

# Cultivating an oForest—Operating a Scalable Experimentation Infrastructure for oTree

Karlsruher Institut für Technologie (KIT)  
Institut für angewandte Betriebswirtschaftslehre und Unternehmensführung  
Lehrstuhl für Human Resource Management

**Prof. Dr. Petra Nieken**  
**Dr. Paul M. Gorny**

Wintersemester 2022/2023

Vorgelegt von:  
**Jasper Anders**  
2000059  
uyxzd@student.kit.edu  
5. Mastersemester

**Karlsruhe, 31.03.23**

## **Abstract**

In this work, we introduce oForest, a tool that streamlines and automates the deployment of oTree experiments on self-hosted infrastructure. oTree is a popular Python library for creating digital, interactive and online capable experiments for research in behavioral economics. Due to conflicts with the European General Data Protection Regulation, it has become difficult to deploy and host these oTree experiments with US cloud providers such as Heroku. Ideally, research institutions should provide their researchers with infrastructure, but they often don't have the resources and capabilities to do so. oForest gives institutions a fast way of creating an environment where researchers can deploy and conduct digital experiments easily. oForest uses containers, container orchestration, and continuous integration pipelines, to build an simple to use solution, where researchers can act autonomously and quickly.

# Contents

List of Figures	IV
List of Tables	V
<b>1 How Digital Infrastructure Enables Behavioral Economics</b>	<b>1</b>
<b>2 The Importance of Digital Experiments in Behavioral Economics</b>	<b>3</b>
2.1 Reproducibility is Trust . . . . .	4
2.2 Digitized Experiments . . . . .	4
2.3 Legal Implications of Running Experiments . . . . .	9
<b>3 An Introduction to Modern Infrastructure Tooling</b>	<b>10</b>
3.1 Containerization Technology . . . . .	11
3.2 Container Orchestration . . . . .	14
3.3 Automation Tooling . . . . .	19
3.4 Linux Servers . . . . .	20
<b>4 Designing Experimentation Infrastructure</b>	<b>21</b>
<b>5 The oForest Components</b>	<b>24</b>
5.1 Containerizing oTree . . . . .	25
5.2 Cluster Configurations . . . . .	27
5.3 GitLab as the oForest User Interface . . . . .	33
5.4 The CI / CD Pipeline . . . . .	34
5.5 Server Setup . . . . .	39
5.6 Security and Privacy . . . . .	41
5.7 Longevity and Maintainability . . . . .	42
5.8 Extensibility . . . . .	43
5.9 Limits of oForest and Further Work . . . . .	43
<b>6 Conclusion</b>	<b>44</b>
<b>References</b>	<b>46</b>
<b>Eidesstattliche Erklärung</b>	<b>54</b>

## List of Figures

1	The outdated design of z-Tree. . . . .	6
2	Conceptual Overview of oTree . . . . .	8
3	The DevOps Approach . . . . .	10
4	Architecture comparison between virtual machines and containers . . . . .	12
5	Intuition: How interactions with oForest work . . . . .	25
6	Interaction of components inside the cluster. . . . .	27
7	oTree experiment architecture inside the cluster . . . . .	29
8	The Grafana dashboard . . . . .	32
9	GitLab’s user management interface. . . . .	33
10	oForest GitLab Project structure . . . . .	34
11	An overview of the oForest pipeline interface . . . . .	34
12	A failed acceptance test stage. . . . .	36
13	GitLab runner status as seen in the oForest group settings. . . . .	38

## List of Tables

1	System Load of selected Components. . . . .	41
2	Literature Table for Economical Sources, Part I . . . . .	50
3	Literature Table for Economical Sources, Part II . . . . .	51
4	Literature Table for Economical Sources, Part III . . . . .	52
5	Literature Table for Technical Sources . . . . .	53

# 1 How Digital Infrastructure Enables Behavioral Economics

The field of behavioral economics studies the psychological and social factors that influence the economic decision-making of individuals (Beck, 2014, pp. 9). Today, it affects almost all fields of economics. It is relevant in Game Theory (Durlauf und Blume, 2010, p. 23, 42), finance (Durlauf und Blume, 2010, p. 32, 178), and market and mechanism design (Falk und Fehr, 2003; Durlauf und Blume, 2010, p. 32, 178). Besides academic applications, results of behavioral economics research have been used to find the best auction mechanism to distribute telecommunication frequencies, to find the preferences of people for better advertisement, to find how to optimally distribute parking or landing sites, or the most profitable way to sell off bonds (Beck, 2014, p. 15). In macroeconomics behavioral studies have examined the dynamics of bank runs (Diamond und Dybvig, 1983) and monetary policy decision-making (Engle-Warnick und Turdaliiev, 2010). Further it is used by consulting firms such as Delloitte<sup>1</sup> and TWS<sup>2</sup>.

Much of the groundbreaking research in behavioral economics, such as Kahneman und Tversky (1979), Thaler und Sunstein (2008), and Bielefeld (1988), use experimentation to generate insight. These experiments are conducted either in the laboratory or in the field. Whereas laboratory experiments try to emulate an environment to observe behavior, field experiments try to collect data in a more natural setting. Specifically for lab experiments, Levitt und List (2007) find multiple unintended factors that influence the behavior of study participants. The limits of the laboratory setting and their more flexible nature, have made field experiments increasingly popular. Chen et al. (2016) find field experiments to bridge the gap between naturally occurring data and data that was generated in a lab.

Today, software almost always aids in creating and conducting experiments. It allows researchers to reach larger groups of participants, simplifies data collection and analysis, and can increase experiment reproducibility (Baker, 2016). Further, during the Covid-19 pandemic, the ability to conduct experiments online has proven valuable. These online capabilities also allow researchers such as Chen et al. (2016) to access crowd-sourcing platforms such as Amazon M-Turk.

oTree is one of the most popular tools for developing economic experiments in behavioral research.<sup>3</sup> Although building these experiments is relatively simple, deploying them

---

<sup>1</sup><https://www2.deloitte.com/de/de/pages/innovation/contents/innovationsmanagement-verhaltensoekonomie.html>

<sup>2</sup><https://www.tws-partners.com/de/loesungen/verhaltensoekonomie/>

<sup>3</sup>[https://scholar.google.de/scholar?cites=5704587714172176698&as\\_sdt=2005&scioldt=0,5&hl=en](https://scholar.google.de/scholar?cites=5704587714172176698&as_sdt=2005&scioldt=0,5&hl=en)

manually can be more complex. Therefore, oTree recommends using the US-based cloud provider Heroku.<sup>4</sup> For researchers in the European Union, who are processing personal information in their studies, this approach is conflicting with the General Data Protection Regulation (GDPR). The Court of Justice of the European Union ruled the basis of data transfer to the US without “supplementary measures” as illegitimate.<sup>5</sup> Such measures can include encryption and require additional development efforts, which results in researchers looking for alternative ways of hosting their experiments.

Unfortunately, maintaining their own infrastructure often is too complex for the individual researcher. Having an economics background, they often lack the know-how and proficiency in administrating servers, networking, building applications for production, and managing databases. For research institutions, it can thus make sense to centralize this complexity and provide a digital infrastructure for the deployment of experiments. Personal data would only ever be processed by the institution itself and never by a third party, which reduces complexity as no *joint controller contracts* (Art. 6, 26 GDPR) need to be negotiated. Also, the data stays within the European Union and thus is not tangent by the court ruling.

To provide such a deployment environment, institutions currently would need to build this infrastructure from scratch. This can be an obstacle as it requires time, effort, and know-how. In this work, we introduce oForest, a tool that streamlines and automates the deployment of oTree experiments on self-hosted infrastructure. oForest allows the deployment of many oTree experiments with different requirements on the same server. As oForest centralizes complexity, it allows researchers to focus on their research instead of operating servers. Built with automation in mind, oForest makes it easy to deploy new applications. Setting up the oForest experimentation infrastructure requires only a few simple steps. oForest enables a simple and reliable way to conduct research in the field of behavioral economics and beyond.

In this work, we start by introducing the foundations of behavioral economics research and then explore how digital infrastructure provides the necessary tools for conducting experiments. On a more technical side, we discuss the driving technologies behind modern digital infrastructure such as containerization technology, container orchestration, and automation tooling.

With the foundations laid, we move on to exploring the requirements of oForest. As we have chosen a user-centered design approach, we explore the user requirements by the

---

<sup>4</sup><https://otree.readthedocs.io/en/latest/server/heroku.html?highlight=heroku#basic-server-setup-heroku>

<sup>5</sup>[https://edpb.europa.eu/sites/default/files/files/file1/20200724\\_edpb\\_faqoncjuec31118\\_en.pdf](https://edpb.europa.eu/sites/default/files/files/file1/20200724_edpb_faqoncjuec31118_en.pdf)

means of creating personas for crucial user groups.

In the main section of our work, we describe the oForest components and how they interact with each other in detail. These include the building of oTree images, the inner workings of the deployment pipeline, how cluster components interact and are configured, and what to consider when provisioning a server. Further, we provide some remarks about security, maintainability, and longevity. We close our work by looking at the limitations of oForest, areas where further work might be needed, and a more general conclusion.

## 2 The Importance of Digital Experiments in Behavioral Economics

Experimentation in behavioral economics dates back to at least 1959, in their work [Sauer-  
mann und Selten \(1959\)](#) chose to make use of an experimental setting to better understand human behavior and with this understanding improve their model of oligopolies. Contrary to a model solely based on rational agents, they describe the strength of controlled experiments in their ability to create a transparent setting where actual behaviors are well observable. Today, experimentation is an integral part of economic research and the results have been awarded several Nobel Prizes. Among the winners are Daniel Kahnemann and Richard Thaler for their *Prospect-* ([Kahneman und Tversky, 1979](#)) and *Nudge-Theory* ([Thaler und Sunstein, 2008](#)) respectively. Their works have also found their way into the general public by the means of two best-selling books. The books by [Kahneman \(2011\)](#) and [Thaler und Sunstein \(2008\)](#) have both been sold over 1.5 million times each ([Shotton, 2014](#); [Penguin Randomhouse, 2023](#)).

Another important character in experimental economics is Vernon Smith. He also received a Nobel Prize “for having established laboratory experiments as a tool in empirical economic analysis, especially in the study of alternative market mechanisms”.<sup>6</sup> According to [Smith \(1994\)](#), all types of experiments fulfill some common functions:

1. Test a theory, or discriminate between theories
2. Explore the causes of a theory’s failure
3. Establish empirical regularities as a basis for a new theory
4. Compare environments
5. Compare institutions
6. Evaluate policy proposals
7. Test institutional design

---

<sup>6</sup><https://www.nobelprize.org/prizes/lists/all-prizes-in-economic-sciences/>



The fundamental nature of these functions shows us how important experiments really are. Further, they can be categorized by the environment they are taken in. According to Hanno Beck ([Beck, 2014](#), pp. 16) there are laboratory, field and natural experiments. They differ not only in how they are carried out but also in what kind of data they utilize. In a laboratory setting, researchers aim to emulate a situation in an artificial environment, where participants are aware that they are taking part in a study. In the field, results are generated in a more natural setting. Here participants often don't know they are being monitored. Lastly, the natural experiments differ from the other two as it uses naturally occurring data to test a hypothesis.

## 2.1 Reproducibility is Trust

Reproducibility has always played a big role in experimental research. [Harrison und List \(2004\)](#) write, “in some sense, every empirical researcher is reporting the results of an experiment.” The value of research diminishes if the results cannot be replicated. An increasing inability to reproduce results has coined the term *Reproducibility Crisis*. In a survey, [Baker \(2016\)](#) found that 70% of researchers across all fields have tried to reproduce an experiment, but failed to get the same results. Further, 50% of researchers couldn't even reproduce their own experiments. Around 1500 researchers were questioned and 52% agree there is a significant crisis going on. Another 38% agree that there is at least a slight crisis.

Besides selective reporting, the pressure to publish, and many other reasons, [Baker \(2016\)](#) notes that the “unavailability of methods or code” is seen as a contributor to irreproducible research by close to 80% (45% say it always contributes, 35% say it sometimes contributes) of researchers.

Whereas Baker's results apply to all areas of research, a study by [Camerer et al. \(2016\)](#) confirms this general notion for the field of economics. They looked to replicate laboratory experiments and, on average, were able to only replicate 66% of the effect size of the original study. In their conclusion, they call for well-designed and documented experiments with replication in mind. This documentation goes hand in hand with the publication of data and the means of collecting this data.

## 2.2 Digitized Experiments

Given the importance of experimentation in behavioral economics, the question of how these experiments are actually conducted arises naturally. Prior to the widespread use of computers, paper surveys and manual data collection were the preferred tools. However, advances in technology have significantly impacted how experiments are carried out.

Digitized experiments can aid in the reproducibility of results. Because of their digital na-

ture, experiment data, software artifacts, and code can easily be shared among researchers and reused if needed. By casting the experimental design into code, reproducing the exact setup becomes unambiguous and allows other researchers to build upon previous work more easily.

Further, the way in which experiments are conducted often co-determines their scope. For example, surveys are unfit for investigating behavior in interaction. Also, the use of digital experiments eases data collection and evaluation, makes experiments scalable to hundreds of participants, allows for a more granular assignment of treatments, and they can even make lab experiments available in the field. The last point is of particular interest as there is an ongoing debate on how well findings in the lab are extrapolated to the real world. [Levitt und List \(2007\)](#) find five other factors, that influence a participant’s behavior beyond the monetary incentive set by the lab. Among these factors are “the presence of moral and ethical considerations”, “self-selection of the individuals making the decisions” and “the stakes of the game”.

Today, software tools allow researchers to set up ad hoc experiments in the field. In their introduction to oTree, a tool for creating digitized and interactive experiments, [Camerer et al. \(2016\)](#) argue that experimentation in the field bridges the gap between data that was generated in a laboratory and data that occurred naturally. The ubiquity of mobile devices and tablets together with digital experimentation platforms have made field experiments more accessible than ever. To better understand the advantages of a tool like oTree it is essential to set them into the context of earlier approaches to building digitized experiments.

### **2.2.1 z-Tree**

For a long time, developing software for experimental research required significant programming experience ([Fischbacher, 1999](#)). As a reaction, Urs Fischbacher introduced z-Tree (Zürich Toolbox for Readymade Experiments), a tool that made building experiments without specialized coding knowledge feasible. The z-Tree program provides its users with a mixture of graphical interfaces and the ability to write simple code snippets. Due to its simplicity, experiments can be created quickly. The resulting experiments are saved in a z-Tree-specific file format that allows for the rudimentary sharing of experiments via e.g. email. During the experiments, study participants are presented with interfaces that allow for both surveys and interaction studies. Data that is generated with z-Tree can be exported as a CSV file and then processed further. Although z-Tree still is a popular tool, it has some drawbacks that originate both in its software architecture and in its age.

