# Dagger.io for defining generic pipelines

Nico Guldin, Göktug Özdemir, and Jasper Fülle

University of Cologne, Albertus-Magnus-Platz, 50923 Cologne, Germany

**Abstract.** Dagger.io, a new CI/CD tool created by Docker's founder Solomon Hykes, introduces a unique approach to Software deployment and automation by enabling portability with the use of container technology. This paper delves into the theoretical foundations in the CI/CD landscape, detailing their fundamental concepts and widely-used tools. We present a guide on installing and executing a Dagger.io Pipeline, illustrated with our own example. Furthermore, we explore the Integration of Dagger with tools such as GitHub Actions, GitLab CI and Jenkins, highlighting its compatibility in diverse environments. The evaluation of Dagger ends in a critical discussion of its features, benefits and potential for the feature of the CI/CD process. This study aims to provide valuable insights into Dagger.io's position within the broader context of CI/CD technologies.

**Keywords:** Dagger.io · CI/CD Pipeline · DevOps

## 1 Introduction

In the past, development environmets for software engineering projects required manual steps when writing, building and deploying code. These tasks were extremely time consuming for both developers and operations teams, as they were responsible for code testing and code releases. The introduction of automation in the code deployment process with deployment pipelines allowed development teams to focus on innovating and improving the end product rather than waste time with time consuming manual tasks. Deployment pipelines are systems of automated processes to quickly move code additions and updates from version control to production [1].

### 1.1 Problem definition and objective

Software developers use different tools to implement these so called CI/CD pipelines. But using different CI/CD tools can become problematic. Assume a software was developed using GitLab as the version control platform and the corresponding GitLab CI/CD tool for defining and running pipelines. If the development team decides to release the software as open source on GitHub the existing GitLab CI/CD tool will no longer be viable. Consequently, the pipeline would need to be migrated to an alternative, such as GitHub Actions. The challenge lies in migrating these pipelines between different CI/CD tools with the least amount of effort. In this work, we will analyze ways to define generic pipelines that can easily be migrated between different CI/CD environments. To do so, we will take a close look at Dagger.io, a tool that aims to solve the aforementioned problem.

### 1.2 Methodology and structure

To tackle this problem, the following methodology was used. At first, a literature review about Dagger was to be conducted. But we quickly realized, that Dagger was a relatively new tool, so we had to resort to the official Dagger documentation. The second part of our methodology is prototyping an example Dagger pipeline to test, whether it delivers on its promises. But before we take a closer look at Dagger, one has to first analyze the state of the art in CI/CD practices and theoretical foundations of CI/CD. Then, we will dive deeper into selected CI/CD tools, mainly GitLab CI, GitHub Actions and Jenkins. We will look at the differences and similarities between these three tools. At the end of the second chapter, we will analyze the challenges in pipeline migration. In the third chapter, we will explain the functionalities of Dagger in detail. We will then give a small tutorial on how to implement a pipeline in Dagger. In the forth chapter, we will examine how to integrate Dagger into three different CI/CD tools and talk about the pipeline prototype we created. In the fifth chapter, we will give a critical assessment of Dagger. Advantages and Disadvantages of using Dagger will be discussed. The work finishes with a conclusion.

## 2 State of the Art in CI/CD Practices

### 2.1 Theoretical foundations of CI/CD

Continuous Integration (CI) and Continuous delivery/deployment (CD) are software development industry practices that enable a more efficient transition from development to production. These practices are primarily characterized by their reliance on the automation of the delivery pipeline. Advantages of embracing CI/CD include improved code quality, faster feedback loops, and a reduced time-to-market for software products [3]. CI/CD pipelines are customized with various steps and configurations to meet the unique demands of different projects.
**Continuous Integration (CI)** centers on a source control management system (SCM), which serves as a repository for application code. The fundamental process of CI involves the automation of building and testing code upon each commit or update to the repository [2]. This includes not only compiling of the code but also running a variety of automated tests to ensure the code functionality.
**Continuous Delivery/Deployment (CD)** differentiate in the deployment process: Continuous Delivery typically involves manual steps for deploying into production upon a series of manual quality assurance testing, whereas Continuous Deployment automates the deployment process entirely. The choice between

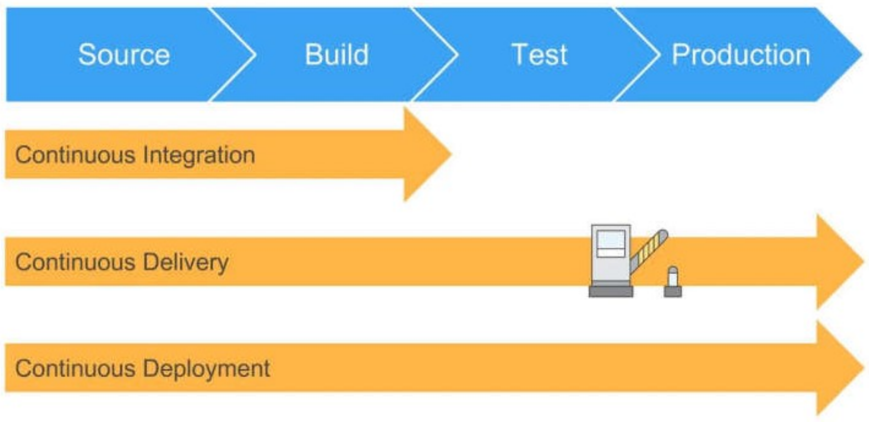


**Fig. 1.** Simplified diagram of CI/CD. Reference: [2]

these approaches depend on the specific requirements and preferences of the project.
Key activities in CD include automated testing in a production-like setting to ensure that the application also meets the required non-functional requirements like performance, security and resource usage. This ensures that any code changes are immediately ready for release, significantly enhancing the efficiency and frequency of deployments while maintaining software quality [3].

