Production Systems and Information Engineering
Volume 10 (3), pp. 65–69
doi: 10.32968/psaie.2022.3.6

BUILD AUTOMATION SYSTEMS AGAINST CI LOCK-IN – A COMPARATIVE STUDY OF DAGGER AND MAGE

Áron Kiss
University of Miskolc, Hungary
Institute of Information Technology
kiss.aron@uni-miskolc.hu

Abstract. Vendor lock-in is a well-known phenomenon in the software industry. Strongly relying on vendor-specific implementation may cause financial and technological hardships to manufacturers and can establish monopoly situation of a vendor. With the spread of cloud-based development tools, vendor lock-in is present not only during operation, but also during development. This article provides an overview of risk types introduced to projects by vendor lock-in situation. Key factors of vendor lock-in are also identified, especially with regard to modern cloud-based CI/CD services. Later, a test software architecture is demonstrated how to minimize CI lock-in, followed by a detailed comparison of two build automation systems that can be used in practice for this purpose. The applicability of build automation systems is demonstrated on the implementation and test results.

Keywords: software build automation, continuous integration, agile software development, software quality assurance

1. Introduction

Vendor lock-in is a well-known phenomenon in IT systems. With the growing popularity of serverless computing, PaaS solutions and other cloud technologies, the transfer of cloud-based software between service providers became a more difficult task for software engineers. When a complex IT system is excessively depends on vendor-specific implementation, the migration to another service provider (or either to an in-house solution) may become technologically complicated, time-consuming and costly [1].

Vendor lock-in is present during the development and operation of today's modern software systems. During the planning of software products and their operation, software manufacturers must pay special attention to this problem, otherwise switching barriers will arise and the adaption to unforeseen events

(e.g. major changes in customer requirements, new legal constrains, significant price increase, termination of a specific service) become technologically complicated and/or financially impossible.

2. Vendor lock-in in cloud-based software products

Organizations developing modern software products often follow the "cloud-native" approach in the development and operation of their product. It means that cloud-based tools and services are used in the project to design, build, test and later maintain the system in order to achieve elastic, scalable, loosely-coupled and self-contained software components [2].

[3] examined cloud computing migration from a business perspective. [4] created a model for classifying lock-in risk factors. Literature shows that vendor lock-in may introduce risks to a system in the following areas:

1. **Data transfer**: Data migration and conversion might be costly between different cloud providers. Data transfer is likely to cause outage of service. Loss of application functionality may occur due to incompatible data storage solutions.
2. **Application transfer**: When a system depends on proprietary standards and interfaces of a certain cloud service provider, migrating to another cloud solution involves major changes in application code. Depending on standard interfaces and open APIs is suggested to avoid this type of risk.
3. **Infrastructure transfer**: Virtual machine technologies, offerings of managed services and the applied pricing models may vary from vendor to vendor. It can be a complex problem to calculate which services should be configured how to reduce costs (or keep it at the same level) when switching between cloud providers, compared to an application intentionally designed for a multi-cloud environment.
4. **Human knowledge**: Development and Operations team of a software system is likely to gain a deeper level of knowledge among the current vendor's products. It may take significant amount of time to learn different infrastructure formats and implementation processes of the new vendor.

Relying on open standards, protocols and interfaces are generally support interoperability and migration capability of cloud-enabled software systems well. In addition, novel methods, strategies and frameworks are being published with the aim to avoid or mitigate the risks of vendor lock-in.

[5] proposed a software design pattern to handle risks of vendor lock-in by introducing a standardized process of interacting with cloud service providers. [6]

proposed an abstraction model called "vendor agent" to support cloud interoperability. The vendor agent serves as a unified resource provisioning interface between application and different cloud service providers. [7] introduced an open-source system for small and medium-sized enterprises for deploying and scaling containerized applications and for supporting transferability of the containers between IaaS providers. [8] created a framework for enterprises to avoid being locked to individual cloud service providers and proposed several strategies to avoid and mitigate lock-in risks when migrating an already existing system to the cloud.

3. Vendor lock-in in the world of managed CI services

Cloud-based technologies are not only used during the operation of software projects, but are also often present in the development process itself. The usage rate of managed Continuous Integration (CI) services illustrates this phenomenon well.