In their basic form, z-Tree applications are bound to run experiments in a laboratory setting, due to its network architecture. z-Tree acts as a server and communicates with

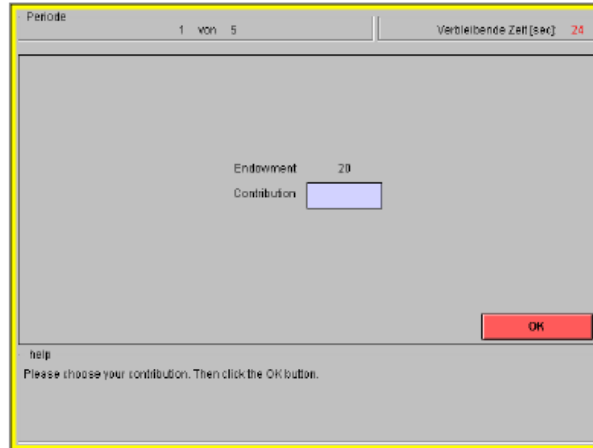


Figure 1: The outdated design of z-Tree.

the participants through a client called z-Leaf. The z-Leaf clients discover the server by a statically assigned IP address. This allows z-Tree to run in a local network without a dedicated file or database server. However, it limits z-Tree’s use outside the lab as this would require setting up a local network and installing z-Leaf clients on every participant’s device. In almost all cases this is not feasible, rendering z-Tree inflexible.

Despite the architectural limitations, z-Tree also lacks some other crucial capabilities. For instance, collaboration and version control with tools such as git is not possible. Another thing to note is, that z-Tree is distributed under a proprietary (though free) license. The closed-source code and lack of documented APIs hinder the development of extensions and modifications. Further, z-Tree was first introduced in 1999 and the user interface is reminiscent of the time. As can be seen in Figure 1, the interface looks outdated and does not match today’s usability standards. This is true both for the interface used by researchers and the interface encountered by participants. Lastly, z-Tree is single-threaded, which can heavily impact performance, especially in larger experiments (Fischbacher, 1999).

### 2.2.2 oTree

oTree was introduced by Chen et al. (2016). It is the spiritual successor to z-Tree and aims to fix the previously discussed shortcomings. oTree replaces the z for Zürich with an *o* and aims to be “open-source, object-oriented and online” (Chen et al., 2016). Although researchers can build experiments with little to no code using a graphical interface on oTree Hub (2023), most researchers will probably interact with oTree through coding in Python.

**Python.** Python is a popular high-level programming language that has gained immense popularity in recent years, thanks to its simplicity, readability, and versatility. It is an open-source language, which means it’s freely available to everyone and has a vast

community of users and developers who contribute to its development. Python’s ubiquity in science stems from its ability to handle large and complex datasets, which are common in scientific research. Scientists commonly use Python to write scripts and applications for data analysis, visualization, machine learning, and other tasks. Further, numerous libraries and packages that facilitate scientific computing, such as NumPy, Pandas, SciPy, and Matplotlib are available. The developer experience is also great as there are plenty of tutorials, bootstrapped projects and mature editors such as PyCharm.

As a Python library, oTree can leverage not only the wide capabilities of the language itself but also the vast ecosystem of tools and resources surrounding it. oTree’s usage of Python also has advantages in terms of collaboration, maintainability, extensibility, and testability. Multiple researchers can work on the same experiment by making use of tools like GitHub or GitLab. In section 2.1 we talked about how sharing of code can improve the reproducibility of research. Using a familiar language enables other researchers not only to re-run experiments but also to better understand how the experiment is built.

Because oTree is open-source, it is also easier to adapt it to specific use cases of individual researchers. For example, monitoring biosensor output previously was impossible with z-Tree, but with oTree it’s possible to allow researchers to incorporate completely novel data sources. Beyond the Python community, oTree users can also connect, and exchange experiences on a forum<sup>7</sup> or share their code using [oTree Hub](#).

One of the biggest advancements of oTree is its use of modern web technologies. The performance and capabilities of web applications today are on par with native applications but have the advantage of being cross-platform. Further, their usage does not require the installation of external clients. Compared to z-Tree, experiments are no longer bound to a single laboratory but become portable. By using iPads or Laptops, experimenters can conduct field experiments. With the experiments being available via the internet, researchers don’t even need their subjects to be physically present, and experiments can be conducted fully online. [Chen et al. \(2016\)](#) even report the integration of [Amazon M-Turk](#), a “crowdsourcing marketplace that makes it easier for individuals and businesses to outsource their processes and jobs to a distributed workforce who can perform these tasks virtually”.

oTree has become a popular tool. As its usage requires the citation of [Chen et al. \(2016\)](#), the citation count is a good proxy for its adoption. According to Google Scholar, more than 1300 papers have been published using oTree. In 2022 alone it were 478 citations<sup>8</sup>.

---

<sup>7</sup><https://www.otreehub.com/forum/>

<sup>8</sup>[https://scholar.google.de/scholar?as\\_ylo=2022&hl=de&as\\_sdt=2005&scioldt=0,5&cites=5704587714172176698&scipsc=](https://scholar.google.de/scholar?as_ylo=2022&hl=de&as_sdt=2005&scioldt=0,5&cites=5704587714172176698&scipsc=)

**Conceptual Overview of oTree.** Experiments in oTree are organized in sessions in which multiple participants can take part. A session consists of a series of subsessions, where each subsession in turn is a sequence of pages. In each subsession, one or multiple groups of players can take part. A player is a participant in one particular subsession. As seen in Figure 2, the oTree objects are arranged hierarchically, with the session at the top, followed by the subsession, the group, and the player. Each object can access all of its parent’s properties, making it easy to create and organize complex, multistage experiments with various participants.<sup>9</sup>

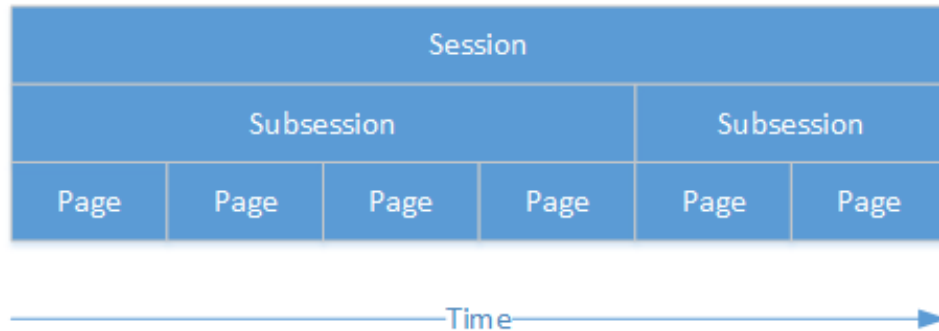


Figure 2: Conceptual Overview of oTree

A finished oTree experiment is a web application that must be deployed to a server.<sup>10</sup> As mentioned in the introduction, many researchers don’t host experiments themselves, but use Heroku to deploy their applications. Although one can also use other cloud providers, Heroku is endorsed by oTree and is thus an obvious choice.<sup>11</sup> Through the platform oTree Hub<sup>12</sup>, researchers can even deploy their experiments directly to Heroku. In most cases, deploying experiments with Heroku is straightforward, and it has even been free, as long as you stayed within certain boundaries.

### 2.2.3 Alternatives to oTree

Besides oTree and z-Tree there are of course many other tools that help in conducting economic research. Only to name a few there are Wextor (Reips und Neuhaus, 2002), the MIT Seaweed project (Chilton et al., 2009), Breadboard (McKnight und Christakis, 2016), and nodeGame (Baliatti, 2017). All of these offer a similar feature set. They offer experiments built on web technologies like HTML5 and JavaScript, synchronous and interactive experiments in the lab or online. At the time of this writing, none of the alternatives has achieved the same popularity as oTree.

<sup>9</sup>[https://otree.readthedocs.io/en/latest/conceptual\\_overview.html](https://otree.readthedocs.io/en/latest/conceptual_overview.html)

<sup>10</sup><https://otree.readthedocs.io/en/latest/server/intro.html>

<sup>11</sup><https://otree.readthedocs.io/en/latest/server/heroku.html#heroku-setup>

<sup>12</sup><https://www.otreehub.com/>

It thus makes sense that we focus on oTree in this work. Still, the tools we use and the artifact we propose are flexible enough to be adopted to suit other experimentation frameworks, if the need arises.

## 2.3 Legal Implications of Running Experiments

In most behavioral experiments, a wide range of data, such as age, gender, income, etc. is collected. Most of this information is related to an either identified or identifiable natural person and thus is, according to Art. 4, No. 1 GDPR, considered *personal data*. As soon as a subject handles this data inside the EU, it needs to adhere to the General Data Protection Regulations (GDPR). Generally, this is not a big deal, as the research institution must only obtain the processing consent of the participants, which is a standard procedure.

Unfortunately, problems arise as soon as a researcher wants to use a third-party hosting service. According to law, this service now acts either as a *joint controller* (Art. 26 GDPR) or as a data *processor* (Art. 28 GDPR). In both cases, a contract on the basis of cooperation must be concluded. This can be either a custom contract or based on the standard contract clauses (SCC) provided by the European Union. Without such a contract, the transmission to third parties of any data is illegitimate.

Research institutions of small size are unlikely able to negotiate such a custom contract with any of the large cloud providers. Thus, the collaboration must be based on the SCCs. But since the Schrems II ruling (Legal Matter C-311/18)<sup>13</sup>, the Court of Justice of the European Union deemed SCCs only valid under special conditions. The researcher now has to ensure an adequate level of protection, e.g. taking special measures such as encryption of data before it is transmitted. This would mean that for a lawful usage of e.g. Heroku, a researcher must encrypt all personal data before it is sent to the US, additional work that is unfeasible for most.

The goal of oForest is to get rid of such a *processor* altogether. By providing an infrastructure that is simple in use and setup, institutions can self-host experiments and don't need to rely on third parties. This simplifies the legislative requirements, as now the processing of experimentation data is just embedded into the existing privacy measures of the organization's infrastructure. In other words, the standard privacy consent clause, which participants sign anyway, is enough to cover legislative requirements.

---

<sup>13</sup>[https://www.europarl.europa.eu/RegData/etudes/ATAG/2020/652073/EPRS\\_ATA\(2020\)652073\\_EN.pdf](https://www.europarl.europa.eu/RegData/etudes/ATAG/2020/652073/EPRS_ATA(2020)652073_EN.pdf)

### 3 An Introduction to Modern Infrastructure Tooling

As soon as researchers can no longer outsource the hosting of oTree experiments, they find themselves in the game of software development and delivery. They must program their application, run tests on it, and need to assure that experiments work reliably and safely while subjects are participating. With oForest we try to remove this complexity for individual researchers. To understand the architecture of oForest, it is important to introduce its building blocks first.

In building oForest, we made sure to respect modern infrastructure best practices. This will make it easier for other developers or researchers to quickly understand how oForest works and how to maintain it. Today, the infrastructure landscape is shaped by the DevOps movement. Before this movement, professionals in software development and operations were working in different departments, often with opposite objectives and performance indicators (Kim et al., 2021, pp. 28). While developers and product managers are incentivized to move fast and react quickly to changing markets and requirements, the operations department is concerned with running systems that are as stable, reliable, and safe as possible. This mismatch of incentives creates lots of friction and ultimately leads to a downward trend in overall performance (Kim et al., 2021, pp. 29).

The DevOps approach now tries to combine software development and IT operations by improving collaboration and communication between these two groups. The main idea of DevOps is to shorten the development cycles, increase deployment frequency, and thus bring development and operations closer together. Besides rolling out new features more quickly, this also enables teams to fix bugs or deploy patches with increased speed. In this way, high-quality software products can be delivered more rapidly, while still keeping software stable and reliable.

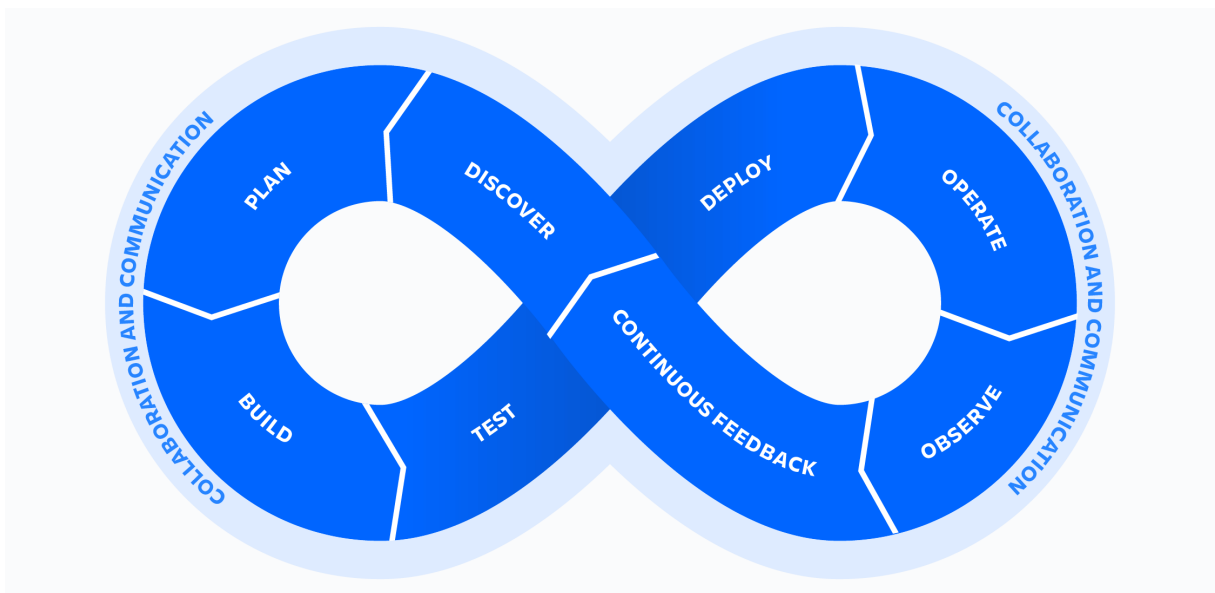


Figure 3: The DevOps Approach<sup>14</sup>



DevOps focuses on a continuous cycle of planning, building, testing, deploying, operating, observing, gathering feedback, and discovering areas to improve. This cycle requires a high level of communication and collaboration, but above all a high level of automation. This automation effort eliminates manual errors and speeds up the development process in general, it includes automating tasks such as bundling applications for production, testing, and deploying applications. Especially, the demand for automation shaped the tools we use to build oForest.