## 2.2 Comparative Analysis of CI/CD Tools

The landscape of CI/CD tools are offering a range of solutions for different pipeline requirements. According to a survey conducted by JetBrains the most used tools in the industry include GitHub Action, GitLab CI/CD and Jenkins [4]. This analysis will therefor focus on these three widely-used tools to understand their unique features and capabilities.

**GitHub Actions**
GitHub Actions enables developers to initiate workflows in response to specific events within GitHub, such as pushing code to a branch or opening a pull request. This integration means that its usage is limited to source code hosted in GitHub repositories. GitHub offers free usage of GitHub Actions for public repositories, while private repositories are provided with a number of hosted runner minutes depending on your pricing plan. Additionally hosted runner minutes are available for purchase.
A GitHub Actions workflow is structured in various components. The structure of a GitHub Actions pipeline is presented in Figure 3. A workflow is initiated by an *event* within your repository, which triggers the execution of the workflow. Each workflow has one or more *jobs*, which are sequences of *steps*. These steps

can execute scripts defined by the user or utilize an *action*, a reusable component designed to streamline the workflow. Each job operates within a *runner*, which is a dedicated virtual machine or container where the job's tasks are carried out. A repository is able to have multiple workflows, each can perform a different set of tasks, such as building and testing a pull request, deploy your application every time a release is created or adding a label every time someone opens a new issue. GitHub Actions also provides the functionallity to reuse workflows from another workflow to avoid duplication. Initially jobs run in parallel with each other, but you can configure dependencies such that it will wait for the dependent job to complete, before it runs. Workflows are defined in the `.github/workflows` directory and are defied and configurable by a YAML file. One of the standout features offered by GitHub is the ability to use over 21.000 actions created by the GitHub community [5]. This aims to simplify creating complex workflows by providing pre-built solutions, thereby reducing the time spent to writing repetitive, foundational code. In Listing 1.11, an example of a GitHub Actions pipeline for a Node.js project is presented.

**GitLab CI/CD**
GitLab CI/CD shares several similarities with GitHub Actions. The pipeline configuration is also defined in a YAML file located within the repository, and at the free tier the usage of GitLab CI/CD is exclusively for repositories hosted on GitLab [6]. While there are similarities, the syntax of the YAML files differs between GitHub Actions and GitLab CI/CD. A distinctive feature of GitLab CI/CD is its CI Lint tool, which validates the YAML file before committing. This preemptive check helps to prevent potential misconfigurations. Moreover, GitLab CI/CD employs a structure similar to GitHub Actions for defining pipelines but introduces the concept of *stages*. These stages organize jobs into sequential phases, allowing for more control of the pipeline execution flow. GitLab is widely recognized for its comprehensive DevOps approach, containing not only CI/CD pipelines but also project management and continuous monitoring. Key features include Auto DevOps, which automates pipeline configuration, streamlining the development process. Additionally, GitLab's robust support for containerization and Kubernetes enhances efficient management of cloud-native applications, highlighting its capability in modern software development ecosystems. An exemplary GitLab CI/CD pipeline of a Node.js project is preseted at Listing 1.12 in the Appendix.

**Jenkins**
Jenkins is a Java-based open-source automation server, which enable pipeline execution in a platform-independent manner [7]. Unlike GitHub Actions and GitLab CI/CD, Jenkins does not rely on specific repository hosting platforms, offering greater flexibility. Jenkins allows pipeline definition in two ways: through a `Jenkinsfile` or via the Blue Ocean interface. The `Jenkinsfile` serves as a configuration file, comparable to the YAML files used in GitHub Actions and GitLab CI/CD. Upon installation, Jenkins provides a user interface where these configuration files can be accessed and edited. This interface also supports var-

ious administrative tasks, such as user role management. Blue Ocean, a plugin for Jenkins, aims to enhance the user experience. It offers a visual representation of pipelines, a guided process for creating and editing pipelines via its pipeline editor, and seamless integration with branches and pull requests from GitHub or BitBucket [8]. Jenkins' popularity is enhanced by its community support. The platform's extensibility is provided in its library of over 1800 community-created plugins, which add a wide range of features, reducing the need for custom coding [9]. However, setting up a Jenkins pipeline in a project can require more effort compared to GitHub Actions or GitLab CI/CD. It is essential to establish a server where Jenkins will be installed, configure the build environment, and integrate the version control system. These steps, which are integral to a Jenkins setup, are not necessary with GitHub Actions or GitLab CI/CD, as they provide more out-of-the-box solutions with their cloud-based platforms. Listing 1.13 showcases a Jenkinsfile configured for a Node.js project.

## 2.3 Challenges in Pipeline Migration

The process of migrating CI/CD pipelines between different tools or platforms can be challenge due to the distinct functionalities, configuration setups, and foundational principles of each tool. Nevertheless a migration can be necessary for example while switching the SCM and therefore it is not able to use GitHub Actions or GitLab CI/CD anymore. Understanding these challenges is crucial for a smooth and effective pipeline management.

- **Differences in Configuration Syntax:** Each CI/CD tool, whether it is Jenkins, GitLab CI/CD or GitHub Actions, has its own unique configuration syntax and file format, typically defined in YAML or similar markup languages. For migrating a pipeline, this means translating its configuration to a new, potentially unfamiliar format. This task is particularly demanding for complex pipelines that include numerous steps or stages.
- **Compatibility of Plugins and Extensions:** Tools like Jenkins heavily rely on plugins, whereas GitHub Actions and GitLab CI/CD utilize extensions or pre-built actions. Finding equivalent functionalities in the new platform can be a challenge, as direct substitutes might not always exist, thus requiring custom solutions.
- **Environment and Context Variabilities:** Differences in execution environments between tools can lead to unexpected behavior in pipelines. For example, the way environment variables are managed or the context in which scripts are executed can vary, leading to potential inconsistencies.
- **Integrations with External Services:** CI/CD pipelines often integrate with external services like artifact repositories, code analysis tools, or deployment platforms. A Migration can particularly be a challenge if the services have specific integrations or optimized workflows for a particular CI/CD tool.
- **Security and Access Control:** Migrating to a new CI/CD tool might also involve reconfiguring access control and security settings. This is especially

crucial when moving from a self-hosted tool like Jenkins to a cloud-based service like GitHub Actions or GitLab CI/CD, where security models might differ significantly.