Managed CI/CD solutions (e.g., GitHub Actions, Azure Pipelines, TravisCI, AppVeyor) are hosted, scaled, secured and generally maintained by an external organization. These external organizations only provide CI/CD capabilities to the development team through APIs. This can offload a large amount of work from the development team. However, managed CI/CD services are often considered less secure than on-premise solutions [9].

[10] found that approximately 40% of open-source projects are use at least one CI service. They find, that most popular reasons behind using a CI service are: 1) less chance of breaking the build, 2) possibility of early bug catching, 3) by running tests in the cloud, resources on local machines freed up.

Users of managed CI services are in danger of vendor lock-in, or as other sources call it, *CI lock-in*. Termination, a significant price change or a usage limit exhaustion of a CI provider's service can originate the need to migrate the entire CI pipeline of a project to another provider's infrastructure. However, different vendors are applying different configuration rules and methods for the definition of the CI/CD pipeline. I present major factors that may complicate transfering the pipeline between different CI solutions.

YAML is currently the de-facto standard language for CI pipeline configuration, but other languages (e.g. *Jenkinsfile*s are written in a language with Groovy-like syntax [11]) and methods (e.g. AppVeyor supports GUI-based configuration of the entire pipeline) are also present.

The terminology used to construct the build pipeline shows differences between CI service providers:

- Pipelines in **Github Actions** are made up of *steps* of *jobs*. Jobs are organized into *workflows* [12],
- **Azure Pipelines** are similarly uses the terminology of *steps* and *jobs*, but it organizes jobs into *stages* rather than workflows [13],
- **Travis CI** applies the terminology of *phase* for the sequential step of a *job*. A group of parallel jobs are organized into a *stage*. The pipeline (which is called *build*) consists of a sequence of jobs [14],
- **AppVeyor** provides *jobs*, which are consist of *scripts* with *lines*. Jobs are assigned to pre-defined *phases* of the build. Phases are ran sequentially to build the project [15].

There are also differences in the execution model of the CI/CD pipeline. For example while for certain services, pipeline can be divided into arbitrary parts and the build process can be assembled by chaining these [12, 13, 14], other services have pre-defined phases of the build and requires the developer to define callback scripts for these phases [15].

The vast majority of CI services are support defining a *build matrix*. This means that user can define various environments by adding one or more "dimensions" (e.g. operating system, architecture, platform, runtime tools) to the configuration. After that, the build process will run on every combination of values from each dimension.

Table 1 shows an example of a 2D build matrix with a set of 2 *values* in each **dimension**. This means that the build process will run in 4 different environments.

Table 1. Example of a build matrix

		Operating system	
		Windows 11	*Ubuntu 22*
Runtime env.	*Node.js 16*	Win11+Node16	Ubuntu22+Node16
	Node.js 18	Win11+Node18	Ubuntu22+Node18

Each provider supports specifying partially different dimensions and expects the build matrix to be specified with a different data structure.

The factors mentioned above are affect complexity of migrating the build pipeline between CI providers. Project migration can be especially complicated, when the compilation process of the project is intricate, and the build needs to be run in several environments to ensure reliability of the software.

4. Mitigating CI lock-in with modern build automation systems

Historically, automation of the build process was most often accomplished through makefiles. Nowadays, there are many modern build automation tools

and systems are available for software engineers [16, 17, 18, 19]. These systems allow developers to define tasks with several steps and specify dependencies between these tasks. The build automation system then executes and orchestrates these tasks (e.g. independent tasks run in parallel) in order to build the project as efficiently as possible.

Automated build systems can also be used to mitigate CI lock-in in projects where a managed CI service is used. The proposed architecture is shown in Figure 1. The vendor-specific code is indicated by rectangles surrounded with dashed lines.

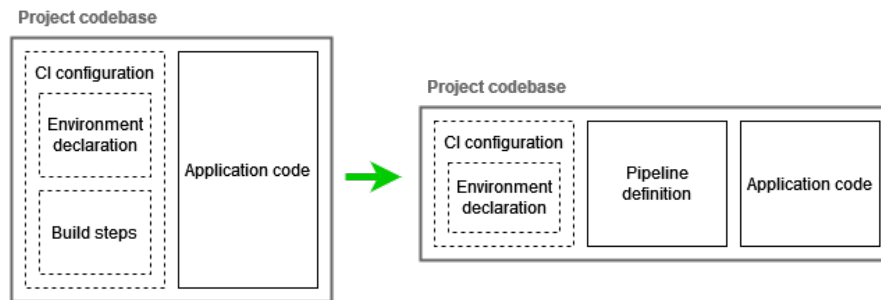

Figure 1. Software architecture to mitigate CI lock-in

On the left is the traditional architecture, where the CI/CD pipeline of the project (practically, the environments' declaration and the build steps itself) is declared in a vendor-specific data structure and format. In the proposed architecture, the pipeline definition is moved out from the configuration, and the provider-specific CI configuration file only contains environment declarations and the invocation of the build pipeline. In this architecture, the pipeline is implemented with the help of an automated build system.