When we set the DevOps cycle in the context of research, most would agree that researchers should use their time looking for interesting questions to answer (discover) or figuring out how to test their hypothesis (plan), instead of trying to figure out how to deploy and reliably operate a web application. This is exactly where researchers benefit from the automation efforts of the DevOps movement. They enable hundreds of researchers to *discover* and *plan* while automation takes care of testing, deploying, operating, and observing.

### 3.1 Containerization Technology

To be able to run multiple oTree experiments in an automated manner, we need each application to work consistently across multiple environments. For this, we can make use of software containers. One of the leading container runtime providers defines a container as “a standard unit of software that packages up code and all its dependencies, so the application runs quickly and reliably from one computing environment to another”.<sup>15</sup> Containerization technologies are software tools that help developers to build, package, and run containers. On a host, containers run in an isolated and consistently reproducible environment and thus can run across multiple machines, operating systems, and platforms. This means an oTree container will run just as well on the computer of a researcher as on the web server where it is deployed, and also on the computer of another researcher who wants to reproduce the results of the experiment.

In concept, containers are similar to a virtual machine, as they both create an isolated environment for an application. Still, they differ in the way they achieve this isolation. Whereas a virtual machine uses a *Hypervisor* to virtualize resources such as CPU, memory, and disc storage and on top runs a completely independent operating system, a container utilizes large parts, e.g. the kernel and various drivers, of the host system it’s running on (Schneller und Pustina, 2015). A visual comparison can be found in Figure 4.

---

<sup>14</sup><https://www.atlassian.com/devops>

<sup>15</sup><https://www.docker.com/resources/what-container/>

<sup>16</sup><https://www.docker.com/resources/what-container/> In section “Containers and Virtual Machines Together”



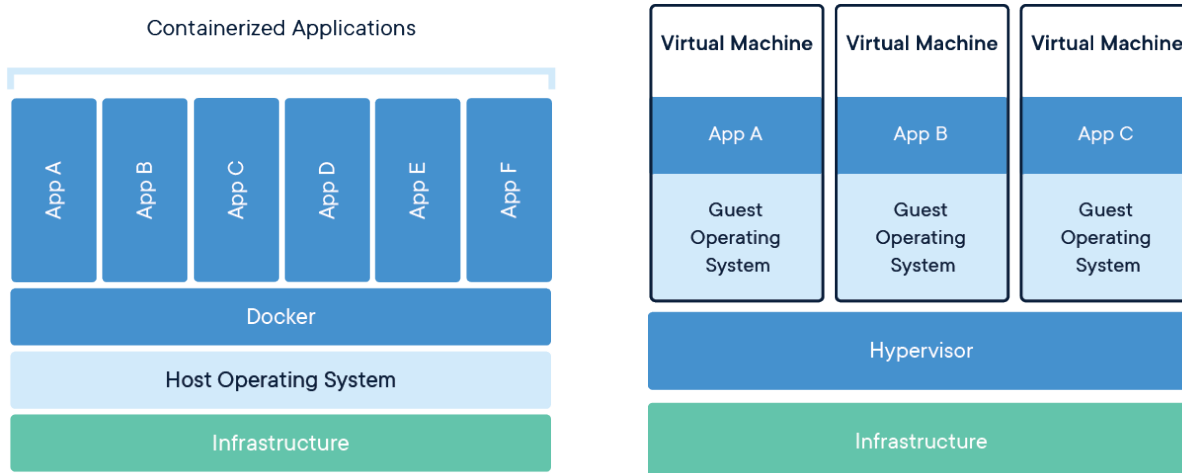


Figure 4: Architecture comparison between virtual machines and containers<sup>16</sup>

This makes containers far lighter than virtual machines. Where virtual machines can take up to tens of gigabytes of disc space, containers typically only take up a couple of megabytes of space. As a result, we can run many containers on a single machine, without sacrificing too much performance.

### 3.1.1 The Container Lifecycle

In the following, we will illustrate some of the terminology surrounding containers, for this we will be using Docker, one of the most popular container runtimes.

Running a container is straightforward. Consider the following command that runs a simple example:<sup>17</sup>

```
docker run hello-world
```

```
> Hello from Docker!
```

```
> This message shows that your installation
    appears to be working correctly.
```

The container starts, prints out a message and stops after it is done. In this example, **hello-world** is the name of a so-called image. This image specifies the contents of a container, including dependencies and code. Starting such an image will always deem the same results, no matter the environment. Executing the **run** command starts a container from the specified image. An image can be seen as a snapshot of a container, whereas the container is a running instance of an image.<sup>18</sup>

<sup>17</sup><https://docs.docker.com/engine/reference/commandline/run/>

<sup>18</sup><https://docs.docker.com/get-started/>

When starting a container, the container runtime, in our case Docker, tries to either find the specified image locally or tries to pull it from a registry. A registry can be seen as a database that stores many different images and makes them available to other developers. These registries can be both public and private. We can tell Docker to explicitly pull an image by running the following command:<sup>19</sup>

```
docker pull hello-world
```

This makes an image now available locally. When executing the `run` command, Docker won't need to pull the image again from the remote registry.

When developing a new oTree experiment, researchers may want to make it available to other researchers on a registry. For this, they first need to build an image from the experimentation code using the `build` command. Using a configuration file called `Dockerfile` or `Containerfile`, they need to specify the build instructions. A `Dockerfile` contains a base image, and further describes multiple other steps that need to be executed in the build process. These steps can include installing dependencies, copying source code into the container, and specifying commands to be run on the container startup.<sup>20</sup> A simple `Dockerfile` for an oTree image might look like this:

```
FROM python:latest
RUN mkdir /src
WORKDIR /src
COPY ./src/ /src/
RUN pip install -r requirements.txt
EXPOSE 8000
CMD [ "otree", "prodserver", "8000"]
```

First, the `Python:latest` image is chosen as a base image, enabling containers to run Python code. Then a new directory called `src` is created in the container, set as working directory, and all the source code is copied to this new folder. Next, all dependencies specified in the `requirements.txt` are installed. In the following step, the internal port 8000 is exposed. As mentioned, containers run in an isolated environment, which means they also have their own networking, in particular, they also have their own `localhost`. If we start an application that serves some content to `localhost:8000`, we wouldn't be able to access this content unless we exposed that port to the host system.<sup>21</sup> In the last step, we specify the command to be run when the container is started. In the above case, the oTree production server is started, by running `otree prodserver 8000` on the

---

<sup>19</sup><https://docs.docker.com/engine/reference/commandline/pull/>

<sup>20</sup>[https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)

<sup>21</sup>[https://docs.docker.com/develop/develop-images/dockerfile\\_best-practices/](https://docs.docker.com/develop/develop-images/dockerfile_best-practices/)

creation of the container. After building an image, it can be pushed to a registry<sup>22</sup>, where other developers can then pull the image to run it themselves.

```
docker build Dockerfile
docker push dockerhub.io/my_image
```

### 3.1.2 Containers are Ephemeral

An important thing to point out is the ephemeral nature of containers. As containers are created from static images, they can't hold any persistent data.<sup>23</sup> If you were to write data to a file inside a container and restart said container, the data will be lost. Although inconvenient at first glance, this is a desirable behavior that enables containers to be reproducible.

Of course, data persistence is necessary for many circumstances. In these cases, persistent volumes, such as local files, can be mounted to the corresponding containers. Typically, data persistence is separated from application logic. For our case, this means we will run stateless experiment containers that connect to their own database containers, which in turn have a persistent file system mounted to them.

### 3.1.3 Alternatives to Docker

In 2015 Docker, together with other industry leaders, launched the [Open Container Initiative](#) (OCI) which standardized the *runtime*, the *image*, and the *distribution* of containerized applications. This standardization effort is not only great news for the longevity of containerization, but also enabled the emergence of tooling besides Docker.

For example, Kubernetes, the Container Orchestration tool we will talk about in the next section, does not use Docker anymore.<sup>24</sup> Kubernetes now runs on a more lightweight runtime alternative to Docker called [Cri-o](#). Further, there are alternatives that act as a drop-in-replacement to Docker, like [Podman](#). Although containers are almost always named in the same breath as docker, today they are no longer bound together.

## 3.2 Container Orchestration

In the previous section, we discussed the power of containerized applications. Managing a multitude of containers manually becomes unfeasible quite fast. At this point, we can make use of container orchestration tools such as Kubernetes ([Kubernetes](#)) or Docker Swarm<sup>25</sup>. [Red Hat](#) defines container orchestration as follows:

---

<sup>22</sup><https://docs.docker.com/engine/reference/commandline/push/>

<sup>23</sup><https://docs.docker.com/storage/>

<sup>24</sup><https://kubernetes.io/blog/2020/12/02/dont-panic-kubernetes-and-docker/>

<sup>25</sup><https://docs.docker.com/engine/swarm/>

“Container orchestration automates the deployment, management, scaling, and networking of containers. [... It] can be used in any environment where you use containers. It can help you to deploy the same application across different environments without needing to redesign it. And microservices in containers make it easier to orchestrate services, including storage, networking, and security.”

In the oForest environment, we deploy and remove experiments frequently. These deployments do not only rely on a single container but also on a database, an Ingress resource, and on an SSL certificate issuer. In such a volatile environment, the automation capabilities of Kubernetes are essential.

### 3.2.1 Infrastructure as Code

Kubernetes and many other modern infrastructure tools work in a declarative fashion. Declarative definitions focus on describing the desired outcome or state of a system without specifying how to achieve it. In other words, declarative definitions tell a system what needs to be done, but not how to do it. These definitions are called manifests and are often written in configuration languages like YAML or JSON. For example, a declarative definition might specify that a server should have a specific package installed, but does not directly tell the server to run an installation command directly. The opposite, imperative definitions can be more granular and provide more control over how a system is built. However, they can also be more difficult to manage and maintain over time. Declarative definitions are often easier to read and understand and can be more flexible and adaptable. That’s why declarative approaches have become popular in a movement known as *Infrastructure as Code*. In it, code is used to describe and manage infrastructure. This not only opens up great automation potential but also gives administrators a comprehensible overview of the state of their system. Instead of trying to understand in what state a system ends up after some code has been executed, they can consult the declarative state definitions of a system.

In the oForest setting, using such a declarative approach is especially useful as it can assure a consistent and reproducible environment. Further, the declarative definition files also act as system documentation.

### 3.2.2 Kubernetes and its Components

Kubernetes follows exactly this declarative approach and was created by developers at Google. It is by far the most popular orchestration solution, now open source and managed by the *Cloud Native Computing Foundation*.<sup>26</sup> For oForest, we selected Kubernetes,

---

<sup>26</sup><https://kubernetes.io/>

because of its popularity but also because of its particularly good documentation<sup>27</sup> and community support. Further, it can scale easily from accommodating only a few to billions of containers without much additional complexity. oForest runs on [Microk8s](https://microk8s.io/), a lightweight and production-ready Kubernetes installation. MicroK8s comes with “batteries included and sensible defaults”<sup>28</sup> and makes the setup of a Kubernetes cluster very easy, as it offers build-in add-ons for things like SSL certificate creation, observability dashboards, Helm, Ingress controllers, and many more.<sup>29</sup>

The power of Kubernetes is its high level of abstraction. This allows for underlying technologies to be swapped out when requirements change or technology advances. For example, we can start off by using a local storage solution and as the system scales we could upgrade to distributed network storage, with little to no change to the cluster. The only downside of this abstraction is that new users first must understand the role of different Kubernetes components and how they interact.

In the oForest environment, we currently run a cluster with only a single node. In Kubernetes, a node is a physical or virtual server, while a cluster is a set of one or many nodes that are managed by Kubernetes. With a single node cluster, we can leverage all of Kubernetes’s automation capabilities but don’t need to worry about data consistency problems that often arise in distributed systems.

**Pods.** The “smallest deployable unit of computing” that can be managed with Kubernetes is called Pod.<sup>30</sup> A Pod is an abstraction layer that sits on top of one or multiple containers that work very closely together. In oForest, we will only ever have one container per Pod, so whenever we refer to a Pod, we will indirectly refer to the single container that is running inside.

More often than not, developers will not directly create or remove Pods themselves. Rather, they will create other components like Deployments that handle the Pod’s lifecycle. Pods are, just like containers, ephemeral, this means individual Pods can get deleted and recreated at any time, meaning that any data stored only inside the Pod can also be lost at any time. As long as we use a database, this is no reason for concern, as Kubernetes assures constant availability of Pods.

**Deployment.** A Deployment is a resource that manages and scales Pods, i.e. instances of applications. It makes sure that a desired number of replicas are running at all times,

---

<sup>27</sup><https://kubernetes.io/docs/home/>

<sup>28</sup><https://microk8s.io/>

<sup>29</sup><https://microk8s.io/docs/addons>

<sup>30</sup><https://kubernetes.io/docs/concepts/workloads/pods/>

handles rollouts of updates, and restarts Pods without downtime.<sup>31</sup>

Creating a Deployment includes specifying the desired state of an application, including the container image, the number of replicas to run, and further configuration, like environment variables. Kubernetes uses this information to create and manage Pods, ensuring that the actual state always matches the desired state. As Deployments are only responsible for making sure they maintain an expected state. They have no reliable way of communicating with the outside, or other Deployments, for this we need another component, the Service.

**Service.** Pods, by themselves, are not reliably reachable by other applications or resources, because of their ephemeral nature. As they can get deleted and recreated at any time, we must wrap them in a resource that manages this internal networking. A Kubernetes Service does exactly this.

A Service wraps around a Deployment and allows other Deployments to communicate with it from inside the cluster. For example, creating a Service for a database Deployment allows other Deployments, like an oTree application, to access said database.<sup>32</sup>

Services are only reachable from inside a cluster. For components like databases, this is fine as we only want to access them from the experiment that is also running in the cluster. The experiment itself on the other hand should be accessible from the outside. If we want to expose a Service to the world, i.e. the internet, we must create another Kubernetes resource, an Ingress.