- **Acquiring Knowledge and Training:** Each CI/CD tool comes with its own set of best practices and operational knowledge. Teams might face a learning curve understanding the new tool's features, limitations, and optimal usage practices, which can impact the efficiency of the migration process.
- **Pipeline Optimization:** Migrating a pipeline is an opportunity to optimize and improve it. However, this can also be a challenge, as it requires an in-depth understanding of the new tool's capabilities and how they can be best leveraged to enhance the pipeline's performance and efficiency.

In summary, pipeline migration requires careful planning and a thorough understanding of both the source and target CI/CD tools.

## 3 Dagger.io

### 3.1 Introduction

With the increasing significance of DevOps practices such as CI/CD pipelines, a plethora of tools are emerging to aid organizations in creating, evaluating, and deploying their applications in a more effective and automated manner. As previously mentioned, there exist a multitude of tools and frameworks that can be utilized to construct such CI/CD pipelines. But such a diversity of platforms and tools brings new challenges, especially during the migration process, as shown above. This is where Dagger.io comes into play, which was created by Solomon Hykes, the founder of Docker. The Dagger platform is an open-source platform for developing programmable CI/CD pipelines and execute them entirely as standard containers based on the Open Container Initiative (OCI), Dagger itself is written with CUE lang, which is an open source data validation language [11]. At a high-level technical view the Dagger platform consists of the Dagger Engine, Dagger SDKs and the Dagger Cloud which will be given a brief overview in the following [10].

**The Dagger Engine** allows developing CI/CD pipelines as code instead of YAML or a similar markup language and run those pipelines in Docker-based standard containers based on the Open Container Initiative (OCI) [10]. The OCI is an open industry standard which uses as a basis container runtimes and formats. It was created with the support and umbrella of the Linux Foundation and launched by leaders of the container industry like CoreOS and Docker. The overall goal of this open industry standard is to endorse the expectations users of container engines like Docker have, particularly the expectation to use a container image with no need for additional augments. On a high-level technical perspective, the workflow implemented by OCI downloads an OCI image, extracts the image into an OCI runtime file system package, and then runs it from the OCI runtime [16]. The usage of standard OCI containers in the context of

DevOps practices and the execution of CI/CD pipeline helps to address some of the challenges mentioned above [10]:

- **Instant local testing:** With the ability to develop CI/CD pipelines as code, testing during operations becomes easily possible.
- **Portability:** Pipelines become portable and can be executed locally, on a dedicated server or any other container hosting server.
- **Superior caching:** Standardized chaching and in addition chaching is done by default on every operation.
- **Compatibility:** By running and executing the pipeline as standard OCI containers, it is now possible to integrate everything that works in containers and add it to the CI/CD pipeline.
- **Cross-language instrumentations:** Without having to learn each other's programming language to a full extent, teams can integrate each other's work much more easily.

**The Dagger Cloud** is a complementary optional service to the Dagger Engine and comes with a production-grade control plane. It is compatible with every machine where the Dagger Engine can run. The Dagger Cloud provides three main features [12]:

- **Pipeline visualization:** The usage of the Dagger Cloud allows visualizing and analyzing the corresponding CI/CD pipeline with a provided web interface. This provides the ability to understand the pipeline better and delve deeper into detailed logs, represent lead times of operations and see if operations were being cached.
- **Operational insights:** Dagger Cloud has the ability to gather data and measurements from the Dagger Engine and every associated pipeline across a whole organization, both post- and pre-push. It summarizes and visualizes the collected telemetry in one interface and makes it possible to have an overview of all pipelines on an organizational-level.
- **Distributed caching:** On a single machine caching is fairly simple, the Dagger Engine which is reading is also the one who is writing to the cache. But in organizations there are mostly multi-machine environment which leads to more complexity in caching, for the reason that every machine in this configuration is continuously generating and consuming cache data. Which means that it is necessary to get the right data at the right time to the corresponding machine without misspending resources. Dagger Cloud can address this issue because of the capability to collect telemetry from every Dagger Engine across the organization and therefore allows modelling the state of the whole cluster and make better caching decisions.

**The Dagger SDKs and API**
The Dagger Engine uses GraphQL, which is a low-level language-agnostic API framework, for all operations executed in Dagger, but interacting directly with the GraphQL API is only optional, because of the several provided SDKs from Dagger. Nevertheless, the GraphQL API can be directly used for example if a

pipeline needs to be built in a programming language not provided by the Dagger SDKs [13]. The Dagger platform provides the following SDKs:

- **Go SDK**
- **Node.js SDK**
- **Python SDK**
- **Elixir SDK (Experimental)**

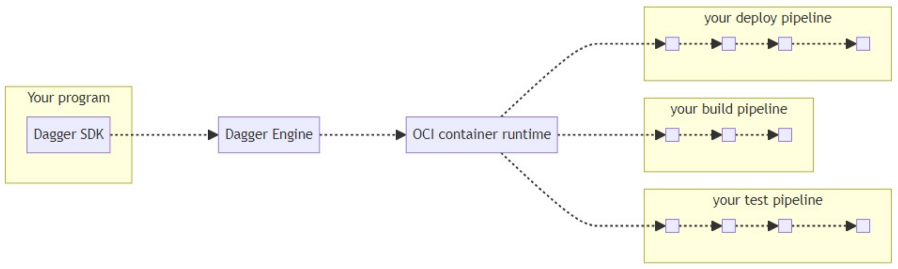


**Fig. 2.** Simplified structure of Dagger. Reference: https://docs.dagger.io/

The figure below gives a broad overview on how Dagger works [14]:

1. Your program imports the Dagger SDK.
2. It uses the SDK to open a new session to a Dagger Engine and prepares API requests describing pipelines to run, then sends them back to the engine.
3. When the engine receives an API request, it computes the result and starts processing operations.
4. After all operations in the pipeline have been resolved, the engine sends the pipeline result back to your program.