By applying this architectural solution, if a service provider changes, the implementation of the build process remains the same, it does not need to be rewritten according to a new data structure or format, not even in the case of complex build tasks. When switching a new CI provider, development and operations team only need to configure the environment(s) used, the installation of the build system, and the starting of a specific build target. This helps to minimize the risk of vendor lock-in, because only the actual vendor-specific settings need to be changed. The migration between CI service providers or switching to an in-house solution is thus significantly simplified this way.

Modern build systems support different models to implement the build process, basically 2 approaches are typical:

- **Declarative** approach focuses on the description of the build goals, without explicitly specifying its control flow (e.g. YAML-based managed CI services are applying this approach),
- **Imperative** approach focuses on the concrete steps need to be done to reach the build goals (e.g. Jenkins' Scripted Pipeline is based on this approach).

5. Comparison of Dagger and Mage

The applicability of two considerably different build systems have been examined in order to eliminate CI lock-in in a test software project. Automated build system called *Dagger* applies the declarative approach, and supports building pipeline definitions written in CUE language [18]. An another build automation system called *Mage* follows the imperative approach, because it allows to define build steps using plain Go functions [19].

5.1. Build pipeline for demonstration

Ability of the above mentioned build automation systems to avoid CI lock-in was demonstrated on the codebase of an actively maintained open-source NPM package [20], which supports managing e-payment transactions through a payment gateway service.

The original project currently has two CI pipelines (unit tests are executed at every push to the remote repository, integration tests are scheduled and can be started manually) that are executed in several environments – defined in a build matrix – on GitHub Actions.

For the demonstration, the pipeline that runs the unit tests is implemented based on the compared build systems. Figure 2 shows the flowchart of the pipeline.

The pipeline consists of an initialization step, during which two environments need to be set up: an Ubuntu instance with Node.js 14 and an another Ubuntu instance with Node.js 16. Then, the different environments execute the same build steps in parallel: 1) fetching the codebase from the central repository, 2) installing production and development dependencies of the project, 3) running unit tests and measuring code coverage, 4) reporting coverage statistics to a third-party service. When all the environments have finished running the build process successfully, a webhook endpoint of the code coverage service must be called to merge statistics from the parallel measurements.

The pipeline was implemented separately with Dagger and with Mage, focusing on keeping the amount of vendor-specific code as few as possible.

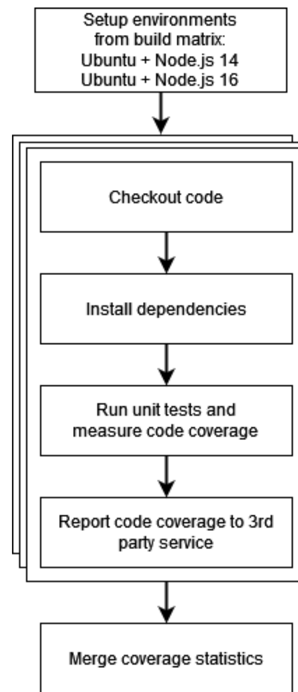

Figure 2. Flowchart of the build pipeline

5.2. Build process

The two build systems are offer different tools and terminology to structure the build process.

Dagger pipelines are defined in CUE, which is a novel declarative language, designed to describe data schemas and configurations (CUE stands for "configure, unify, execute"). The pipelines have built on the top of 4 main building blocks: *secrets*, *client*, *actions*, and a *plan* [18].