**Ingress.** An Ingress resource handles incoming and outgoing traffic, like HTTP requests. They are connected to a Service and resolve incoming requests to a specific URL to said Service. Ingress resources enable load balancing, and SSL termination and allow applications to be accessible at a specified domain.<sup>33</sup>

**Storage with Persistent Volumes.** We mentioned the ephemeral nature of containers and how we can mount volumes in containers to achieve data persistence in section 3.1.2. If we want data persistence with Kubernetes Pods, we need to make use of the Kubernetes storage abstraction. This abstraction allows us to easily use different *storage classes*, such as *AWSElasticBlockStore*, *Network File Storage (NFS)* or local disc storage.<sup>34</sup>

For using a storage device we create a PersistentVolume resource, with the correct storage class and a PersistentVolumeClaim resource to be able to mount the volume to the desired

---

<sup>31</sup><https://kubernetes.io/docs/concepts/workloads/controllers/deployment/>

<sup>32</sup><https://kubernetes.io/docs/concepts/services-networking/service/>

<sup>33</sup><https://kubernetes.io/docs/concepts/services-networking/ingress/>

<sup>34</sup><https://kubernetes.io/docs/concepts/storage/storage-classes/>

Pod, e.g. a database. PersistentVolumeClaims allow users to request storage resources, and PersistentVolumes on the other hand allow administrators to provide storage with various properties. Data created or stored by a Pod that has claimed a PersistentVolume now gets persisted, depending on the storage class, on local disc, on AWS, or on network storage. In oForest, we for now only deal with local disc storage on a single node. If the need should arise the storage solution of course can be changed.

**Resource Quotas and Limits.** We explained how multiple applications can run side by side on a single cluster. As soon as these applications are deployed by different groups, it may happen that one application uses more resources than its fair share. In order to prevent this behavior, Kubernetes offers the capability to restrict the resource usage of Pods or namespaces. These restrictions come in the form of Resource Quotas or Limit Ranges.<sup>35</sup>

Resource Quotas can be used to impose resource restrictions on whole namespaces. If a user tries to create a resource that would exceed this limit, the resource is not created and instead, a 403 **Forbidden** status code will be thrown. Resource Quotas do not impose restrictions on a single Pod, but only ensure that the quotas are not exceeded overall. A LimitRange on the other hand can be used to assure that every new Pod in a namespace gets assigned specific resource limitations. Thus, ensuring that no Pod impairs the performance of other applications.

Limitations can be set to the CPU or memory. Memory limits are specified in megabytes (Mi), and CPU limits are defined in absolute CPU units. A CPU limit of 1 would mean that a Pod can use 1 CPU core at maximum, and a limit of 0.1 would mean a Pod can use a tenth of a CPU core.<sup>36</sup>

If a Pod uses more resources than it is entitled to, one of two things happens. In case the Pod uses more CPU, the Pod gets throttled. It will still run but at a capped rate. If, on the other hand, the Pod uses more memory, than it is allowed to, it is stopped by Kubernetes, as memory throttling is not possible.<sup>37</sup>

### 3.2.3 Helm

As discussed earlier, all the above-mentioned Kubernetes resources are managed declaratively. This means that for many deployments, we in theory also have to manage a large amount of `.yaml` files. When we deploy many similar applications, these can get repetitive

---

<sup>35</sup><https://kubernetes.io/docs/concepts/policy/resource-quotas/>

<sup>36</sup><https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#meaning-of-cpu>

<sup>37</sup><https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/#how-pods-with-resource-limits-are-run>

fast. In such a case, we ideally want to define a template and generate the corresponding manifest files, for this, we can use [Helm](#).

Helm is a package manager for Kubernetes that simplifies the installation and management of applications in Kubernetes. Helm allows users to easily define, install, and upgrade complex Kubernetes application deployments using pre-defined templates called charts.<sup>38</sup>

Essentially, Helm charts are a collection of Kubernetes resource files that are bundled together to form a reusable application deployment. These charts can be shared and reused across multiple Kubernetes clusters, making it easy to manage and deploy applications at scale. Further, charts can be customized, allowing the quick deployment of similar applications.

Additionally, Helm provides built-in support for versioning and rollbacks, allowing users to easily manage and track changes to their application deployments. This allows for a more robust and flexible application deployment process on Kubernetes.

In the context of oForest, Helm allows us to create a chart for a typical oTree experiment. We can specify the name and the corresponding image of our experiments and Helm deploys a suite of Kubernetes components for us.

### 3.3 Automation Tooling

The great power of the tools we employ in the oForest environment is their automation capabilities. Still, we need a place that triggers all of our automated workflows. In oForest, we make use of mainly two different tools. The first ones are GitLab pipelines, these allow us to run all automation steps that need to be executed if a researcher wants to deploy an experiment. The second one is Ansible, which we use to automate the setup of the oForest environment. Whereas we use GitLab pipelines every time an experiment gets updated or deployed, we only use Ansible on the initial setup of oForest.

#### 3.3.1 GitLab

[GitLab](#) is a platform that allows for management and collaboration on git projects. In this, it is similar to the more popular GitHub. Additionally, to this functionality, GitLab positions itself as “The DevSecOps Platform”.<sup>39</sup> As a platform focused on operations, it thus offers essential tools like a container registry, a package registry, and a *continuous delivery / continuous deployment* (CI/CD) pipeline. A container registry is, as discussed

---

<sup>38</sup>[https://helm.sh/docs/chart\\_template\\_guide/getting\\_started/](https://helm.sh/docs/chart_template_guide/getting_started/)

<sup>39</sup><https://about.gitlab.com/platform/?stage=plan>



earlier, an essential part of working with containers, package registries allow us to store and distribute Helm charts and CI/CD pipelines automate the deployment process.

Inside a CI/CD pipeline, we define stages that may depend on one another and which can build images, test code, deploy code to different testing or production environments and execute other management tasks.

### 3.3.2 Infrastructure Provisioning with Ansible

Ansible is a powerful open-source tool for task automation, such as deployment and configuration management. It allows users to define and manage infrastructure as code, making it easier to automate the provisioning, configuration, and deployment of various services and applications. It is commonly used in IT environments to manage and configure large numbers of servers and other devices, allowing administrators to easily control and manage complex systems.

Ansible uses a simple markup language, called *YAML*, to define automation steps. A file that contains a series of steps is called a *playbook*. Playbooks can not only be used for task automation, but they are also a useful form of documentation for how a system is configured. Further, Ansible has a large and active community. In the Ansible Galaxy community, members can share and collaborate on playbooks.<sup>40</sup>

Ansible is especially useful for oForest. Here we need to install and configure MicroK8s, Docker, and GitLab runners. Further, we must open firewall ports, set up Helm Charts, add command-line aliases and much more. We want to reliably recreate this complicated system without abstracting too much away: We want an administrator to understand all the parts of our system and how it's configured. Also, we want to enable them, to make traceable changes.

## 3.4 Linux Servers

Linux has become the de facto operating system for servers on the modern web. This ubiquity can be attributed to its ability to support a wide range of applications, from web servers and databases to cloud infrastructure and containers. Especially Kubernetes work natively on Linux. This makes it an obvious choice for the operating system we run oForest on. It may be possible to adapt oForest to Windows servers, but we strongly advise against it. Currently one of the strengths of oForest lies in its use of standard procedures, using another operating system would counteract this advantage.

---

<sup>40</sup><https://galaxy.ansible.com/>

## 4 Designing Experimentation Infrastructure

After we have introduced the technologies powering oForest, we will soon talk about how we put these tools together and how they interact. Prior to delving into the specifics of implementation, however, we will outline the overarching design objectives we aimed to achieve when constructing oForest. Our fundamental goal in building oForest is to enable researchers to deploy their experiments to infrastructure that is owned by their institution. To realize this objective, we must design our system with researchers and other relevant groups in mind. Hence, in developing oForest, we adopted an orientation towards the ISO standard for user-centered design (9241-210:2019). Our endeavors, in particular, emphasized the following design principles recommended by the mentioned standard:

- We use personas to base our design on an understanding of users and their environment. These personas allow us to derive requirements for oForest.
- We involve researchers, administrators, and laboratory management in our development process.
- We use feedback provided by the above groups to improve oForest.
- Our process is iterative.

By adhering to these principles, our objective is to create a user experience that approximates the ease of use offered by cloud-based platforms such as Heroku.

From our interactions with an experimental laboratory, we could identify different user groups. These groups interact with oForest in different ways and thus also have different requirements, namely, they are:

- Researchers
- Administrators
- Subjects

Each of these three user groups interacts with oForest from a different perspective, and thus each also has different functional requirements. We will explore these requirements by the means of user stories. We base these stories on personal interactions with the respective user group.

### **Persona 1: David (29), Researcher**

David is a researcher in behavioral sciences. He has limited experience with coding, but he is starting to get the hang of it as he started using oTree recently. He has access to a computer lab where he can invite participants to take part in his experiments. First, he handed his code to the local IT admin who then manually deployed the experiment. But as the admin was

overworked, deployment was a slow process. Thus, he was not able to make last-minute changes, also he had no way of accessing the error logs of his experiment.

Because of that, he decided to move to Heroku. The deployment process was well-documented and supported. He found it easy, as it only used intuitive graphical user interfaces or tooling that he already was familiar with. This allowed him to quickly get up and running with his experiments. Further, Heroku provided him with insights such as logs.

Then he got notified by the data protection officer of his university, that the data he is collecting is considered personal data. Because of GDPR concerns, he is no longer allowed to transmit this data to Heroku. The officer tells him, to comply with GDPR, it would be the easiest and safest if his research institution would keep the data in its own infrastructure.

We find the major requirements of the researcher to be:

1. **Simplicity.** He needs to focus on his research, he has no time to become an infrastructure expert any time soon. Thus, he needs a solution that he can get familiar with quickly.
2. **Autonomy and Speed.** He wants the deployment process to be fast and effortless, ideally, he wants to deploy experiments without the oversight of a third party.
3. **Data Privacy.** He wants all his work to comply with GDPR and wants to be assured that his data is safely stored.
4. **Transparency.** In case of a failure, he wants to understand what has gone wrong and access application logs.

With oForest, we can complete the user story.

This is where oForest comes into play. David talks to his IT admin and convinces him, to set up oForest. Now he just needs to push his oTree experiment to a GitLab repository, wait for his code to be built into a container, and then be deployed. Instead of having to deal with GDPR regulations or struggling with servers, he can focus on his research.

## Persona 2: Emilia (40), IT-Administrator

Emilia is overworked because she has to do lots of manual and repetitive work when a researcher wants to deploy a new experiment. Further, she is annoyed by the constant conflicts between different python versions and dependencies, that occur when she wants to set up a new experiment on the server. She knows many of the things she does for the experimentation lab, can be automated.

For adapting a new system, it would be important to her that it is secured from outside malicious intent. She further thinks the system needs to be protected from accidental misconfiguration by already accredited researchers.

To make her life easier, she would be willing to learn some new technologies. She also has heard a lot of good things about Kubernetes and the containerization of software in general. But if she uses a new tool, it should be well-documented and already widely used. Also she wants to get insights into how the system is running so that she can react on changing load requirements.

We find the main requirements of the IT admin to be:

1. **High level of Automation.** She doesn't want to do repetitive tasks again and again.
2. **Isolation and Consistency.** She wants multiple experiments to run independently from each other in a reliable reproducible fashion.
3. **Well Supported.** She wants to only use a tool that is well-documented and based on industry-proven technologies.
4. **Secure.** The system should not be changeable from unknown third parties. As researchers inside an organization work closely together, malicious behavior between them is not expected. Still, the system should be protected from unintended changes.
5. **Observability.** She wants to understand how her system is behaving and if she needs to increase available resources.

Again, oForest completes her user story:

When Emilia heard about oForest from David, she was delighted to find out, that the setup is almost completely automated and comprehensibly documented. Not only is the infrastructure setup automated, but more importantly, also the deployment of the research experiments. As oForest uses industry-standard tooling such as Kubernetes, Ansible, and Containers, Emilia can find plenty of additional resources to maintain and expand oForest.

### Persona 3: Hanna (20), Participant

Hanna is a student and was invited to take part in a behavioral economics study. Because she is a privacy-conscious person, she is a bit concerned about what will happen to her personal data. She would feel better if data on her behavior would not leave the research facility.

We find the requirements of the participant to be:

1. **High Privacy Standard.** The participant expects their data to be treated re-

sponsibly as it may reveal personal and intimate details.

With oForest we can complete the user story as follows:

Arriving at the research lab, Hanna is informed about the privacy statement. Here she learns that the data that is collected on her, will only be kept in the university infrastructure and not accessible to any outsider. With this peace of mind, she happily takes part in the experiment.

Through the use of personas, we could derive the user needs and requirements of our respective user groups. When we build oForest we always had these requirements in mind. We then iteratively designed oForest, by testing the system with real users and integrating their feedback. In the following sections, we will explain how the architectural components work together in order to match the above-mentioned requirements.

## 5 The oForest Components

Before we look at the components of oForest in detail, we will provide some higher-level intuition on how the parts work together. For this, we will follow the deployment of a new oTree experiment. Figure 5 shows a visual overview. We introduced the technologies we used to build oForest in section 3.

