, and retrieve results, enabling the creation of web or mobile applications that offer on-demand access to groundwater insights.

For example, developers could create applications that allow users to explore the impacts of different recharge rates and hydraulic conductivity values on groundwater levels. The API's ability to run MODFLOW executables on-demand through requests significantly reduces the barrier to entry for using MODFLOW in diverse applications.

This case study represents an approach to deploying MODFLOW models as APIs, expanding the possibilities for integrating sophisticated groundwater modeling into web-based applications. The workflow enables developers to create custom applications that leverage complex groundwater models without needing to implement MODFLOW directly.

For more details on implementation and additional examples, readers are encouraged to visit the GitHub repository at https://github.com/IGWM/modflow-web (accessed on 13 September 2024).

4. Discussion

The framework presented in this paper offers a significant advancement in the accessibility and deployment of complex hydrological models. By transforming Python-based models into web APIs, it addresses a critical need in the hydrological modeling community for more interoperable and readily deployable tools. While previous approaches such as Tethys, HydroDS, and HydroShare have made strides in improving model accessibility, they often involve complex setups or primarily focus on data sharing rather than model execution. This framework builds upon these efforts by providing a more streamlined approach to model deployment, requiring minimal modifications to existing Python-based models and leveraging containerization for consistent deployment across environments.

The successful application of this framework to both machine learning-based models (GRACE downscaling and synthetic time series generation) and physics-based models (MODFLOW) demonstrates its versatility across diverse hydrological applications. The rapid deployment achieved in the case studies, with models converted to web APIs in less than 15 min, represents a substantial improvement in the operational efficiency of sophisticated hydrological tools. This efficiency is particularly noteworthy given the complexity typically associated with deploying such models.

The resulting standardized RESTful API interface aligns hydrological modeling with contemporary software development practices, opening new avenues for integration and innovation. This alignment enables developers to incorporate advanced hydrological models into a wide array of applications, from web-based visualization tools to decision support systems. For instance, the GRACE downscaling model API could be leveraged to create dynamic mapping applications that visualize groundwater changes over time, while the synthetic time series generator could be integrated into risk assessment tools for water resource management.

By providing a consistent interface for model execution, the framework allows developers to create modular, flexible applications that can easily switch between different hydrological models or combine outputs from multiple models. This interoperability could lead to more comprehensive water management solutions that integrate surface water, groundwater, and climate models seamlessly. The ability to run model executables on-demand through API requests, as demonstrated in the MODFLOW case study, represents a particularly significant advancement in making complex models more accessible and operationally efficient.

The ease of deployment and standardized access also opens possibilities for educational applications, allowing students and researchers to interact with complex models through user-friendly interfaces without the need for extensive setup or computational resources. This democratization of access to sophisticated hydrological tools has the potential to accelerate learning and innovation in the field.

The integration of these web APIs enables comprehensive water assessment workflows. For instance, in a region of interest, the GRACE downscaling API could provide high-resolution groundwater storage change estimates. The synthetic well time series API could generate data for testing interpolation algorithms for groundwater storage maps. These outputs could then feed into the MODFLOW API for groundwater simulations. As web APIs, these tools could be combined in a single web application, allowing stakeholders to access the entire workflow through a unified interface. This approach streamlines complex hydrological processes, making sophisticated tools more accessible for water resource applications.

While the initial Docker configuration may require some trial and error, especially for models with external files or dependencies, the framework provides a comprehensive blueprint that significantly streamlines this process. This approach reduces barriers to model deployment and usage, potentially broadening the user base for sophisticated hydrological tools and accelerating both research and practical applications in water resource management.

Transitioning from a development to a production environment involves addressing several critical aspects. Scalability can be achieved through container orchestration platforms like Kubernetes or commercial cloud-based solutions. Kubernetes offers benefits such as flexible scaling and efficient resource utilization but adds complexity to deployment and maintenance. Cloud-based solutions may simplify management but can come with reduced control and potentially higher costs.