Actions are the basic building blocks of the Dagger platform. An action encapsulates an arbitrarily complex process of the build pipeline. Actions can be safely shared, because they are reproducible on other machines and environments, due to virtualization techniques used by the Dagger engine. An action can depend on other actions in an implicit way (e.g. it can use other action's output as an input). When an action is invoked by the user with the command-line interface (CLI), Dagger engine pieces together dependent actions based on a Directed Acyclic Graph (DAG) generated from the CUE definition. Explicit action chaining is not supported yet, only workarounds are exist [18]. Actions are defined as building blocks of the plan. For the implementation of the plan,

the developer can import external packages from the *Dagger Universe*. These external packages can contain pre-defined actions and data schemas [21]. A plan can also read and use the state of the local machine through the "client" object (e.g. read and write the local filesystem, load environment variables, run commands, etc.). Of course, dependence on the state of the local machine should be reduced to a minimum in order to keep the CI pipeline reproducible in other environments as well. Secret values also appear in the plan. Secrets can be read from files and environment variables and handled by the Dagger runtime in a special way to minimize the risk of leak.

Mage uses a different concept for the definition of build pipelines. It stores the pipeline in the form of plain-old Go functions [19].

When developers define the pipeline, their task is to implement private and exported functions, that are run on the local machine. The exported functions are called *build targets*. Build targets can be called from the CLI of Mage.

For the implementation of the build targets, any traditional Go packages and modules can be imported in the code. This supports minimizing the implementation time of the build pipeline through code reuse. Mage also contains several built-in packages, which support interaction with the system shell (e.g. run commands, read environment variables, customize log messages, etc.) [19]. Build targets can be chained explicitly, dependencies can be defined with classical function call.

5.3. External dependencies

Both build systems have external dependencies. These must be present on the host system to run build pipelines:

- Dagger requires a Docker set up in order to be able to build containers using BuildKit, and also to be able to run the created containers,
- Mage requires the Go standard library installed, to be able to compile build targets (functions) written in Go programming language.

5.4. Build isolation and reproducibility

In comparison with Dagger, Mage does not natively support virtualization (however it can be achieved with external Go packages, like s2i or Docker SDK), the build pipeline's code will run directly on the local machine. This does not guarantee the reproducibility of the pipeline in different environments by design. Dagger runs the pipeline in Docker containers created with BuildKit from the CUE definition, but it can also access the state of the local machine, so the developer needs to be aware of reproducibility through the implementation of the pipeline.

Mage does not include a separate procedure for managing secret values, leaving it up to the developer to prevent them from leaking. Dagger supports handling secrets in a special way, when the developer uses dagger.#Secret datatype. These secrets are then managed by the Dagger runtime and being only referenced as opaque identifiers. Furthermore, the system masks the actual secret values on logging, in order to prevent unintentional leakage.

5.5. Build matrix support

Defining the build matrix is usually a vendor-specific task, because different vendors provide different dimensions and the use of different value sets on these dimensions. However, Dagger runs the actions in containers, which are virtualized environments. Therefore, it is possible to run the same build pipeline in several different environments.

Listing 1. shows a simple implementation of a build matrix. It uses templating in CUE to define the same build actions in different environments. This code creates two tasks in the "build" action with the same content. The variable "docker_image" in line 7 can be used to pull the appropriate Docker image in each task for running the build.

Listing 1. Build matrix in Dagger

```
1  dagger.#Plan & {
2    actions: {
3      build: {
4        "node:lts-gallium": _    // Debian with Node.js v16
5        "node:lts-fermium": _    // Debian with Node.js v14
6
7        [docker_image=string]: {
8          // Dagger runs the same tasks defined here
9          // in containers constructed from Docker images
10         // "node:lts-gallium" and "node:lts-fermium".
11       }
12     }
13   }
14 }
```

This is not a complete build matrix solution, as all the possible environments must be listed explicitly. This is inadvantageous when a particularly large number of environments are used.

Mage does not have this level of flexibility by design, because it runs the build pipeline directly on the host machine. Using a virtualized environment is only be possible by introducing external Go packages and modules. The definition of the build matrix is thus specified in a vendor-specific data format,

which in most cases cannot be transferred without changes between two service providers.

5.6. Managing dependent tasks

The two build systems use different models for managing dependent tasks.

In the case of Dagger, it is only possible to specify dependencies implicitly. When an action is invoked through Dagger CLI, the system parses CUE definition to create the DAG, which models the sequence of execution. For instance, an action can depend in other action's output or final state (success or fail). Actions that depend on each other's status can only be executed sequentially, while actions that do not have such a dependency can also run in parallel.

In the case of Mage, dependencies between tasks are defined explicitly. When a function depends on the return value of another function, it simply calls the appropriate function and processes its return value/error.

5.7. Availability of pre-defined tasks and environments

Reusable functionality is available for both of the build systems.