1. To deploy a new experiment to the oForest infrastructure, the respective researcher first must be invited to a GitLab-group, that has access to GitLab Runners running in a Kubernetes cluster. This is an action done by an administrator, e.g. the lab manager.
2. Once invited, the researcher can create a new project inside this group. For this, she can either create a clean repository and add necessary configuration files later or use a template. She will now have to clone her project to her local computer in order to add her experimentation code, for this she will use Git. After adding her code, she will commit and push her changes. With the push, an automatic GitLab pipeline is triggered.
3. This pipeline will build the experimentation code into an image and store it on the GitLab Container Registry. Further, an acceptance test will be run that ensures the experiment works properly. Once built and tested, the image can be deployed. For this, a set of new Kubernetes resources must be created. As these resources are equivalent for every experiment, oForest provides a Helm package for oTree applications.
4. Helm now creates and applies the new manifest files to the cluster. As soon as the desired cluster state changes, Kubernetes starts matching it. It now creates an experiment Deployment, Pod and Service, a database Deployment, Pod and Service, an Ingress to

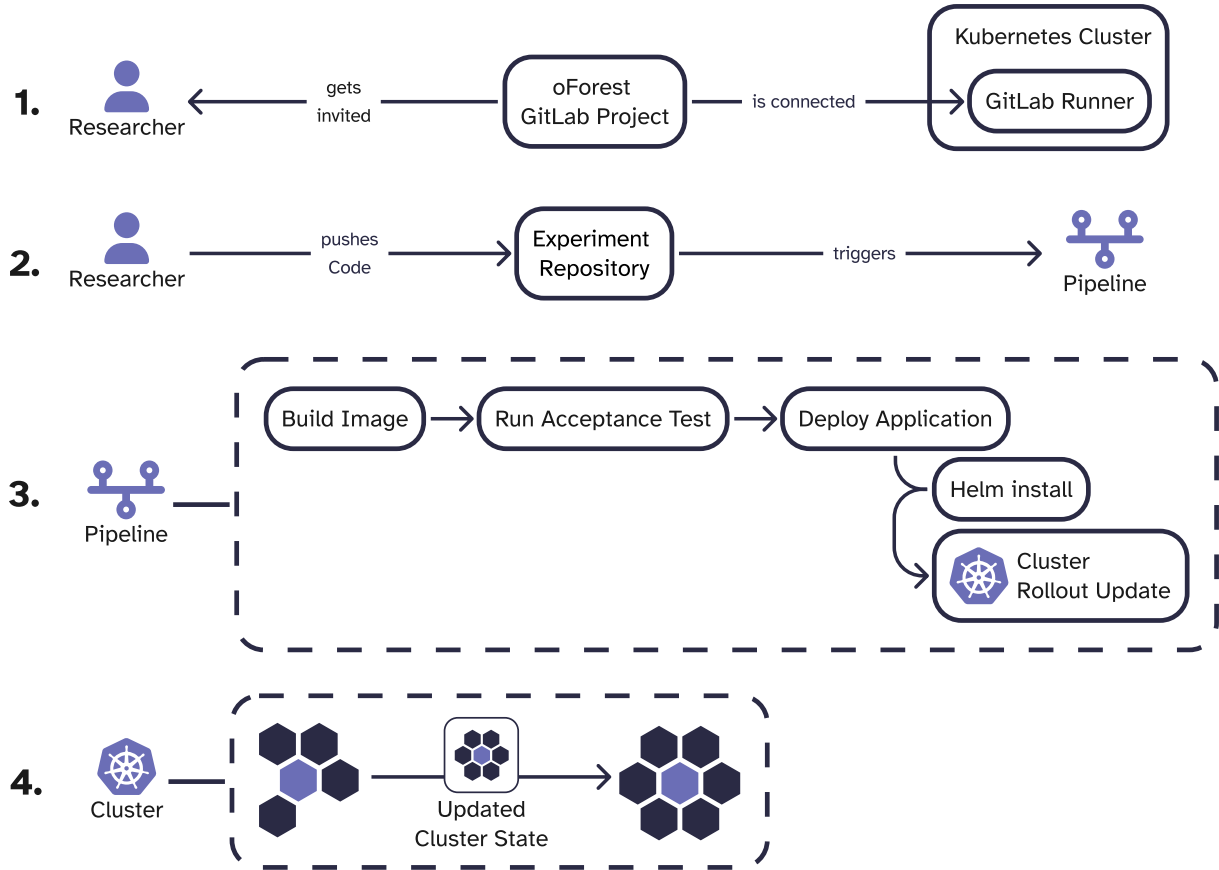


Figure 5: Intuition: How interactions with oForest work

make the experiment accessible from the outside, and a new SSL secret for encrypted HTTPS traffic.

The experiment is now reachable at a specified domain and ready to be used. Additionally, the user now can also trigger some maintenance jobs, such as wiping their experiment database or deleting their deployment. As we now have provided a broad overview of how the different components work together, we will explain them in greater detail.

## 5.1 Containerizing oTree

We want oForest to not only host one but many oTree experiments. All these experiments can in theory be based on different oTree versions, rely on varying dependencies, and use different software packages. It is thus unfeasible for all experiments to share the same environment. With a high number of different experiments, it also becomes too much work to set up a different environment for different experiments manually. Containerizing the oTree experiments thus is a logical step.

We introduced the concept of containers in chapter 3.1. Containers allow us to create standard units of software that contain all the required dependencies. By building images for all our experiments, we can then deploy containers from these to our infrastructure. As containers won't interfere with other containers, experiments run independently of one

another.

### 5.1.1 oTree Dockerfile

A `Dockerfile` specifies the building instructions of a container image. As we want to build an experiment-specific image, we also require an experiment-specific `Dockerfile`. In it, we need to do the following things:

1. **Choose a base image.** oTree is a python application, so the `python:latest` image is an obvious choice. It allows us to execute python code inside a container.
2. **Copy the experiment code from the local source folder to the container.** Copying the source code to the image is an essential step, it makes sure the image actually contains the experiment. We also copy a script to the image that, when called, runs the `otree resetdb`<sup>41</sup> command.
3. **Install all dependencies.** Here we install not only the requirements specified in the Python `requirements.txt` but also `libpq-dev` and `gcc`, both packages that are needed for interacting with a Postgres database.
4. **Expose port 8000.** When the oTree server is started, it expects requests on port 8000. As containers are encapsulated, we must specifically expose the given port. Otherwise, we would have no way of receiving any outside traffic.
5. **Define the startup command.** Last, we define the command that should be run if the container is started. In the case of an oTree container, we want to start the production server.<sup>42</sup>

### 5.1.2 Weighing Storage against Convenience

In the previous section, we argued that because oTree is a Python application, we should also just use the official Python image in our `Dockerfile`. Any additional packages that are required by the application, would be then installed during a build. For example, if an experimenter uses `pandas` or `numpy` in their code, they would not only need to install it to their local machine but also needed to add these packages to the `requirements.txt` of their project. In running tests, we found that many researchers had used [Anaconda](#) as their Python environment. As Anaconda denotes, it comes with “thousands of packages” preinstalled. As researchers didn’t specifically install additional packages, they often also had forgotten to add used packages to their `requirements.txt`. In such a case, the application inside the container wouldn’t run properly, although it did in their local environment, leading to errors during experimentation.

---

<sup>41</sup><https://otree.readthedocs.io/en/latest/server/ubuntu.html#reset-the-database-on-the-server>

<sup>42</sup><https://otree.readthedocs.io/en/latest/server/ubuntu.html?highlight=prodserver#testing-the-production-server>

In solving this problem, we can take two approaches. We either let the researchers build application images that won't work and inform them of the specific error messages that occur when trying to start the application, or we instead of Python use the Anaconda base image. This has the advantage that most of the popular packages researchers will use are already included in the image, and it won't matter if someone forgot to update their `requirements.txt`. The only drawback we face with the latter approach is a higher storage requirement. Whereas the Python base image is only around 300 MB in size<sup>43</sup>, the Anaconda base image is more than three times larger (1.19 GB)<sup>44</sup>.

For oForest, we valued the reduced risk of an error occurring higher than a more resource-efficient image. Ultimately, though, this is an individual preference. Changing back to a standard Python image can be done by changing the first line of the respective `Dockerfile`.

The usage of containers in oForest allows us to run experiments in a reliably reproducible environment. The container image we build of an experiment will always run in the exact predictable way. Further, the isolation between running experiments means that no versions or package requirements of different experiments will clash. Thus, containers help us in achieving the *Isolation and Consistency* requirements.

## 5.2 Cluster Configurations

We introduced some basic Kubernetes components earlier, we will now show how we actually used these components in the oForest infrastructure. We will start out by explaining the components that make up an oTree experiment and then will touch upon the remaining resources that are required to keep our system running smoothly. An overview of components inside the cluster can be found in figure 6. We will start by explaining the Kubernetes components that make up an experiment

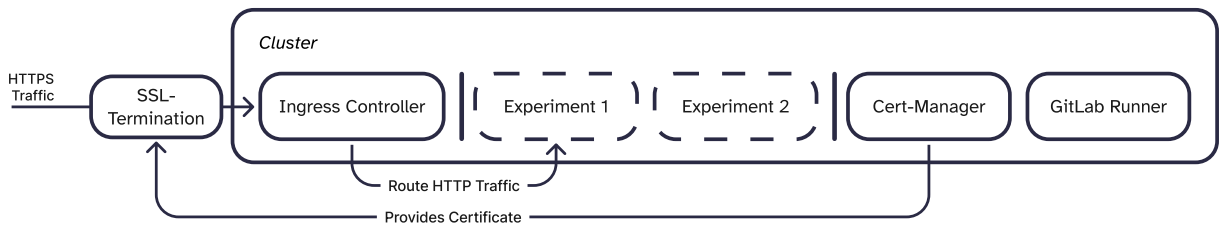


Figure 6: Interaction of components inside the cluster.

<sup>43</sup>[https://hub.docker.com/\\_/python/tags](https://hub.docker.com/_/python/tags)

<sup>44</sup><https://hub.docker.com/r/continuumio/anaconda3/tags>



### 5.2.1 Experiment: Deployments

The first Kubernetes component we discuss is an experiment. There can be many experiments in one cluster and for each experiment we need multiple Kubernetes resources. The first of these resources is a Deployment. This Deployment will manage the lifecycle of the Pods, inside which oTree is running. In the definition of such a Deployment, we pass different kinds of information:

- **The Image.** A container needs an image that it's based on. We build this image from the source code a researcher provides using the GitLab pipeline. Inside the Deployment manifest, we define the registry location of this image. In our case, this is the GitLab registry of the experiment repository.
- **Metadata.** Each oTree Deployment has a name that consists of `-otree` prefixed with the name of the repository from which the experiment is deployed.
- **Number of Replicas.** This defines how many instances there are running of each experiment at any given time. To achieve seamless availability, we require to always have two experiments present.
- **Environment Variables.** The user might want to manually pass environment variables to the application. These could be oTree specific variables, API keys, or custom variables. Further, we provide standard required variables that should not be altered by the user, such as the `DATABASE_URL`.

Besides the Deployment component for the oTree application, we also need a second Deployment for the database. As oTree works with Postgres out of the box, we create a second Deployment based on the `postgresdb` image to be deployed. In the configuration, we provide a master password as an environment variable. For data persistency, we define both a storage device, i.e. the server disc, and a mount path, which defines in what directory data is stored.

Although we now have both a Deployment for the experiment and for the database, currently they are not able to communicate. To expose these Deployments to other cluster resources, we need to define two Services.

### 5.2.2 Experiment: Services

The first Service, we need, exposes the database. For this, we specify which Deployment we want to make accessible to the cluster. Now, other Deployments can access the default database under `postgres://postgres:[PASSWORD]@[EXPERIMENT NAME]-postgres-service:5432/django_db`.<sup>45</sup> With the database accessible, data can now be safely read and written

---

<sup>45</sup><https://otree.readthedocs.io/en/latest/server/server-windows.html?highlight=database#database-postgres>

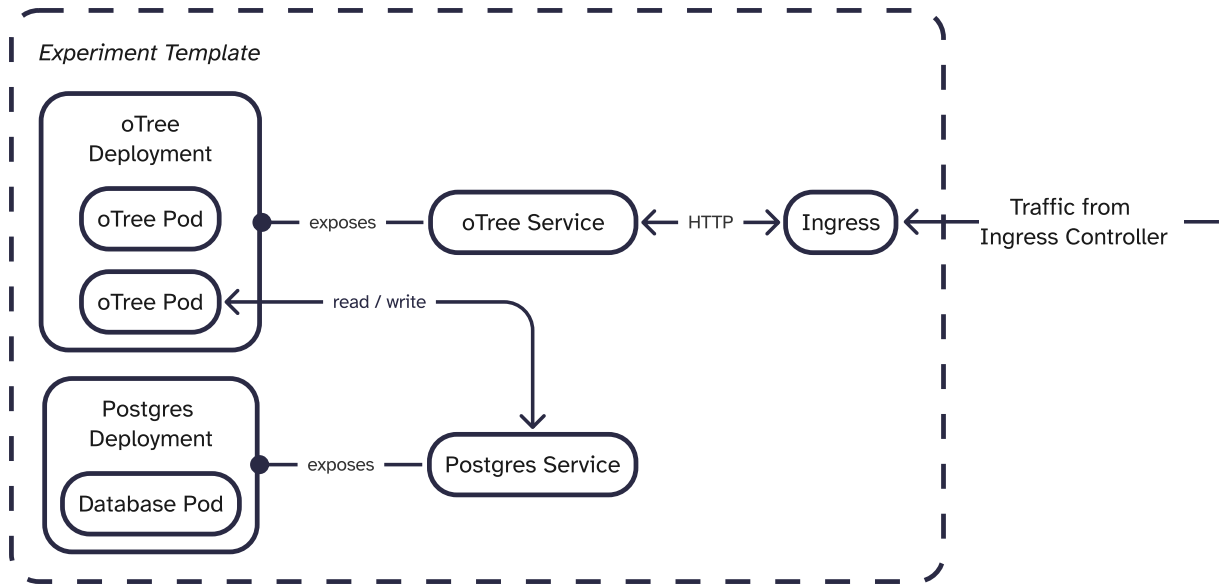


Figure 7: oTree experiment architecture inside the cluster

by our experiment.

The second Service we need is for the oTree Deployment. Without it, we have no way of actually reaching the experiment from outside the cluster, i.e. the internet. The Service for the oTree Deployment is similar to the database Service and differs mainly in the name.

### 5.2.3 Experiment: Ingress Resources

To be able to route traffic from the outside to the oTree Service and then to the corresponding Deployment, an Ingress is required. The Ingress is connected to a load balancer and routes incoming HTTP traffic to the correct application inside the cluster based on a specified hostname. This means we define a hostname, e.g. `https://[EXPERIMENT NAME].kit.edu`, and a Service name, e.g. `[EXPERIMENT NAME]-service` inside our Ingress resource, and whenever there is incoming HTTP traffic directed at that domain, it automatically gets redirected to the specified Service.<sup>46</sup> In this way, we make an experiment available to the wider internet.

Besides the host and Service definitions, we also define a TLS secret name and specify which certificate issuer the Ingress route should use. This essentially ensures that all traffic is SSL encrypted and can use HTTPS.

Lastly, we make some further adjustments to the Ingress route to allow for HTTP traffic to be upgraded to a web socket connection. As web sockets are long-standing connections,

<sup>46</sup><https://kubernetes.github.io/ingress-nginx/user-guide/basic-usage/>

we mainly raise the timeout duration and allow for `HTTP Connection upgrade-headers`.<sup>47</sup>

With the Ingress in place, experiments are now reachable for participants of oTree experiments. The resources we just described, which are the two Deployments, the two Services, and the Ingress, are required for every experiment in the oForest infrastructure. We can manage multiple resources using Helm.

#### 5.2.4 Bundling Resource Deployments using Helm Charts

In the previous sections, we explained the different Kubernetes components that need to be created every time an experiment is deployed. Using the Helm package manager, we can install all the above-described components using only a single command.