## 3.2 Example Dagger pipeline

Our team has developed a demonstration application to illustrate the capabilities of the Dagger pipeline (https://github.com/jasperfue/dagger-circle-example). In this example the Python SDK is used and it provides guidance on setting up the necessary environment to operate the pipeline. These steps can be implemented wherever you choose to run your pipeline, such as locally, on a dedicated server, in the cloud, etc. Integration with other CI tools, like GitHub Actions or GitLab CI/CD, is discussed in Section 4.

### Set up of the application

Cloning the example applicaton into the local development enivronment:

```
git clone https://github.com/jasperfue/dagger-circle-example.
    git
```

**Listing 1.1.** Command Prompt

### Install Docker

The pipeline will run in a Container, so Docker has to be installed. You can download it here: https://www.docker.com/

### Install the Dagger CLI

For Windows open PowerShell and run:

```
Invoke-WebRequest -UseBasicParsing -Uri https://dl.dagger.io/
    dagger/install.ps1 | Invoke-Expression
```

**Listing 1.2.** PowerShell

Installation guides for other operating systems can be found here: https://docs.dagger.io/cli/465058/install/
Check if Dagger is correctly installed by opening the Command Prompt terminal and run:

```
where dagger
C:\<your home folder>\dagger\dagger.exe
```

**Listing 1.3.** Command Prompt

### Install the Dagger SDK for Python

To install the Dagger Python SDK via pip run:

```
pip install dagger-io
```

**Listing 1.4.** Command Prompt

Or, install the Dagger Python SDK via Conda:

```
conda install dagger-io
```

**Listing 1.5.** Command Prompt

### Example Dagger pipeline

Our pipeline is defined in the ci/main.py file:

```python
import sys
import anyio
import dagger


async def main():
    config = dagger.Config(log_output=sys.stdout)

    async with dagger.Connection(config) as client:
        # use a node:16-slim container
        # mount the source code directory on the host
        # at /src in the container
        source = (
            client.container()
            .from_("node:16-slim")
            .with_directory(
                "/src",
                client.host().directory(
                    ".", exclude=["node_modules/", "ci/", "
    build/"]),
            )
        )

        # set the working directory in the container
        # install application dependencies
        runner = source.with_workdir("/src").with_exec(["npm"
    , "install"])

        # run application tests
        test = runner.with_exec(["npm", "test", "--", "--
    watchAll=false"])

        # build application
        # write the build output to the host
        build_dir = (
            test.with_exec(["npm", "run", "build"])
            .directory("./build")
        )

        await build_dir.export("./build")

        e = await build_dir.entries()

        print(f"build dir contents:\n{e}")

anyio.run(main)
```

**Listing 1.6.** Dagger.io pipeline in Python example

The initial step involves creating a Dagger client and initiating a new container from a base image. Then we mount the source code directory at the /src mount point in the container. After that we can go on with the testing of our application. Here we set the working directory to the mount point we just created and define the commands to install dependencies. In this case, npm install. Then we can run the tests. The last part in our minimal pipeline would be to build our application. This is done after the test passed successfully. We define the command npm run build in the container and write the build/ directory from the container to the host.

**Running the pipeline**

With this setup you can run the pipeline with the command:

```
dagger run python ci/main.py
```

**Listing 1.7.** Command Prompt

The output in the Terminal can be viewed in the Appendix at Figure 4.
As we can see, it initiated the engine, set up a Docker Container, installed dependencies, executed tests, built the application, and exported it to our build folder. The build process is a good example of the caching capabilities, as it skipped this step due to unchanged conditions from a previous pipeline run.

### 3.3 Comparison and classification of Dagger

In order to be able to classify Dagger in the context of DevOps practices and CI/CD tools, a comparison to other methods, tools and alternatives is given below [15].

**Dagger vs. CI tools:** In comparison to state of the art CI tools like GitHub Actions, Jenkins or GitLab, Dagger does not replace them but rather extends and improves them by providing the opportunity to integrate those CI tools into the Dagger pipeline. This allows for easy migration and reuse of pipelines across different CI environments.

**Dagger vs. Platform as a Service (PaaS):** PaaS (such as Heroku or Firebase) offer standardized services but also comes with disadvantages like the limitation to customize or the full control over the infrastructure and used tools. Dagger is not a PaaS, but comes with some of the benefits like a lower level degree of standardization of pipelines without sacrificing the benefits of custom CI/CD pipelines.

**Dagger vs. artisanal deploy scripts:** Custom artisanal deployment scripts are a common practice to deploy a application but often come with limitations when it comes to changing and troubleshooting those custom scripts. When using Dagger, there are two options for this kind of task:

– **Replace** the old script with a Directed Acyclic Graph (DAG)
– **Extend** the old script by wrapping it into a Directed Acyclic Graph (DAG).

**Dagger vs. Infrastructure as Code (IaC):** IaC tools like Pulumi, Terraform or Cloudformation can aid in comprehending and analyzing an organization's current infrastructure state and comparing it to the desired state. Dagger, in particular the Dagger Cloud, can help to support this process, which means that Dagger does not replace IaC tools but is rather a complementary platform for those.

**Dagger vs. Build Systems:** Dagger helps to integrate build systems such as Make, Maven, Bazel, Npm/Yarn or Docker Build which means that Dagger complement such tools but does not replace them.