In the case of Dagger, Docker images can be pulled from private and public Docker registries, and these can then be used to construct environment for the build. Dagger also provides the usage of Dagger Universe packages and the overall CUE package ecosystem [21]. That means, that pre-defined tasks and data schemas can be imported. Dagger Universe is in early stage of development, it contains few tools, and has a very basic ecosystem now.

Mage supports the usage of functions from the Go standard library and allows developers to use other published packages and modules in the project too. The Go ecosystem applies a decentralized publishing model, which means that anyone can freely publish Go modules in their code repository. Registering in a central service and pushing the code to it is not necessary for the maintainers of modules [22]. Mage's built-in modules can also be used to implement the build pipeline.

5.8. Running builds in a managed CI/CD service

The build pipelines are ran on Github Actions to present how much vendor-specific configuration is required in addition to the vendor-independent pipeline implementation.

For Dagger, the vendor-specific configuration involved:

- set the required environment variables,

- configure the used environment (Ubuntu virtual machine),
- setup Dagger build system,
- invoke build actions sequentially.

In the event of a service provider change, this configuration must be transferred to the data structure of the new service provider. For specifying the build matrix, there is no full-fledged solution in Dagger, as all the environments (containers) used must be listed explicitly, so if many environments are used, their definition may also be included in the vendor-specific configuration structure.

For Mage, the vendor-specific configuration involved:

- set the required environment variables,
- configure build matrix (Ubuntu with Node.js 14 and with Node.js 16),
- setup Mage build system,
- setup the execution flow for build targets by using jobs:
  1. firstly, the build process must be run in the environments defined by the build matrix,
  2. after the separate builds in all of the environments are finished successfully, a webhook call should be sent to the remote code coverage service.

It can be seen that in the case of Mage, more vendor-specific configuration settings are required (e.g. a build matrix and dependent jobs need to be defined), compared to Dagger. This is due to the lack of built-in virtualization, since in the case of Dagger the build pipeline itself can start different environments (containers), while Mage runs the pipeline relying on the shell of the specified host system. Running the build thus requires more vendor-specific code, however, the build commands are also simple enough for Mage to be considered as portable.

5.9. Build time benchmarks

The execution time of the build process was measured for both build systems on a local computer, with the following configuration:

```
OS Name:              Windows 10 Pro 64-bit
OS Version:           Build 10.0.19044
System Manufacturer: FUJITSU
System Model:         LIFEBOOK A555
CPU:                  Intel Core i3 5005U @ 2.00GHz
RAM:                  8GB Dual-Channel DDR3
Storage:              Samsung SSD 870 EVO 500GB (SATA)
```

3 different test cases were defined for the measurement:

1. running the build with all the built-in caching mechanisms turned off,
2. running the build while caching is turned on and changes are made on the codebase after the last build run,
3. running the build while caching is turned on, but without changes on the codebase.

As Dagger can run build in several Docker containers, but Mage does not have a built-in virtualization support, the Dagger build was only ran in the Node.js v16 container for the measurement.

Table 2. Measured build times

	Dagger			**Mage**		
	#1	#2	#3	#1	#2	#3
Without cache	102.814s	100.339s	104.547s	67.648s	63.698s	67.095s
With cache with modifications	37.025s	36.882s	37.164s	58.575s	55.963s	57.605s
With cache without modifications	3.912s	3.914s	3.403s	51.559s	49.832s	50.329s

Table 2 shows the measured build times under different circumstances. The measurement was performed 3 times in each of the configurations, the averages of these values are shown on diagrams below.

5.9.1. Build time without cache

In the first test case, the build time was measured with all the caching mechanisms turned off. For Dagger, the locally available images and containers are pruned, and all the data related to the BuildKit daemon (i.e. "dagger-buildkitd" container and the volume attached to it) was removed too. For Mage, the NPM cache on the host system was cleaned completely. Measured times are shown in Figure 3.

Mage performed better under these circumstances. Mage uses a more lightweight build process than Dagger, while Dagger depends heavily on virtualization. Initialization of the BuildKit daemon, pulling an approx. 300 MB Docker-image, and constructing a container took much more time, than running the build using the host system's shell directly.