For this, we define a Helm chart that has the following structure:

```
oTree Chart
|- templates
    |- app.yaml
    |- db.yaml
    |- secrets.yaml
    |- required.yaml
|- Chart.yaml
|- values.yaml
```

Inside the template folder, we find the Deployment, Service, and Ingress for the oTree app and the Deployment and Service for the Postgres database. Further, we provide secrets such as the admin password through the `secrets.yaml`. Finally, we define values that the user needs to provide when installing the chart inside the `required.yaml`. Among these required values are:

- **baseDomain**. Experiments are deployed to `https://[Experiment Name].baseDomain`. the domain that is set here should be owned by the institution that is running the oForest instance. The DNS entry of the **baseDomain** must point to the IP address of the cluster.
- **name**. This is both the name of the Helm deployment and the prefix of the components that get created.
- **dbPw**. This is the master password that gets used both as a database password and as an administrator password for the oTree admin interface.

Inside the `values.yaml` we find the remaining values with their respective defaults. For example, many of the oTree environment variables have their defaults set here. Lastly,

---

<sup>47</sup><https://kubernetes.github.io/ingress-nginx/user-guide/miscellaneous/#websockets>

we find some metadata such as the chart version inside the `Chart.yaml`.

This chart now gets bundled and lives inside the oForest GitLab group. From here it can be pulled and installed like any other Helm chart. In the case of oForest, we install the chart from inside the GitLab CI/CD pipeline. Inside the pipeline, we also set or modify the values mentioned above.

Helm does not only provide us with a high level of automation but also with customizability. As these customizations can be done by researchers without the help of an administrator, Helm helps us to fulfill the requirement of *Autonomy*.

### 5.2.5 Certificate Issuer

We mentioned that MicroK8s, the Kubernetes installation oForest is based on, is powerful because it provides many add-ons out of the box. One of these add-ons is a certificate manager. A certificate manager can issue X.509 certificates, the default used in SSL encryption, automatically. The `cert-manager`<sup>48</sup> specifically uses the servers of the *Let's Encrypt*<sup>49</sup> initiative.

In order to work with the `cert-manager` we need to create a *ClusterIssuer* resource in our Kubernetes cluster. In the definition of this resource, we only need to specify the server, that should be used to issue new certificates. Now, every time a new Ingress resource gets created, that uses the *ClusterIssuer*, a new certificate gets automatically issued. After the issuing is processed, the connection to the specific Ingress is now SSL encrypted.

### 5.2.6 Resource Limits and Quotas

To make sure no experiment impairs other experiments by taking too many resources, we use Limit Ranges and Resource Quotas. We base these restrictions on the size of the server we are running and the amount of resources consumes in their idle state, as described in Table 1. As a result, we restrict experiments to a total ram usage of 500 MiB and to 0.1 CPU usage.

We further limit the resource usage of the experimentation namespace to roughly 80% of both CPU and RAM. In this way, we give administrators room to momentarily increase the allowed utilization, while they scale up the system. As we mentioned before, it is not a good idea to run the system close to its capacity.

We apply both the Limit Ranges and Resource Quotas using Ansible and a `quota.yaml` configuration file, this makes it easy for administrators to adapt these values to their

---

<sup>48</sup><https://cert-manager.io/docs/usage/certificate/>

<sup>49</sup><https://letsencrypt.org/>

specific circumstances.

### 5.2.7 Storage

We will now explicitly show how the Postgres Pod persists its data. As explained, we require a single PersistentVolume of the type `local-storage`, which persists data to the local disc. Specifically, data is stored on the server in the folder `/root/cluster_storage/`.

Second, the database Deployment uses a PersistentVolumeClaim, to request access to the volume storage. As all experiment database Pods share the same PersistentVolume, i.e. the same disc, we mount the Postgres data to a sub-path equal to the experiment name. In other words, data that belongs to a Deployment named `experiment_1` gets saved to `/root/cluster_storage/experiment_1/`.

### 5.2.8 Observability

Another useful feature we can leverage when using MicroK8s is the observability add-on. This add-on enables metrics gathering using [Prometheus](#) and exposes a dashboard with Grafana.



Figure 8: The Grafana dashboard

Prometheus scrapes numeric measurements such as CPU utilization, I/O load, network speed, and running processes and stores it as time series data.<sup>50</sup>

This data then can be queried and visualized using [Grafana](#). Dashboards can be manually built, but most of the relevant information can be already found in one of the preset dashboards. Among these metrics are CPU utilization, RAM utilization, disc utilization,

<sup>50</sup><https://prometheus.io/docs/introduction/overview/>

disc I/O, network traffic, resource usage of Kubernetes namespaces, resource usage of individual Kubernetes Pods and many more.

To make the dashboard accessible, we created an Ingress resource for Grafana. With it, the dashboard interface is reachable under `https://observability.[baseDomain]`. Having observability for administrators is one of the requirements we ask of oForest, we achieve it by using Prometheus and Grafana.

### 5.3 GitLab as the oForest User Interface

Researchers who want to deploy their experiment store their code in a git repository inside a GitLab group. GitLab not only makes collaboration on code much easier, but it also powers all user interactions with the oForest cluster.







Account	Source	Max role	Expiration	Activity
 Alice	oForest by Admin	Maintainer	Expiration date 	User created: Feb 02, 2023 Access granted: Feb 02, 2023 Last activity: Mar 15, 2023
 Admin <span>It's you</span>	oForest	Owner	Expiration date 	User created: Nov 23, 2021 Access granted: Dec 17, 2021 Last activity: Mar 20, 2023
 Bob	oForest by Alice	Developer	Expiration date 	User created: Dec 08, 2022 Access granted: Dec 08, 2022 Last activity: Mar 15, 2023

Figure 9: GitLab’s user management interface.

As a user interface, it also takes over the task of user management. As said, all experiments need to live in a specifically configured GitLab group. This group has the capability of deploying applications to the oForest cluster. If an administrator wants to enable a researcher to deploy an experiment, he can simply invite them with permission to create new projects inside the oForest group. This means we not only leverage the automation capabilities of GitLab but also use it as a user admission and management system. Without GitLab, oForest itself would need to store user accounts, enforce admission control, handle user roles, and send password-reset-mails. Using GitLab thus strongly reduces the complexity of the overall system, which in turn makes oForest more maintainable.

For oForest to work, we require the GitLab group to have a few special repositories. A visualization of these files can be found in Figure 10. The first one hosts the Helm chart, which allows us to quickly deploy all required components for an oTree experiment to the cluster. The second hosts a sample project that acts as a blueprint for all oTree experiments. It contains a `Dockerfile` that will work for a standard oTree application. The last project contains the CI / CD pipeline configuration file. This file controls the automatic execution of the deployment process, it is the centerpiece of oForest.

As oForest leverages the mature user admission system of GitLab, we make sure the system is secure. Only invited parties with the correctly set rights, can deploy applications or

change configurations. Having the CI / CD pipeline isolated in a different repository also protects it from accidental misconfiguration. The usage of GitLab thus helps us to fulfil the requirement of being *Secure*.

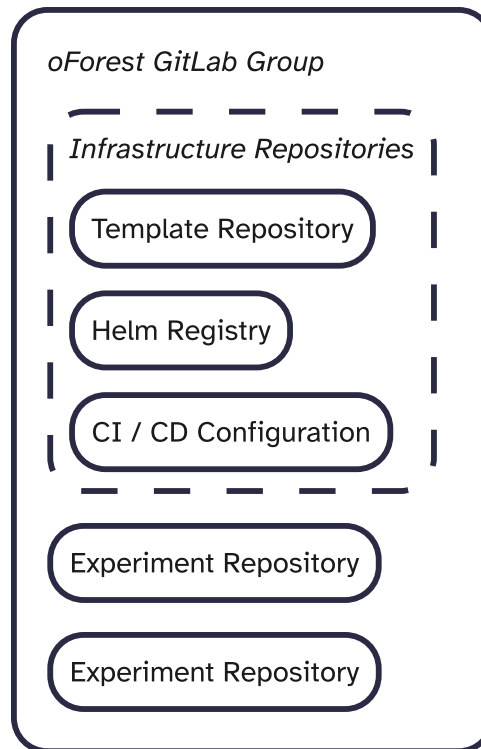


Figure 10: oForest GitLab Project structure

## 5.4 The CI / CD Pipeline

Thus far, we have shown how pieces of oForest work together, but only in a manual way. It is apparent that the manual execution of commands is not feasible, we thus require a high level of automation. To achieve this automation, we define multiple automation steps in a `gitlab-ci.yaml` file. This file is referenced in each GitLab project and triggers the execution of different tasks on every commit. Because the pipeline is defined for each project separately, they are highly extensible. Users could thus easily add a stage for e.g., unit testing, if they needed to. The commands for the respective jobs are then executed by so-called GitLab Runners. Each job that a runner tries to execute can either succeed, which in many cases triggers the next job, or can fail. In the latter case, the user receives visual feedback that something went wrong.

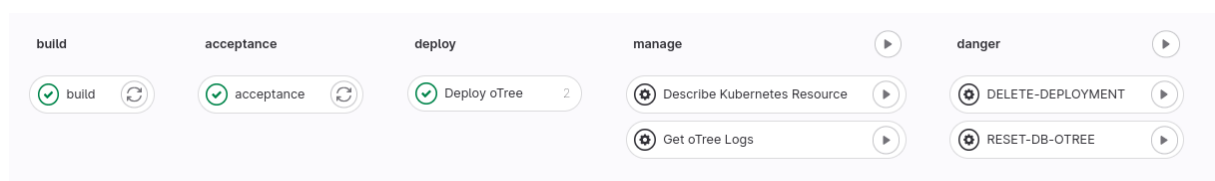


Figure 11: An overview of the oForest pipeline interface

Besides the stages of the pipeline that allow users to deploy their experiments, we also supply dedicated management jobs. These must be triggered manually and allow users to get logging information, reset their databases, or delete their experiments. We will look at these stages in the following section.

#### 5.4.1 Stage 1: Building Containers

One of the most important steps in the pipeline is building the experiment image. As mentioned, in this step all dependencies, runtimes, system tools and libraries, settings, and the actual oTree code gets built into one single image. In section 3.1.1 we discussed how the `Dockerfile` of a containerized oTree experiment might look, and also how to build such a `Dockerfile` into a container image locally by using the `build`-command.

As this build command relies on Docker, using it inside the pipeline is not straightforward. The executing instance of the pipeline step is already running in a container inside the oForest cluster. This means trying to run the `build`-command, would execute Docker inside a container. Although it is possible to run Docker inside a Container using the *Docker in Docker Service*<sup>51</sup>, this requires the original container to run in `privileged` mode.

As Walsh (2020) explains, the `privileged` flag tells the container engine to launch the container process without any further “security lockdown”. He advises not to use the `privileged` mode if not necessary, meaning that we need a fallback solution for building containers in our pipeline.

Luckily, we can use Kaniko to power our build process. Kaniko works by taking the file system of the base image, executing various commands on top, and snapshotting the user space. These snapshots then get appended to the base image<sup>52</sup>, creating a new and final image of the oTree experiment.

After building the image, we then push it to the container registry of the corresponding project. From here, Kubernetes can later pull the image and deploy it to the infrastructure.

#### 5.4.2 Stage 2: Running Acceptance Tests

After the container image is built, it is crucial to check if it is even working. Building images will work, even if applications have runtime errors. These errors might include missing dependencies, misconfiguration of the application, or faulty code.

---

<sup>51</sup><https://www.docker.com/blog/docker-can-now-run-within-docker/>

<sup>52</sup><https://github.com/GoogleContainerTools/kaniko#how-does-kaniko-work>



The best way of finding out if an application will run is by starting it. Thus, we use the acceptance stage, which starts the application and uses `curl`<sup>53</sup> to run a simple HTTP request against the `version` endpoint of an oTree application<sup>54</sup>. This query returns the oTree version that the experiment is running on, but more importantly, it will only work if the application started properly.

If the request succeeds, the pipeline job will also succeed, and the experiment can be deployed. In the case that the request fails, the pipeline job also fails and the error message responsible for the failure is returned.

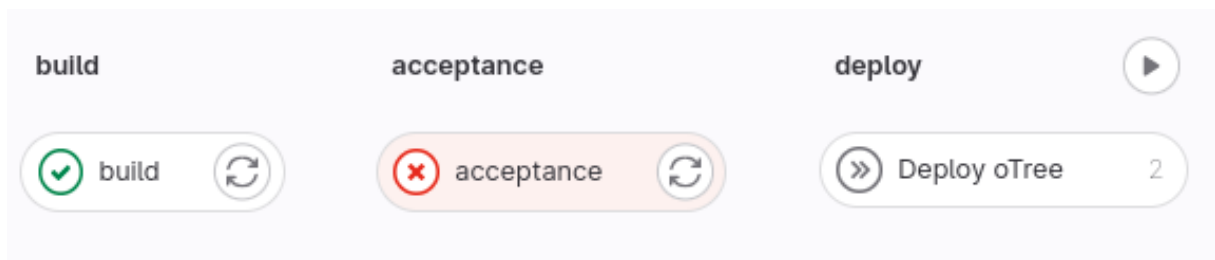


Figure 12: A failed acceptance test stage.

The acceptance step is only a very basic safeguard, it checks if applications will start without errors. This stage only makes sure that experiments that don't even start, also won't be deployed to the cluster. Despite this, experiments should be thoroughly tested both locally and in the live environment.

### 5.4.3 Stage 3: Deploying to Kubernetes

After the build and the acceptance stage, experiments can be deployed to the Kubernetes cluster. For this, we use Helm, the Kubernetes package management we introduced in section 3.2.3.

To be able to execute Helm commands, we need to make sure to use the `alpine/helm`<sup>55</sup> image for the job execution. In our execution, we first need to add the registry, from where we can pull the correct Helm chart. As we are using the build in GitLab package registry, we can log in to the registry using the pipeline credentials. After this, we can deploy a new experiment based on the oTree Helm chart to the cluster.

While executing the deploy command, we specify important constants such as the experiment name, the password for both the database and the oTree Admin interface, the base domain under which we want to reach the experiment later and the container image that should be used for the deployment. Most of these variables are set automatically.

<sup>53</sup><https://curl.se/>

<sup>54</sup>[https://otree.readthedocs.io/en/latest/misc/rest\\_api.html#otree-version-endpoint](https://otree.readthedocs.io/en/latest/misc/rest_api.html#otree-version-endpoint)

<sup>55</sup><https://hub.docker.com/r/alpine/helm>

As we might not only want to deploy completely new experiments but also update already existing ones, we need to restart the Deployment after updating the Helm components. We do this in a separate job, as we require access to the `kubectl` command line tool. We access this command by using the `bitnami/kubectl` image. From there, we issue a restart of the respective Pods.