## 4 Integrating Dagger.io

To run a Dagger pipeline in a CI environment the only prerequisite is having Docker pre-installed. At the core, all you have to do in your CI-environment is to install the Dagger CLI and the desired SDK and run the Dagger pipeline script. In this work, we will focus on integrating the example Dagger pipeline discussed in Section 3.2 into the aforementioned CI/CD-environments Github Actions, Gitlab and Jenkins. GitHub Actions and GitLab CI files can also be viewed in our example pipeline repository (https://github.com/jasperfue/dagger-circle-example).

### 4.1 Github Actions

Using Dagger in Github Actions is pretty straightforward. All you have to do is define a workflow with a single "build" job that sets up the python environment, installs the Dagger CLI and Python SDK and runs the Dagger pipeline. The workflow is written in YAML, so developers will still have to have a basic understanding of the GitHub Actions syntax [17].

```yaml
name: dagger
on:
  push:
    branches: [main]

jobs:
  build:
    name: build
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-python@v5
        with:
          python-version: '3.11'
      - name: Install Dagger SDK
        run: pip install dagger-io
      - name: Install Dagger CLI
        run: cd /usr/local && { curl -L https://dl.dagger.io/
    dagger/install.sh | sh; cd -; }
      - name: Run Dagger pipeline
        run: dagger run python ci/main.py
```

**Listing 1.8.** Dagger integration into Github Actions

The pipeline's appearance in GitHub is shown in Figure 5 in the Appendix. All steps completed successfully, and the Logs provide detailed insights, useful for troubleshooting in case of any failures.

### 4.2 Gitlab CI

To integrate Dagger into Gitlab CI, you once again have to manipulate the corresponding YAML-file. First, you set up a docker environment by choosing the docker service ”dind”(Docker-in-Docker) and declaring docker variables. After installing the Dagger CLI and SDK, the python script containing the pipeline can be run [17].

```
.docker:
    image: python:3.11-alpine
    services:
        - docker:${DOCKER_VERSION}-dind
    variables:
        DOCKER_HOST: tcp://docker:2376
        DOCKER_TLS_VERIFY: 1
        DOCKER_TLS_CERTDIR: /certs
        DOCKER_CERT_PATH: /certs/client
        DOCKER_DRIVER: overlay2
        DOCKER_VERSION: 20.10.16
.dagger:
    extends: [.docker]
    before_script:
        - apk add docker-cli curl
        - cd /usr/local && { curl -L https://dl.dagger.io/
   dagger/install.sh | sh; cd -; }
build:
    extends: [.dagger]
    script:
        - pip install dagger-io
        - dagger run python ci/main.py
```

**Listing 1.9.** Dagger integration into Gitlab

Figure 6 in the Appendix displays the Dagger pipeline outcome in GitLab CI. Similar to GitHub Actions, detailed log examination enhances identifying any failures.

## 4.3 Jenkins

Out of the three CI/CD tools, integrating Dagger into Jenkins is the most complex one. A docker client and python have to be installed on your Jenkins client first. Also, a Docker host(i.e. dind) has to be available. Lastly, agents in Jenkins have to be labeled with Dagger. The resulting Jenkinsfile will look as follows:

```
pipeline {
  agent { label 'dagger' }

  stages {
    stage("dagger") {
      steps {
        sh '''
            pip install dagger-io
            cd /usr/local && { curl -L https://dl.dagger.io/
   dagger/install.sh | sh; cd -; }
            dagger run python ci/main.py
            '''
        }
      }
    }
}
```

**Listing 1.10.** Dagger integration into Jenkins

The Dagger agent runs a shell script that installs all necessary packages and runs the pyhton script with the pipeline [17].

## 5 Discussion

After having looked at how Dagger.io works and how one can integrate it into different CI/CD tools, the next logical step is to analyze its advantages and disadvantages. The most outstanding advantage of Dagger.io is its flexibility. It is operable on any system with Docker installed, which allows for a seamless infrastructure transition, whether it be the transition to a server, the cloud, a GitHub runner or a local system. Another big advantage of Dagger is their code-based pipelines. Developers can choose from various SDK's and build their pipelines in a convenient way without having to deal with complex YAML or Jenkinsfile syntax. Moreover, the platform's commitment to expanding language support ensures that it stays relevant and adaptable to emerging technologies. The superior caching abilities of Dagger are also a great advantage. Like already mentioned, Dagger allows the user to cache data across pipeline runs. This is especially useful when dealing with package managers such as pip or npm. The user can reuse contents across pipeline runs and speed up pipeline operations [10]. The last advantage that needs to be mentioned is that Dagger.io allows for multi-file pipelines. Developers can code their pipelines and define them in different files, which leads to more modularity and better maintainability. However, one also has to talk about the disadvantages of Dagger.io. As of today, there are no community extensions for Dagger, but they are planning on deploying a similar marketplace like GitHub [18]. This is due to Dagger being a relatively new tool and may change in the future. Extensions can be programmed using CueLang, but this would mean that the developer has to first learn and master this language. The biggest risk of implementing Dagger is the technology adoption risk that one takes. If the company behind Dagger discontinues the platform, problems will arise. No more updates for Dagger means that the development team now has to remigrate from Dagger to conventional CI/CD environments once again. This would cause immense costs because developers would now have to reprogram their complex Dagger-based pipelines in another environment. Another downside is, that developers still need to know the basic syntax of the CI/CD tool they are migrating to or from. Nevertheless, there are tools other than Dagger that are independent from the source code host like Jenkins. However, Dagger distinguishes itself through its flexible integration capabilities across various CI environments, thereby enabling smoother migrations. It is not restricted to a specific infrastructure, in contrast to for example Jenkins, which requires a dedicated server.