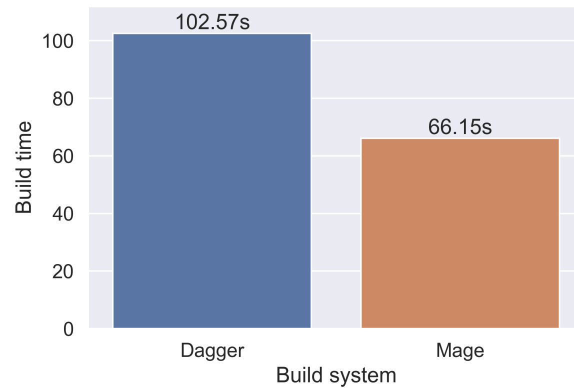

Figure 3. Build times without cache

5.9.2. Build time with cache and code changes

In the second test case, the build time was measured with all the built-in caching mechanisms turned on and slight modifications applied to the code-base. Build times are shown in Figure 4.

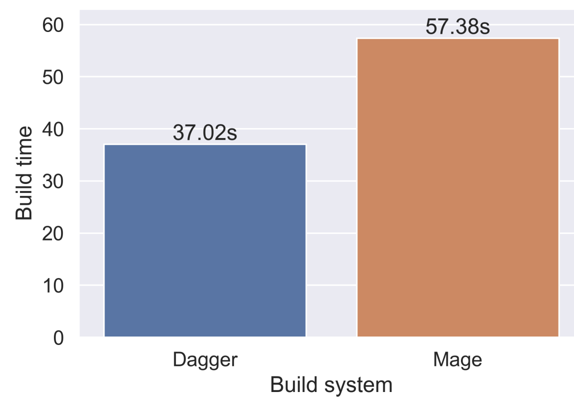

Figure 4. Build times with cache and changes

Dagger performed better in this test case. Pre-fetched Docker images, cached containers and action steps are enabled faster execution of the build compared to Mage.

5.9.3. Build time with cache without changes

In the third test case, the build time was measured with all the built-in caching mechanisms turned on, but without modifications to the project codebase. Results have been presented in Figure 5.

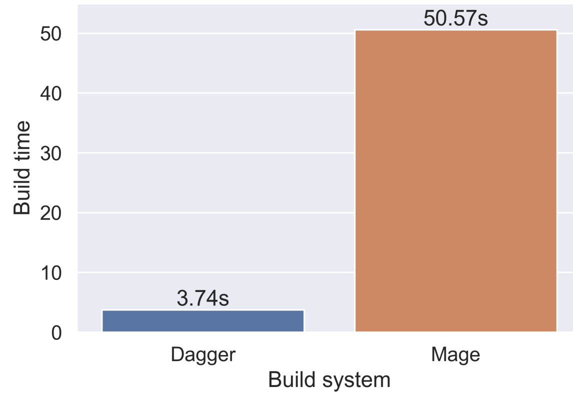

Figure 5. Build times with cache without changes

Dagger performed significantly better in this case. This is because while Dagger can cache the output of all steps of an action based on the running environment's state and the input passed to the command, Mage does not include this level of caching. Mage only uses a binary cache to speed up the compilation of the build pipeline itself. Due to the unchanged codebase and environment, Dagger was able to read the result of installing project dependencies and running the tests from the cache, while Mage performed these steps again from start to finish.

6. Availability of the demonstration project

The complete codebase of the demonstration project is publicly available at [23].

7. Conclusion

I have summarized the risks of vendor lock-in in software projects. Strong dependence on vendor-specific implementation causes risks related to data-, application- and infrastructure portability. It can also cause difficulties for human resource to configure infrastructure provided by a new vendor if the

software does not contain appropriate abstractions to access the used cloud services.

I also identified the factors that may cause close dependence on a specific vendor when using a managed CI/CD service. This type of dependency involves risks related to the reliability and releasing of the software. The risk of losing effective portability comes from different terminology, data structures and formats, execution models and environment definitions used in different managed CI/CD services.

In order to eliminate this problem, I presented an architectural solution that is based on the use of automated build systems and aims to minimize the vendor-specific configuration to mitigate CI lock-in. I examined the applicability of two automated build systems for this purpose. The solution was tested on an open-source NPM package's codebase. The build pipeline was implemented separately with build systems called "Dagger" and "Mage".

The applicability of the build systems was then compared with each other, according to several aspects: building blocks of the pipeline, external dependencies, build isolation, support for defining a build matrix, availability of pre-defined tasks and build time benchmarks.