This concludes the main stages that are required for deploying an oTree application. The remaining jobs help in managing already deployed applications.

#### 5.4.4 Management Jobs

Being observable is one of oForest’s requirements. In practice, this means users can get information from their application Pods or get access to their application logging output.

**Access Logs of Application Pods.** Running the first management job, allows the user to print out the application by logging into the GitLab runner output. This is done by choosing `bitnami/kubectl` as a runner image and interface with the Kubernetes API over the `kubectl` command line tool. We can access the logs by specifying the correct Deployment and then running:

```
kubectl logs deployments/[Experiment Name]
```

**Describe Deployment Resource.** Running the second management job returns general information about Deployments. This information includes

- **The resource name and namespace.**
- **The Replica state.** How many Pods are desired and how many are running?
- **Creation Time.** A timestamp that shows when a resource was created.
- **Pod template information.** Here, environment variables and ports are displayed.

All this information can be used to troubleshoot applications without requiring the help of an administrator. The management stage thus not only creates observability but also autonomy.

#### 5.4.5 Danger Zone

The *danger zone* is another feature that enables autonomy for researchers. It allows them to delete experiments or reset their experiment databases.

**Reset the oTree Database.** When changing oTree experiments after they already have been deployed, i.e. by adding pages of database fields, it might be necessary to reset the database. For this, oTree provides a simple command: `otree reset db`.

For resetting the database from the pipeline, we use the `kubectl` command line tool and run the `resetdb.sh`-script:

```
kubectl exec deploy/[Experiment Name] -- bash .resetdb.sh
```

**Delete Experiment from Cluster.** As the name suggests, the *delete deployment* job deletes the experiment and all its connected resources. These include the Ingress and the database. As we added the experiment deployment using Helm, we can also delete it using Helm. To do this, we again require the `alpine/helm` image to be able to run:

```
helm uninstall [Experiment Name]
```

We have explained the crucial role the GitLab pipeline plays for oForest. It helps us to match many of the user requirements: First, it greatly automates the deployment process matching *Automation*, and it then allows researchers to autonomously deploy and update their experiments without the interference of an administrator, matching *Autonomy and Speed*. Further, many of the steps run behind the scenes and don't require user interaction making it easy for researchers to get going, matching the *Simplicity* requirement. The management jobs not only aid in *Autonomy* but also help to get a *Transparent* insight into running applications.

#### 5.4.6 GitLab Runners

The GitLab runner is the executing instance of each pipeline stage and is connected to a GitLab group or project. With oForest, we suggest creating a group and enabling all group projects to use connected runners. To be able to use such a runner, first, we must choose a fitting *executor*. GitLab offers a wide selection of executors for different scenarios. Executors for instance can be run in the shell, on a virtual machine, in a Docker Container, or in a Kubernetes cluster.<sup>56</sup>

<input type="checkbox"/>	Status	Runner	Owner
<input type="checkbox"/>	<span>Online</span> <span>Idle</span>	<b>#21104601 (ubNuvFPCJ)</b> Group Version 15.8.2 · gitlab-runner-6c44dfc4c-rdjk4 Last contact: 57 minutes ago  129.13.111.98  159  Created 1 month ago	<u>oForest</u>

Figure 13: GitLab runner status as seen in the oForest group settings.

These executors mainly differ in where they run and thus also what they can access. The Kubernetes executor, for example, has access to the Kubernetes API, which allows it to

<sup>56</sup><https://docs.gitlab.com/runner/executors/>

not only spawn new Pods for each CI Job but more importantly to deploy components to the cluster directly itself.<sup>57</sup>

As oForest runs on a Kubernetes cluster, this is a crucial capability. Although the build and acceptance stages do not necessarily require access to the cluster, the deployment and management stages do. We thus selected the Kubernetes executor. It can be installed to any cluster using Helm, the Kubernetes package manager. GitLab itself provides a registry with the respective Helm charts.<sup>58</sup>

When setting up oForest using the Ansible setup, the runner is configured in `gitLabRunner/gitLab-runner-config.yaml` and `gitLabRunner/gitLab-runner-secret.yaml` files.

## 5.5 Server Setup

In the previous sections, we described the components of the oForest environment. As there are more than a few systems that interact with each other, setting up the oForest manually might seem laborious. That's why we use Ansible to automatically create the oForest environment.

We do this in a two-stage process, whereby the first stage requires no administrator configuration. The second requires the administrator to do some minimal configuration. Setting up a fresh cluster can be done in under half an hour and takes even less time, for an experienced administrator. All configuration files can be found in a GitHub repository that has the following structure:

```
oForest
|- Ansible
    |- cert-manager
    |- gitLab-runner
    |- observability
    |- storage
    |- general-setup.yaml
    |- gitLab.yaml
    |- inventory.yaml
    |- README.md
|- GitLab
    |- pipeline
    |- template-experiment
|- HelmChart
```

---

<sup>57</sup><https://docs.gitlab.com/runner/executors/#kubernetes-executor>

<sup>58</sup><https://docs.gitlab.com/runner/install/kubernetes.html>

```
| - oTreeChart
  | - templates
  | - Chart.yaml
  | - values.yaml
```

The Ansible folder contains all the Playbooks and configuration files that are needed to create the oForest environment on the server. The GitLab folder contains two repositories that need to be uploaded to their own project as soon as an administrator has created a GitLab group. Last, the HelmChart folder contains all the Manifest templates, that are needed in deploying a new oTree experiment.

### 5.5.1 General Setup

Running the setup requires a server that has at least Ubuntu 22.04 installed and to which the administrator has `ssh` access. Further, the administrator needs Ansible installed on his local machine. With these requirements matched, running the `general-setup.yaml` playbook will perform the most important setup tasks. Among these are:

- Installing MicroK8s and enabling the used add-ons
- Opening the Firewall for web traffic
- Add a `ClusterIssuer` for SSL certificate issuing
- Expose the Grafana observability dashboard

### 5.5.2 Installing GitLab Runners

The second stage of the setup adds GitLab runners to the cluster. To connect a runner from inside Kubernetes to GitLab, we need to provide it with a registration token. For this, the administrator needs to copy the respective token from the GitLab web interface and add it to the token section of the `gitLab-runner-secret.yaml`. In the same file, the administrator also needs to add an access token that is allowed to pull images from the oForest group registries. This access token can also be created from the GitLab web interface.

### 5.5.3 Choosing the Correct Server Size

Provisioning a server with the correct size is not trivial, as the required size depends on a multitude of variables. For example, it can vary with the number of simultaneous experiments, the number of participants per experiment, and the performance or complexity of the experiment code.

If any resource is fully utilized, this can have bad consequences for any experiment or application. It can lead to throttling or even system failure. We thus suggest running any given server always below capacity. It is a good idea to choose the maximum utilization

Name	RAM	CPU	Disc
Kubernetes (kube-system)	160 MiB	0.04	- (ephemeral)
Experiment deployment while idle	120 MiB	0.003	1.2 GiB
Observability	1 GiB	0.1	6 GiB

Table 1: System Load of selected Components.

depending on how fast administrators are can scale up the system. For knowing when to scale a system, monitoring is extremely important. The observability dashboard we talked about earlier is built exactly for this.

We understand that an *it depends* answer is little satisfying. We will thus provide some clues that might help to correctly size a server for your oTree experiments. For the system running in a testing stage, we currently have 2 CPU cores running at 2.10 GHz, 7 GiB RAM, and 150 GiB disc storage. We found our system disc to be working in good condition, although we have not gotten the chance to run large-scale experiments. Table 1 shows the system load of the respective components.

We only gave an estimate of the load for experiments while no active participants were taking the experiment. With only a few tens of participants, this value will vary only a little. As soon as hundreds of participants take part, the load is bound to increase. At the moment of this writing, it is hard to say how much. This would require the implementation of load-testing bots. Although oTree has capabilities that are named “bot”<sup>59</sup>, they can be only used to automatically play through pages inside a session. As these sessions and pages still must be created manually, they are not suitable for load testing, beyond around 20 participants. Building a proper load-testing tool goes beyond the scope of this thesis. Still, manual load testing by gradually increasing load and closely monitoring the system is possible with the Grafana dashboard.

## 5.6 Security and Privacy

As mentioned, oForest is using GitLab as a user admission system. Third parties that are not invited as collaborating roles into the GitLab group, to which the GitLab runners are connected, have no way of interacting with oForest in any malicious way. As soon though as a researcher has access to the group, he could in theory run `kubect1` commands to remove or manipulate the experiments of other researchers. This is explicitly not true for the other resources like the certificate issuer or the observability dashboard, as these are running in different Namespaces<sup>60</sup>, also researchers are not able to change the server configurations. It is also highly unlikely that a researcher accidentally removes an

<sup>59</sup><https://otree.readthedocs.io/en/latest/bots.html>

<sup>60</sup><https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/>

experiment as every command normally runs in an automated pipeline.

We think this is an acceptable risk as it reduces the complexity of the overall system. Researchers could in theory deploy experiments to their own Namespace and thus would be isolated from each other, but this would mean manually creating and adding new GitLab runners to each new experiment GitLab project. This is because runners can only create new resources in the Namespace that they live in. With very large institutions using oForest such isolation can become a requirement. In such a case, we advise either running multiple clusters or creating different namespaces for individual subunits such as chairs or research groups, instead of every experiment.

With respect to GDPR concerns, we solve them by operating oForest on research institution owned hardware. As elaborated earlier, this makes privacy measurements a standard procedure that is done in any case. By self-hosting experiments, the goal of a *high data privacy standard* is met.

## 5.7 Longevity and Maintainability

With any software, of course, maintainability and longevity are important things to consider. oForest is no different in this regard, but because it is built on stable industry standards, only a few breaking changes that need active maintenance are to be expected in the next years. This stable nature, abundant documentation and active community of the tools we use, help in achieving the *Well-Supported* requirement. This means it is possible to add features to oForest later or fix problems when they occur.

Although well supported, if requirements change or if capabilities should be added to oForest, work still is required. In this case, the question changes from *Is oForest maintainable?* to *Who maintains oForest?* In our eyes, the latter is the crucial question. With oForest, we have ultimately built a tool that centralizes the complexity of deploying oTree experiments, where we then use a plethora of tools to manage it. Handling centralized complexity requires special know-how. For someone to take such a central role and build this know-how requires strong incentives. Institutions must have a plan on how to get the capabilities to work with the tech behind oForest. Most importantly, they must have a plan on how to retain this knowledge.

As soon though as know-how is present, scaling is easy. For oForest, it only makes little difference if it is running a few, tens, or hundreds of experiments simultaneously. This makes it extremely attractive for multiple research institutions to join forces and share both administrators and compute resources.

## 5.8 Extensibility

oForest is open source and configured in a fully transparent fashion. This makes it easily adaptable. Researchers, for example, can provide their own `Dockerfile` or they could deploy entirely custom applications other than oTree by building a custom Helm chart.

Further, researchers and administrators alike can adapt or extend the CI/CD pipeline to include unit tests or the deployment to a testing cluster. The limits here are only really the needs of researchers and the capabilities of those needing to implement new features.

## 5.9 Limits of oForest and Further Work

In its current state, oForest is production ready and is already in use at the KD2Lab at the Karlsruhe Institute of Technology. As we mentioned before, it runs on stable technology that itself has powerful error-handling capabilities. Still, we cannot guarantee that there will be no errors at all. It is obligatory that any system that runs business-critical infrastructure is monitored by someone who knows how to handle suddenly occurring faults. We want to point out some areas that could cause problems

- **Oversized Load.** The greatest danger for an oForest system probably comes from a too weakly provisioned server. If the cluster experiences exceptionally high network traffic or disc pressure, experiments can become unresponsive. Resource quotas assure that this won't affect other experiments. Still, it is important to be able to do proper load testing and adjust the infrastructure accordingly. As we noted before, this requires further development.
- **Changing Security Requirements.** We already mentioned that we don't have security policies against the malicious intent of single researchers, that are invited into the oForest GitLab group. When scaling up operations because multiple chairs or labs want to share resources, this might be no longer acceptable. As security requirements change, naturally the architecture must change with it. Still, the changes for having isolation between organizations is doable and a manageable expense.
- **User Interface.** oForest currently makes use of GitLab as a user interface. We do this because it means one less moving part to maintain. Still, we admit that the GitLab interface is not tailor-made for the specific use case and thus lacks some of the visual clarity of e.g. Heroku. For this work, we accept this tradeoff but acknowledge that this is an area where improvements can be made.
- **Lacking Know-How.** The operation of infrastructure with oForest is well-prepared but still requires know-how. Without it, institutions might not be able to deal with errors properly. Institutions must not only build this knowledge but also have a strategy for retaining it.

Despite these limits, oForest is already in a state where it can be used to run real-world



experiments. For use in smaller institutions, the user interface and current security model pose no problem. Adding proper load testing capabilities to simulate hundreds or more participants, on the other hand, can greatly improve the flexibility of oForest. With it, researchers would have greater security that resources are certainly sufficient. The lack of know-how is the other great threat. Especially for small institutions, getting trained personnel can pose a hard problem. Here forging alliances with other institutions to share resources can be a great help.

## 6 Conclusion

In this work, we introduced oForest, a tool that streamlines and automates the deployment of oTree experiments on self-hosted infrastructure. oForest is born out of the necessity to self-host experiments instead of using cloud providers such as Heroku, due to GDPR concerns. The goals of oForest were to be easy to use, to have a high grade of automation, to keep data private and safe, to be transparent, to enable its users to be autonomous, to be able to run experiments isolated and consistently, to be secure, to be able to observe the running system and to be well-supported and documented.

oForest achieves automation by making use of technologies such as container orchestration, and automation pipelines. It uses containers to create a consistent environment where experiments run independently of each other. By using GitLab as a simple user interface, oForest allows its users to act quickly and autonomously and track their experiments' status. GitLab further provides user management which provides security safeguards for oForest. Running oForest on self-owned infrastructure makes the processing of personal data GDPR-compliant. The usage of system utilization dashboards makes oForest observable.

Further, oForest does not only automate the deployment and management of experiments but also provides automation for the infrastructure setup itself. This makes oForest ready to operate in a matter of minutes.

Administrators do not only profit from the high level of automation, but also from the rich ecosystem of documentation, resources, and a thriving community provided by tools like Kubernetes, Containers (Docker), GitLab, and Ansible. This makes oForest not only comprehensible but also extensible.