**Personal reflection** After having discussed the general advantages and disadvantages of Dagger, we want to give a personal reflection. While developing the minimal Dagger pipeline and integrating it to GitHub Actions and GitLab CI/CD, our perception of Dagger was mainly positive. The integration progress was convenient and fast, taking round about two hours to finish. This was possible thanks to the excellent documentation of Dagger.io. Nonetheless, there were two minor inconveniences that we faced. For one, the Docker Desktop application always has to be running when working with Dagger. This makes sense, but was irritating at first. Another minor inconvenience was, that the Dagger client requested us to disable our firewall when attempting to install the Dagger CLI. This can lead to developers questioning the trustworthiness of Dagger. But all in all, our experience with Dagger was mainly positive. In our opinion, it is a promising tool. The decision whether to use it heavily depends on your use case. It undeniably makes the migration between services more convenient and the whole CI/CD process more accessible. But as with every new technology, you run the risk of wasting resources.

## 6 Conclusion

CI/CD pipelines have become essential for every software development process. Although being the industry standard, CI/CD tools such as GitHub Actions or GitLab CI do have their limitations when it comes to flexibility and portability. Depending on the situation, changing the CI/CD environment and migrating pipelines from one tool to another can be a daunting and laborious task. Dagger uses cutting-edge technologies to solve some of the biggest challenges when migrating pipelines. The Dagger Engine allows developers to build pipelines in the programming language of their choice (given that Dagger provides the SDK) and are freed from learning different syntax for different CI/CD tools. Thanks to Dagger, pipelines become portable and can be executed everywhere. Not only does Dagger make pipelines more compatible and flexible, it also speeds up pipeline operations thanks to its superior caching. Our exemplary pipeline prototype showed that the migration from one CI/CD tool to another can be done in relatively little time. Dagger does not replace the CI/CD tool of your choice, but rather improves it. This makes it possible to use Dagger with almost every CI/CD tool, as long as it supports Docker. One Drawback is the non-existence of Extensions. Depending on specific project requirements, this might lead to a significant effort being invested in pipeline development, in contrast to tools with available extensions that can reduce such a need. In Addition CueLang can be another hurdle to program the extensions. CueLang might be complicated to learn, but when mastered, allows for excellent customization of Dagger. The technology adoption risk is also existent, but that is the case with every new technology and not exclusive for Dagger. We therefore recommend that development teams facing challenges with pipeline migration, or anticipating a future migration to a different CI tool, consider incorporating Dagger into their technology stack.

## References

1. PagerDuty Deployment Pipeline, https://www.pagerduty.com/resources/learn/what-is-a-deployment-pipeline/
2. Farhana Sethi: AUTOMATING SOFTWARE CODE DEPLOYMENT USING CONTINUOUS INTEGRATION AND CONTINUOUS DELIVERY PIPELINE FOR BUSINESS INTELLIGENCE SOLUTIONS. International Journal of Innovation Scientific Research and Review. (2020)
3. Henry van Merode: Continuous Integration (CI) and Continuous Delivery (CD) (2023)
4. Olga Bedrina: Best Continuous Integration Tools for 2023 – Survey Results https://blog.jetbrains.com/teamcity/2023/07/best-ci-tools/ (2023)
5. GitHub Actions Marketplace https://github.com/marketplace?type=actions. Last accessed 31 Jan 2024
6. GitLab CI https://about.gitlab.com/solutions/continuous-integration/. Last accessed 30 Jan 2024
7. Jenkins Documentation https://www.jenkins.io/doc/book/. Last accessed 30 Jan 2024
8. Jenkins Blue Ocean Documentation https://www.jenkins.io/doc/book/blueocean/. Last accessed 30 Jan 2024
9. Jenkins Plugins https://plugins.jenkins.io/. Last accessed 30 Jan 2024
10. Dagger Documentation, https://docs.dagger.io/. Last accessed 30 Jan 2024
11. CUE lang, https://cuelang.org/docs/about/. Last accessed 31 Jan 2024
12. Dagger Documentation, https://docs.dagger.io/cloud. Last accessed 30 Jan 2024
13. Dagger Documentation, https://docs.dagger.io/api. Last accessed 30 Jan 2024
14. Dagger Documentation, https://docs.dagger.io/quickstart/319191/basics. Last accessed 30 Jan 2024
15. Dagger Documentation, https://archive.docs.dagger.io/0.2/1220/vs/. Last accessed 30 Jan 2024
16. About the Open Container Initiative, https://opencontainers.org/about/overview/. Last accessed 30 Jan 2024
17. Dagger.io documentation, https://docs.dagger.io/cookbook#integrations. Last accessed 30 Jan 2024
18. GitHub Issue on Dagger universe for Extensions https://github.com/dagger/dagger/issues/3206. Last accessed 30 Jan 2024

## A Appendix

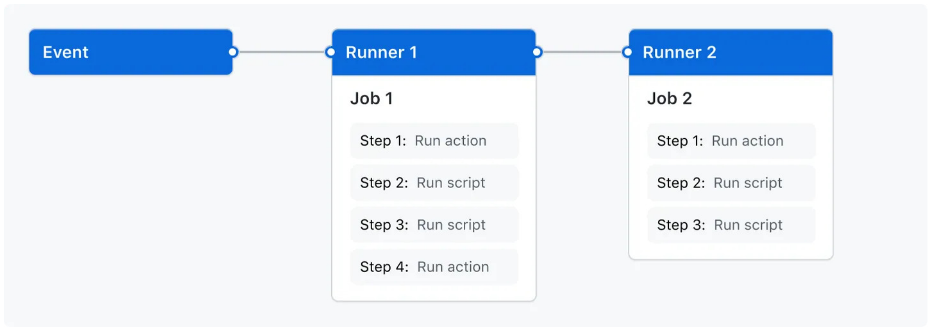


**Fig. 3.** Simplified structure of a GitHub Action workflow. Reference: https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions

```
name: CI/CD Pipeline

on:
  push:
    branches: [ main ]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: Set up Node.js
      uses: actions/setup-node@v4
      with:
        node-version: '18'
    - name: Install dependencies
      run: npm install
    - name: Run tests
      run: npm test
    - name: Run build
      run: npm run build
```