Regarding the results, it can be stated that any of the examined systems can be used effectively in small to medium sized projects, while in larger projects with complex build pipeline, using Dagger is preferable due to its effective virtualization and caching capabilities.

References

[1] Beslic, A., Bendraou, R., Sopena, J., and Rigolet, J.-Y.: Towards a solution avoiding Vendor Lock-in to enable Migration Between Cloud Platforms. In *MDHPCL 2013 - 2nd International Workshop on Model-Driven Engineering for High Performance and CLoud computing*, Miami, Florida, United States, 2013, pp. 5–14, URL https://hal.archives-ouvertes.fr/hal-01216403.

[2] Kratzke, N. and Quint, P.-C.: Understanding cloud-native applications after 10 years of cloud computing - a systematic mapping study. *Journal of Systems and Software*, **126**, (2017), 1–16, URL https://doi.org/10.1016/j.jss.2017.01.001.

[3] Opara-Martins, J., Sahandi, R., and Tian, F.: Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. *Journal of Cloud Computing*, **5**(1), URL https://doi.org/10.1186/s13677-016-0054-z.

[4] Opara-Martins, J.: Taxonomy of cloud lock-in challenges. In *Mobile Computing - Technology and Applications*, InTech, 2018, URL https://doi.org/10.5772/intechopen.74459.

[5] Prasanth, K. V. V. N., Satyanarayana, K. V. V., Akhila, V., Sahithya, M., and Reddy, A. S.: Implementing a solution to the cloud vendor lock-in using standardized API. *International Journal of Computer Science and Information Security*, **16**(1).

[6] Amato, A., Tasquier, L., and Copie, A.: Addressing the interoperability in cloud: The vendor agent. *Int. J. Comput. Sci. Eng.*, **11**(1), (2015), 5–16, URL https://doi.org/10.1504/IJCSE.2015.071357.

[7] Quint, P.-C. and Kratzke, N.: Overcome Vendor Lock-In by Integrating Already Available Container Technologies Towards Transferability in Cloud Computing for SMEs. In *CLOUD COMPUTING 2016: The Seventh International Conference on Cloud Computing, GRIDs, and Virtualization*, ISBN 978-1-61208-460-2, 2016.

[8] Opara-Martins, J.: A decision framework to mitigate vendor lock-in risks in cloud (saas category) migration., 2017, URL http://eprints.bournemouth.ac.uk/29907/.

[9] Gruhn, V., Hannebauer, C., and John, C.: Security of public continuous integration services. In *Proceedings of the 9th International Symposium on Open Collaboration*, ACM, 2013, URL https://doi.org/10.1145/2491055.2491070.

[10] Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D.: Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2016, URL https://doi.org/10.1145/2970276.2970358.

[11] Jenkins: *Pipeline Syntax*. URL https://www.jenkins.io/doc/book/pipeline/syntax/. Accessed 2022-10-14.

[12] GitHub Actions: *Documentation*. URL https://docs.github.com/en/actions. Accessed 2022-10-14.

[13] Microsoft Learn: *Key concepts for new Azure Pipelines users*. URL https://learn.microsoft.com/en-us/azure/devops/pipelines/get-started/key-pipelines-concepts. Accessed 2022-10-14.

[14] Travis CI: *Core concepts for beginners*. URL https://docs.travis-ci.com/user/for-beginners/. Accessed 2022-10-14.

[15] AppVeyor: *Documentation*. URL https://www.appveyor.com/docs/. Accessed 2022-10-14.

[16] Google: *Bazel*. URL https://bazel.build/. Accessed 2022-10-16.

[17] Facebook: *Buck*. URL https://buck.build/. Accessed 2022-10-16.

[18] Dagger: *Dagger*. URL https://docs.dagger.io/. Accessed 2022-10-16.

[19] Mage: *Mage*. URL https://magefile.org/. Accessed 2022-10-16.

[20] Kiss Áron: *aron123/node-barion (GitHub)*. URL https://github.com/aron123/node-barion. Accessed 2022-10-18.

[21] Dagger: *dagger/pkg/universe.dagger.io*. URL https://github.com/dagger/dagger/tree/v0.2.36/pkg/universe.dagger.io. Accessed 2022-10-23.

[22] Go: *Developing and publishing modules.* URL https://go.dev/doc/modules/developing. Accessed 2022-10-23.

[23] Kiss Áron: *aron123/ci-lock-in (GitHub).* URL https://doi.org/10.5281/zenodo.7244304.