Performance optimization is crucial for large-scale simulations and real-time applications. Distributed computing frameworks like Dask or Ray enable parallel processing with minimal overhead, while GPU acceleration can enhance performance for computationally intensive tasks. Implementing caching mechanisms and optimizing database queries are essential for maintaining real-time responsiveness.

Security is a major concern due to the sensitive nature of hydrologic data. While FastAPI has some built-in tools to help with authentication, integrating with additional tools like OAuth2 or OpenID Connect is recommended for robust access control. Additionally, tools from the FastAPI community, such as fastapi-limiter, can help with rate limiting and setting resource quotas to prevent API abuse and manage computational resources effectively. Addressing these concerns is essential for deploying scalable, high-performance, and secure hydrological modeling APIs in production.

In conclusion, this framework for converting hydrological models to web APIs represents a significant step towards more accessible, flexible, and integrated hydrological modeling. By bridging the gap between complex models and modern web technologies, it not only enhances research and practical applications in hydrology and water resource management but also opens new possibilities for the creative and innovative use of these models across various domains. The potential for developers to build upon these APIs and create novel applications underscores the transformative impact this framework could have on the field of hydrology and beyond.

Author Contributions: Conceptualization, H.Y. and S.T.P.; methodology, S.T.P.; software, S.T.P.; validation, S.T.P. and L.D.Y.; formal analysis, S.T.P.; data curation, S.T.P.; writing—original draft preparation, S.T.P.; writing—review and editing, H.Y. and L.D.Y.; project administration, H.Y. and L.D.Y.; funding acquisition, H.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was funded by a research grant awarded by the National Science Foundation (Award No.: OIA 2019561).