**Listing 1.11.** Minimal GitHub Actions example of a Node.js Project

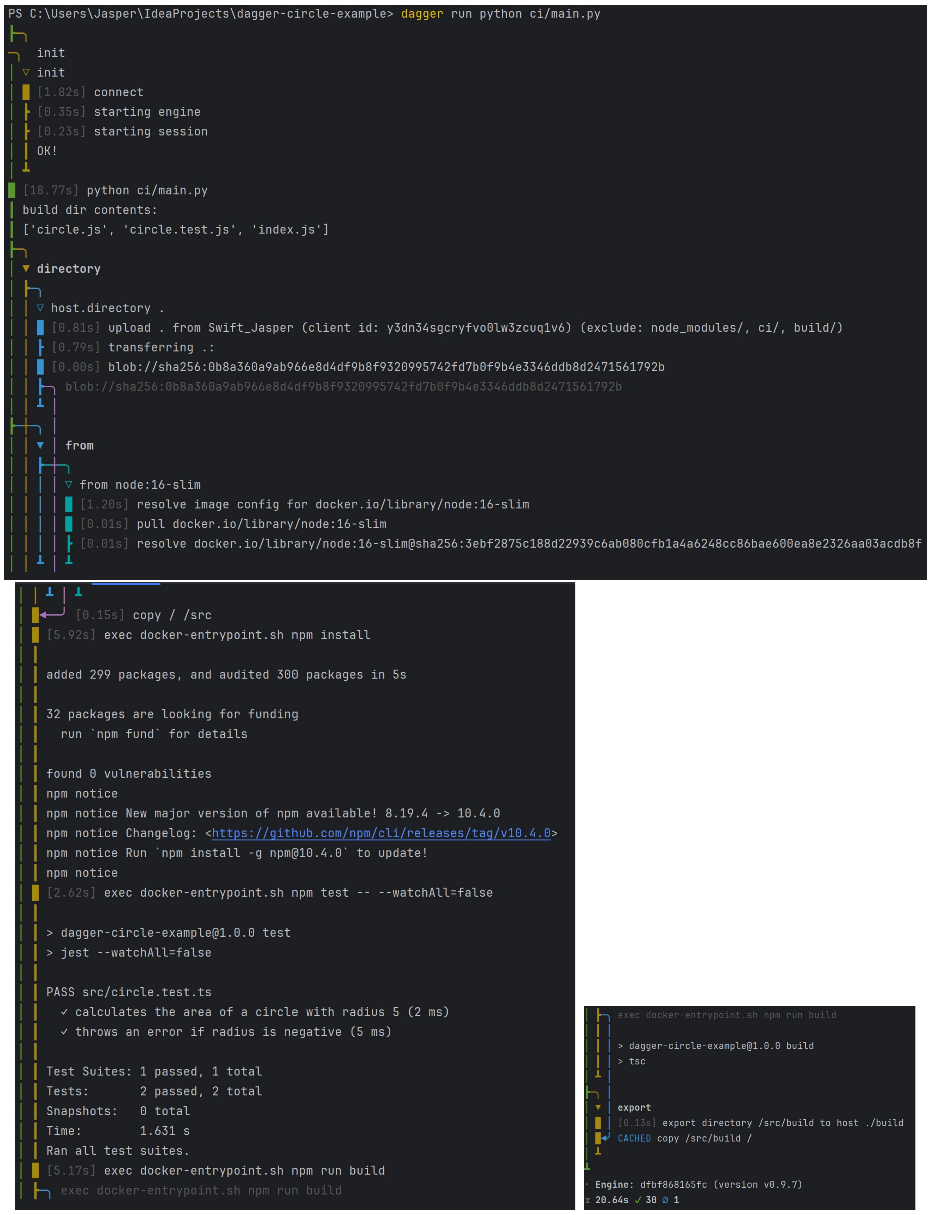
```
stages:
  - build
  - test
  - deploy

```

```
cache:
  paths:
    - node_modules/

build-job:
  stage: build
  image: node:18
  script:
    - npm install

test-job:
  stage: test
  image: node:18
  script:
    - npm test
  only:
    - main

deploy-job:
  stage: deploy
  image: node:18
  script:
    - npm run build
  only:
    - main
```

**Listing 1.12.** Minimal GitLab CI/CD example of a Node.js Project

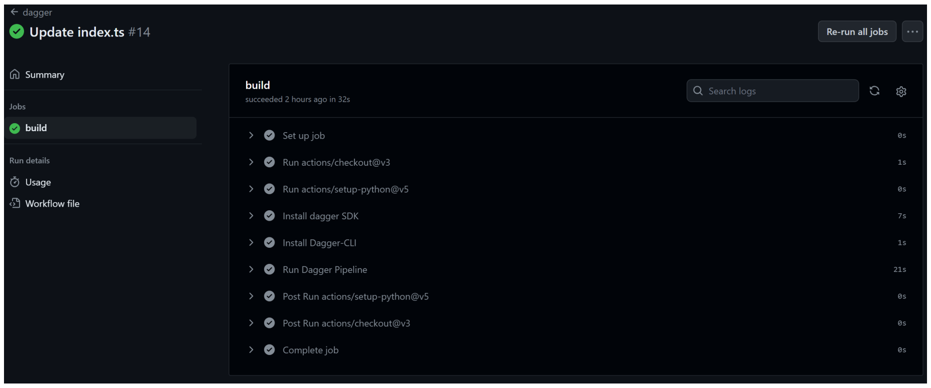
```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                // Pull the Source Code from the SCM
                checkout scm
            }
        }

        stage('Install Dependencies') {
            steps {
                sh 'npm install'
            }
        }

        stage('Test') {
            steps {
                sh 'npm test'
            }
        }

        stage('Build') {
            steps {
                sh 'npm run build'
            }
        }
    }
}
```

**Listing 1.13.** Jenkinsfile of a minimal Pipeline example of a Node.js Project



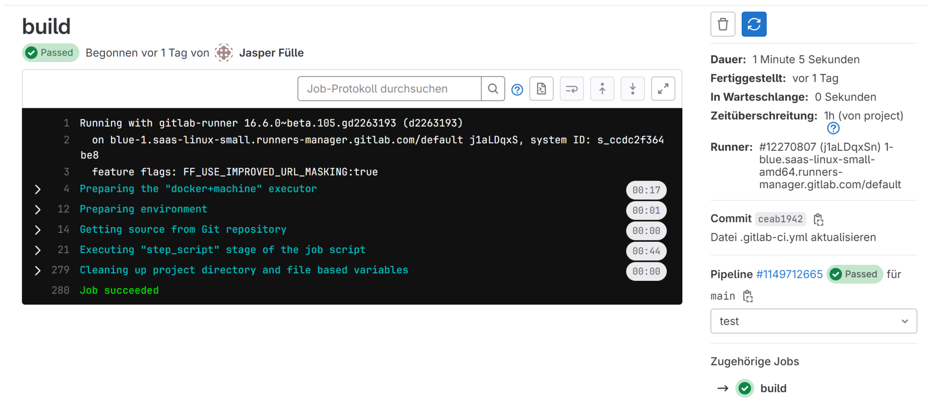
```
PS C:\Users\Jasper\IdeaProjects\dagger-circle-example> dagger run python ci/main.py

  init
  ▽ init
  [1.82s] connect
  [0.35s] starting engine
  [0.23s] starting session
  OK!

[18.77s] python ci/main.py
build dir contents:
['circle.js', 'circle.test.js', 'index.js']

▼ directory

  ▽ host.directory .
  [0.81s] upload . from Swift_Jasper (client id: y3dn34sgcryfvo0lw3zcuq1v6) (exclude: node_modules/, ci/, build/)
  [0.79s] transferring .:
  [0.00s] blob://sha256:0b8a360a9ab966e8d4df9b8f9320995742fd7b0f9b4e3346ddb8d2471561792b
    blob://sha256:0b8a360a9ab966e8d4df9b8f9320995742fd7b0f9b4e3346ddb8d2471561792b


  ▼   from

      ▽ from node:16-slim
      [1.20s] resolve image config for docker.io/library/node:16-slim
      [0.01s] pull docker.io/library/node:16-slim
      [0.01s] resolve docker.io/library/node:16-slim@sha256:3ebf2875c188d22939c6ab080cfb1a4a6248cc86bae600ea8e2326aa03acdb8f

      [0.15s] copy / /src
  [5.92s] exec docker-entrypoint.sh npm install

  added 299 packages, and audited 300 packages in 5s

  32 packages are looking for funding
    run `npm fund` for details

  found 0 vulnerabilities
  npm notice
  npm notice New major version of npm available! 8.19.4 -> 10.4.0
  npm notice Changelog: <https://github.com/npm/cli/releases/tag/v10.4.0>
  npm notice Run `npm install -g npm@10.4.0` to update!
  npm notice
  [2.62s] exec docker-entrypoint.sh npm test -- --watchAll=false

  > dagger-circle-example@1.0.0 test
  > jest --watchAll=false

  PASS src/circle.test.ts
    ✓ calculates the area of a circle with radius 5 (2 ms)
    ✓ throws an error if radius is negative (5 ms)

  Test Suites: 1 passed, 1 total
  Tests:       2 passed, 2 total
  Snapshots:   0 total
  Time:        1.631 s
  Ran all test suites.
  [5.17s] exec docker-entrypoint.sh npm run build
    exec docker-entrypoint.sh npm run build
    exec docker-entrypoint.sh npm run build

    > dagger-circle-example@1.0.0 build
    > tsc


  ▼ export
  [0.13s] export directory /src/build to host ./build
  CACHED copy /src/build /


· Engine: dfbf868165fc (version v0.9.7)
⧗ 20.64s ✓ 30 ∅ 1
```

**Fig. 4.** Output of the example dagger Pipeline



```
build
succeeded 2 hours ago in 32s

Set up job                              0s
Run actions/checkout@v3                 1s
Run actions/setup-python@v5             0s
Install dagger SDK                      7s
Install Dagger-CLI                      1s
Run Dagger Pipeline                    21s
Post Run actions/setup-python@v5        0s
Post Run actions/checkout@v3            0s
Complete job                            0s
```

**Fig. 5.** Example dagger Pipeline in GitHub Actions


build
Passed
Begonnen vor 1 Tag von Jasper Fülle
Job-Protokoll durchsuchen

```
Running with gitlab-runner 16.6.0~beta.105.gd2263193 (d2263193)
  on blue-1.saas-linux-small.runners-manager.gitlab.com/default j1aLDqxS, system ID: s_ccdc2f364
be8
  feature flags: FF_USE_IMPROVED_URL_MASKING:true
Preparing the "docker+machine" executor                          00:17
Preparing environment                                            00:01
Getting source from Git repository                               00:00
Executing "step_script" stage of the job script                  00:44
Cleaning up project directory and file based variables           00:00
Job succeeded
```

**Dauer:** 1 Minute 5 Sekunden
**Fertiggestellt:** vor 1 Tag
**In Warteschlange:** 0 Sekunden
**Zeitüberschreitung:** 1h (von project)

**Runner:** #12270807 (j1aLDqxSn) 1-blue.saas-linux-small-amd64.runners-manager.gitlab.com/default

**Commit** ceab1942
Datei .gitlab-ci.yml aktualisieren

**Pipeline** #1149712665 Passed für main

test

Zugehörige Jobs
→ build

**Fig. 6.** Example dagger Pipeline in GitLab CI