Data Availability Statement: Data are contained within the article.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Zehe, E.; Sivapalan, M. Towards a New Generation of Hydrological Process Models for the Meso-Scale: An Introduction. *Hydrol. Earth Syst. Sci.* **2007**, *10*, 981–996. [CrossRef]
2. Asgari, M.; Yang, W.; Lindsay, J.; Tolson, B.; Dehnavi, M.M. A Review of Parallel Computing Applications in Calibrating Watershed Hydrologic Models. *Environ. Model. Softw.* **2022**, *151*, 105370. [CrossRef]
3. Singh, V.P. Hydrologic Modeling: Progress and Future Directions. *Geosci. Lett.* **2018**, *5*, 15. [CrossRef]
4. SWAT: Model Use, Calibration, and Validation. Available online: https://digitalcommons.unl.edu/biosysengfacpub/406/ (accessed on 8 July 2024).
5. Singh, J.; Knapp, H.V.; Arnold, J.G.; Demissie, M. Hydrological Modeling of the Iroquois River Watershed Using Hspf and Swat1. *JAWRA J. Am. Water Resour. Assoc.* **2005**, *41*, 343–360. [CrossRef]
6. Harbaugh, A.W.; McDonald, M.G. *User's Documentation for MODFLOW-96, an Update to the U.S. Geological Survey Modular Finite-Difference Ground-Water Flow Model*; U.S. Geological Survey; Branch of Information Services: Reston, VA, USA, 1996.
7. Thakur, J.K.; Singh, S.K.; Ekanthalu, V.S. Integrating Remote Sensing, Geographic Information Systems and Global Positioning System Techniques with Hydrological Modeling. *Appl. Water Sci.* **2017**, *7*, 1595–1608. [CrossRef]
8. Lee, S.; Hyun, Y.; Lee, S.; Lee, M.-J. Groundwater Potential Mapping Using Remote Sensing and GIS-Based Machine Learning Techniques. *Remote Sens.* **2020**, *12*, 1200. [CrossRef]
9. Ashraf, A.; Ahmad, Z.; Ashraf, A.; Ahmad, Z. Integration of Groundwater Flow Modeling and GIS. In *Water Resources Management and Modeling*; IntechOpen: London, UK, 2012; ISBN 978-953-51-0246-5.
10. Xu, X.; Li, J.; Tolson, B.A. Progress in Integrating Remote Sensing Data and Hydrologic Modeling. *Prog. Phys. Geogr. Earth Environ.* **2014**, *38*, 464–498. [CrossRef]
11. Munawar, H.S.; Hammad, A.W.A.; Waller, S.T. Remote Sensing Methods for Flood Prediction: A Review. *Sensors* **2022**, *22*, 960. [CrossRef]
12. Choi, M.; Jacobs, J.M.; Anderson, M.C.; Bosch, D.D. Evaluation of Drought Indices via Remotely Sensed Data with Hydrological Variables. *J. Hydrol.* **2013**, *476*, 265–273. [CrossRef]
13. Nobre, R.C.M.; Rotunno Filho, O.C.; Mansur, W.J.; Nobre, M.M.M.; Cosenza, C.A.N. Groundwater Vulnerability and Risk Mapping Using GIS, Modeling and a Fuzzy Logic Tool. *J. Contam. Hydrol.* **2007**, *94*, 277–292. [CrossRef]
14. Hussein, E.A.; Thron, C.; Ghaziasgar, M.; Bagula, A.; Vaccari, M. Groundwater Prediction Using Machine-Learning Tools. *Algorithms* **2020**, *13*, 300. [CrossRef]
15. Mosavi, A.; Ozturk, P.; Chau, K. Flood Prediction Using Machine Learning Models: Literature Review. *Water* **2018**, *10*, 1536. [CrossRef]
16. Anaraki, M.V.; Farzin, S.; Mousavi, S.-F.; Karami, H. Uncertainty Analysis of Climate Change Impacts on Flood Frequency by Using Hybrid Machine Learning Methods. *Water Resour. Manag.* **2021**, *35*, 199–223. [CrossRef]
17. Alizamir, M.; Kisi, O.; Zounemat-Kermani, M. Modelling Long-Term Groundwater Fluctuations by Extreme Learning Machine Using Hydro-Climatic Data. *Hydrol. Sci. J.* **2018**, *63*, 63–73. [CrossRef]
18. Swain, N.R.; Christensen, S.D.; Snow, A.D.; Dolder, H.; Espinoza-Dávalos, G.; Goharian, E.; Jones, N.L.; Nelson, E.J.; Ames, D.P.; Burian, S.J. A New Open Source Platform for Lowering the Barrier for Environmental Web App Development. *Environ. Model. Softw.* **2016**, *85*, 11–26. [CrossRef]
19. Rocklin, M. Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In Proceedings of the Python in Science Conferences (SciPy 2015), Austin, TX, USA, 6–12 July 2015; pp. 126–132.
20. Erickson, R.A.; Fienen, M.N.; McCalla, S.G.; Weiser, E.L.; Bower, M.L.; Knudson, J.M.; Thain, G. Wrangling Distributed Computing for High-Throughput Environmental Science: An Introduction to HTCondor. *PLoS Comput. Biol.* **2018**, *14*, e1006468. [CrossRef]
21. Christensen, S.D.; Swain, N.R.; Jones, N.L.; Nelson, E.J.; Snow, A.D.; Dolder, H.G. A Comprehensive Python Toolkit for Accessing High-Throughput Computing to Support Large Hydrologic Modeling Tasks. *JAWRA J. Am. Water Resour. Assoc.* **2017**, *53*, 333–343. [CrossRef]
22. Horsburgh, J.S.; Morsy, M.M.; Castronova, A.M.; Goodall, J.L.; Gan, T.; Yi, H.; Stealey, M.J.; Tarboton, D.G. HydroShare: Sharing Diverse Environmental Data Types and Models as Social Objects with Application to the Hydrology Domain. *JAWRA J. Am. Water Resour. Assoc.* **2016**, *52*, 873–889. [CrossRef]
23. Horsburgh, J.; Tarboton, D.; Schreuders, K.; Maidment, D.; Zaslavsky, I.; Valentine, D. Hydroserver: A Platform for Publishing Space-Time Hydrologic Datasets. In Proceedings of the AWRA 2010 Spring Specialty Conference: GIS and Water Resources VI, Orlando, FL, USA, 29–31 March 2010; pp. 1–6.
24. Erazo Ramirez, C.; Sermet, Y.; Molkenthin, F.; Demir, I. HydroLang: An Open-Source Web-Based Programming Framework for Hydrological Sciences. *Environ. Model. Softw.* **2022**, *157*, 105525. [CrossRef]
25. Gichamo, T.Z.; Sazib, N.S.; Tarboton, D.G.; Dash, P. HydroDS: Data Services in Support of Physically Based, Distributed Hydrological Models. *Environ. Model. Softw.* **2020**, *125*, 104623. [CrossRef]

26. Ramirez, C.E.; Sermet, Y.; Demir, I. HydroCompute: An Open-Source Web-Based Computational Library for Hydrology and Environmental Sciences. *Environ. Model. Softw.* **2024**, *175*, 106005. [CrossRef]
27. de Sousa, L.M.; de Jesus, J.M.; Čepicky, J.; Kralidis, A.T.; Huard, D.; Ehbrecht, C.; Barreto, S.; Eberle, J. PyWPS: Overview, New Features in Version 4 and Existing Implementations. *Open Geospat. Data Softw. Stand.* **2019**, *4*, 13. [CrossRef]
28. FastAPI. Available online: https://fastapi.tiangolo.com/ (accessed on 21 June 2024).
29. Boettiger, C. An Introduction to Docker for Reproducible Research. *SIGOPS Oper. Syst. Rev.* **2015**, *49*, 71–79. [CrossRef]
30. Ibrahim, M.H.; Sayagh, M.; Hassan, A.E. A Study of How Docker Compose Is Used to Compose Multi-Component Systems. *Empir. Softw. Eng.* **2021**, *26*, 128. [CrossRef]
31. Skorpil, V.; Oujezsky, V. Parallel Genetic Algorithms' Implementation Using a Scalable Concurrent Operation in Python. *Sensors* **2022**, *22*, 2389. [CrossRef]
32. Williams, J. *RabbitMQ in Action: Distributed Messaging for Everyone*; Simon and Schuster: New York City, NY, USA, 2012; ISBN 978-1-63835-384-3.
33. Silva, M.D.D.; Tavares, H.L. *Redis Essentials*; Packt Publishing Ltd.: Birmingham, UK, 2015; ISBN 978-1-78439-608-4.
34. Principled Design of the Modern Web Architecture | ACM Transactions on Internet Technology. Available online: https://dl.acm.org/doi/abs/10.1145/514183.514185 (accessed on 9 July 2024).
35. Pulla, S.T.; Yasarer, H.; Yarbrough, L.D. GRACE Downscaler: A Framework to Develop and Evaluate Downscaling Models for GRACE. *Remote Sens.* **2023**, *15*, 2247. [CrossRef]
36. Pulla, S.T.; Yasarer, H.; Yarbrough, L.D. Synthetic Time Series Data in Groundwater Analytics: Challenges, Insights, and Applications. *Water* **2024**, *16*, 949. [CrossRef]
37. FloPy Workflows for Creating Structured and Unstructured MODFLOW Models—Hughes—2024—Groundwater—Wiley Online Library. Available online: https://ngwa.onlinelibrary.wiley.com/doi/10.1111/gwat.13327 (accessed on 9 July 2024).
38. Bakker, M.; Post, V.; Langevin, C.D.; Hughes, J.D.; White, J.T.; Starn, J.J.; Fienen, M.N. Scripting MODFLOW Model Development Using Python and FloPy. *Groundwater* **2016**, *54*, 733–739. [CrossRef]