Although oForest has been already used in deploying experiments, it still needs fine-tuning. We discussed the limitations that exist in load testing and the user interface, as well as additional considerations regarding security. We especially want to highlight that oForest does not maintain itself and needs the supervision of an administrator with know-how.

Next to the implications for single organizations, we want to emphasize the scalable nature of oForest. It makes joint ventures of multiple research institutions not only possible but also attractive, as they can share both computing and administration capacity. In conclusion, we believe oForest to be a valuable tool for research institutions that want to maintain their own infrastructure.

## References

- Amazon M-Turk, (2023). Amazon Mechanical Turk. <https://www.mturk.com/>. Accessed: 28.03.2023.
- Anaconda, (2023). Anaconda | The World's Most Popular Data Science Platform. <https://www.anaconda.com/>. Accessed: 28.03.2023.
- May Baker, Monya (2016). 1,500 scientists lift the lid on reproducibility. In: *Nature*, 533 (7604), S.452–454. ISSN 1476-4687. DOI: [10.1038/533452a](https://doi.org/10.1038/533452a).
- October Baliatti, Stefano (2017). nodeGame: Real-time, synchronous, online experiments in the browser. In: *Behavior Research Methods*, 49(5), S.1696–1715. ISSN 1554-3528. DOI: [10.3758/s13428-016-0824-z](https://doi.org/10.3758/s13428-016-0824-z).
- Beck, Hanno (2014). Behavioral economics: eine Einführung. Lehrbuch. Springer Gabler, Wiesbaden. ISBN 978-3-658-03366-8. DOI: [10.1007/978-3-658-03367-5](https://doi.org/10.1007/978-3-658-03367-5).
- Bielefeld, R. Selten (1988). Reexamination of the perfectness concept for equilibrium points in extensive games In: *Models of Strategic Rationality*, S.1–31. Springer Netherlands, Dordrecht. ISBN 978-94-015-7774-8. DOI: [10.1007/978-94-015-7774-8\\_1](https://doi.org/10.1007/978-94-015-7774-8_1).
- March Camerer, Colin F.; Dreber, Anna; Forsell, Eskil; Ho, Teck-Hua; Huber, Jürgen; Johannesson, Magnus; Kirchler, Michael; Almenberg, Johan; Altmejd, Adam; Chan, Taizan; Heikensten, Emma; Holzmeister, Felix; Imai, Taisuke; Isaksson, Siri; Nave, Gideon; Pfeiffer, Thomas; Razen, Michael; Wu, Hang (2016). Evaluating replicability of laboratory experiments in economics. In: *Science*, 351(6280), S.1433–1436. ISSN 0036-8075, 1095-9203. DOI: [10.1126/science.aaf0918](https://doi.org/10.1126/science.aaf0918).
- March Chen, Daniel L.; Schonger, Martin; Wickens, Chris (2016). oTree—An open-source platform for laboratory, online, and field experiments. In: *Journal of Behavioral and Experimental Finance*, 9 S.88–97. ISSN 2214-6350. DOI: [10.1016/j.jbef.2015.12.001](https://doi.org/10.1016/j.jbef.2015.12.001).
- Chilton, Lydia B.; Sims, Clayton T.; Goldman, Max; Little, Greg; Miller, Robert C. Seaweed: A web application for designing economic games In: *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '09, S.34–35, New York, NY, USA (2009). Association for Computing Machinery. ISBN 978-1-60558-672-4. DOI: [10.1145/1600150.1600162](https://doi.org/10.1145/1600150.1600162).
- Cri-o, (2023). Cri-o | lightweight container runtime for kubernetes. <https://cri-o.io/>. Accessed: 28.03.2023.
- Diamond, Douglas W.; Dybvig, Philip H. (1983). Bank runs, deposit insurance, and liquidity. In: *Journal of Political Economy*, 91(3), S.401–419. DOI: [10.1086/261155](https://doi.org/10.1086/261155).

- Durlauf, Steven N.; Blume, Lawrence E., editors (2010). Behavioural and Experimental Economics. Palgrave Macmillan UK, London. ISBN 978-0-230-23868-8 978-0-230-28078-6. DOI: [10.1057/9780230280786](https://doi.org/10.1057/9780230280786).
- June Engle-Warnick, Jim; Turdaliev, Nurlan (2010). An experimental test of Taylor-type rules with inexperienced central bankers. In: *Experimental Economics*, 13(2), S.146–166. ISSN 1573-6938. DOI: [10.1007/s10683-010-9233-9](https://doi.org/10.1007/s10683-010-9233-9).
- August Falk, Armin; Fehr, Ernst (2003). Why labour market experiments?. In: *Labour Economics*, 10(4), S.399–406. ISSN 09275371. DOI: [10.1016/S0927-5371\(03\)00050-2](https://doi.org/10.1016/S0927-5371(03)00050-2).
- Fischbacher, Urs (1999). Z-Tree: Zurich toolbox for ready-made economic experiments. In: *Experimental Economics*, 10(2), S.171–178. ISSN 1386-4157, 1573-6938. DOI: [10.1007/s10683-006-9159-4](https://doi.org/10.1007/s10683-006-9159-4).
- GitLab, (2023). GitLab. <https://about.gitlab.com/platform/>. Accessed: 28.03.2023.
- Grafana, (2023). Grafana OSS | Metrics, logs, traces, and more. <https://grafana.com/oss/grafana/>. Accessed: 28.03.2023.
- December Harrison, Glenn W.; List, John A. (2004). Field Experiments. In: *Journal of Economic Literature*, 42(4), S.1009–1055. ISSN 0022-0515. DOI: [10.1257/0022051043004577](https://doi.org/10.1257/0022051043004577).
- Helm, (2023). Helm | the package manager for kubernetes. <https://helm.sh/>. Accessed: 28.03.2023.
- Kahneman, Daniel (2011). Thinking, Fast and Slow. Allen Lane, London. ISBN 978-1-84614-606-0 978-1-84614-055-6.
- Kahneman, Daniel; Tversky, Amos (1979). Prospect Theory: An Analysis of Decision under Risk. In: *Econometrica*, 47(2), S.263–291. ISSN 0012-9682. DOI: [10.2307/1914185](https://doi.org/10.2307/1914185).
- Kaniko, (2018). Kaniko - Build Images In Kubernetes. <https://github.com/GoogleContainerTools/kaniko>. Accessed: 28.03.2023.
- Kim, Gene; Humble, Jez; Debois, Patrick; Willis, John (2021). The DevOps handbook: how to create world-class agility, reliability, & security in technology organizations. IT Revolution Press, LLC, Portland, OR, second edition edition. ISBN 978-1-950508-40-2.
- Kubernetes, (2023). Kubernetes: Production-Grade Container Orchestration. <https://kubernetes.io/>. Accessed: 28.03.2023.
- May Levitt, Steven D; List, John A (2007). What Do Laboratory Experiments Measuring Social Preferences Reveal About the Real World?. In: *Journal of Economic Perspectives*, 21(2), S.153–174. DOI: [10.1257/jep.21.2.153](https://doi.org/10.1257/jep.21.2.153).

- May McKnight, Mark E.; Christakis, Nicholas A. (2016). Breadboard: Software for Online Social Experiments. Vers. 2.. Yale University.
- Microk8s, (2023). MicroK8s - Zero-ops Kubernetes for developers, edge and IoT | MicroK8s. <http://microk8s.io>. Accessed: 28.03.2023.
- Open Container Initiative, (2023). Open Container Initiative - Open Container Initiative. <https://opencontainers.org/>. Accessed: 28.03.2023.
- oTree Hub, (2023). oTree Hub: The Central Place To Build, Launch, and Monitor oTree Experiments. <https://www.otreehub.com/>. Accessed: 28.03.2023.
- Penguin Randomhouse, (2023). Copies sold of Nudge by Richard H. Thaler, Cass R. Sunstein on PenguinRandomHouse.com. <https://www.penguinrandomhouse.com/books/690485/nudge-by-richard-h-thaler-and-cass-r-sunstein/>. Accessed: 28.03.2023.
- Podman, (2023). Podman. <https://podman.io/>. Accessed: 28.03.2023.
- Prometheus, (2023). Prometheus. <https://prometheus.io/docs/introduction/overview/>. Accessed: 28.03.2023.
- May Red Hat, (2022). What is container orchestration?. <https://www.redhat.com/en/topics/containers/what-is-container-orchestration>. Accessed: 28.03.2023.
- May Reips, Ulf-Dietrich; Neuhaus, Christoph (2002). WEXTOR: A Web-based tool for generating and visualizing experimental designs and procedures. In: *Behavior Research Methods, Instruments, & Computers*, 34(2), S.234–240. ISSN 1532-5970. DOI: [10.3758/BF03195449](https://doi.org/10.3758/BF03195449).
- Sauermann, Heinz; Selten, Reinhard (1959). Ein Oligopolexperiment. In: *Zeitschrift für die gesamte Staatswissenschaft / Journal of Institutional and Theoretical Economics*, 115(3), S.427–471. ISSN 0044-2550.
- August Schneller, Daniel; Pustina, Lukas (2015). Operations heute und morgen, Teil 3: Virtualisierung und Containerisierung. <https://www.heise.de/ratgeber/Operations-heute-und-morgen-Teil-3-Virtualisierung-und-Containerisierung-2762412.html>. Accessed: 28.03.2023.
- April Shotton, Richard (2014). Fast and slow lessons for marketers. In: *The Guardian*. ISSN 0261-3077. Accessed: 28.03.2023.
- March Smith, Vernon L. (1994). Economics in the Laboratory. In: *Journal of Economic Perspectives*, 8(1), S.113–131. ISSN 0895-3309. DOI: [10.1257/jep.8.1.113](https://doi.org/10.1257/jep.8.1.113).

Thaler, Richard H.; Sunstein, Cass R. (2008). *Nudge: Improving Decisions about Health, Wealth, and Happiness*. Yale University Press, New Haven. ISBN 978-0-300-12223-7.

June Walsh, Dan (2020). How to use the `--privileged` flag with container engines. <https://www.redhat.com/sysadmin/privileged-flag-container-engines>. Accessed: 28.03.2023.

Jahr	Autor	Title	Contents	Relevance
2016	Baker, M.	Is There a Reproducibility Crisis?	52% of questioned researchers said there is a significant crisis, 38% said its a slight crisis. Reasons are among others: Selective reporting, pressure to publish and methods or code being unavailable.	A survey that shows the reasons for the reproducibility crisis.
2014	Beck, H.	Behavioral economics: eine Einführung	The history and foundations of behavioral economics and the influence it has on the real world	A textbook that provide a good introduction to behavioral economics
2016	Camerer, C. F. et al.	Evaluating replicability of laboratory experiments in economics	Looked to replicate laboratory experiments and, on average, were able to only replicate 66% of the effect size of the original study. In their conclusion, they call for well-designed and documented experiments with replication in mind. This documentation goes hand in hand with the publication of data and the means of collecting this data	Shows the importance of availability of experiment code and that the survey by Baker is also relevant for behavioral economics.
2016	Chen, D. L. et al.	oTree—An open-source platform for laboratory, online, and field experiments	This paper introduces oTree and shows how it can be use in experimentation in the lab, the field and online. Also the architecture of oTree is explaind.	oTree is the Python tool researchers build their experiments with that get then deployed on oForest.

Table 2: Literature Table for Economical Sources, Part I

Jahr	Autor	Title	Contents	Relevance
2010	Durlauf, S. N. and Blume, L. E.	Behavioural and Experimental Economics	Shows how behavioral economics are used in finance, market & mechanism design and game theory.	A collection of articles that shows areas of applications in behavioral economics.
2003	Falk, A. and Fehr, E.	Why labour market experiments?	Shows the importance of behavioral economics in labour market experiments.	Helps in introducing behavioral economics and its relevance in different fields.
1999/ 2007	Fischbacher, U	Z-Tree: Zurich toolbox for ready-made economic experiments	The introduction of Z-Tree, a tool for creating behavioral economics experiments.	z-Tree is the spiritual predecessor of oTree and has shaped experimental economics.
2004	Harrison, G. W. and List, J. A.	Field Experiments	The importance and design of field experiments	Helps in introducing behavioral economics, experimental economics and its relevance in different fields.
2011	Kahneman, D.	Thinking, Fast and Slow	The bestseller that brought Kahnemans' theory to the general public	Helps to show the relevance of behavioral economics and how it also influences public discourse.
2013	Kahneman, D. and Tversky	Prospect theory: An analysis of decision under risk	prospect theory suggests that people's decisions are not always rational and objective, but rather influenced by how the choices are presented to them and the potential losses and gains involved. It proposes that individuals value losses more than equivalent gains and are risk-averse when facing gains but risk-seeking when facing losses.	Helps in introducing behavioral economics and experimental economics and its relevance in different fields.

Table 3: Literature Table for Economical Sources, Part II



Jahr	Autor	Title	Contents	Relevance
2007	Levitt, S. D. and List, J. A.	What Do Laboratory Experiments Measuring Social Preferences Reveal About the Real World?	The paper argues that while laboratory experiments on social preferences can provide valuable insights, there are limitations to their generalizability to real-world settings. The find at least five factors that influence behavior in lab experiments aside from monetary incentives.	Helps to show how z-Tree that is only usable in lab settings might be of debateable usefulness.
1959	Sauermann, H. and Selten, R.	Ein Oligopolexperiment	A laboratory experiment to investigate the behavior of firms in oligopoly markets	Helps to show the relevance of behavioral economics and how it also influencend public discourse.
1994	Smith, V. L.	Economics in the Laboratory	An overview of functions of experiments and a collection of results.	Helps in introducing experiments in economics by illustrating their important functions.
2008	Thaler, R. H. and Sunstein	Nudge: Improving Decisions about Health, Wealth, and Happiness	The book proposes a behavioral economics approach to policy design that involves making small, subtle changes to the environment in which people make decisions to encourage more desirable behaviors.	Helps to show the relevance of behavioral economics and how it also influencend public discourse.

Table 4: Literature Table for Economical Sources, Part III

Name	Description	Usage
Amazon M-Turk	Amazon Mechanical Turk (MTurk) is a crowdsourcing website for businesses to hire remotely located crowdworkers to perform discrete on-demand tasks that computers are currently unable to do.	oTree allows researchers to use Amazon M-Turk to do large scale experiments.
Anaconda	Popular Python Data Science Platform with thousands of packages pre-installed.	Anaconda is the base image for experiment containers.
Docker	A popular container runtime	Used to illustrate how containers work
GitLab	GitLab’s all-in-one project planning, source code management, CI/CD, and security platform provides a single source of truth for all users, regardless of role.	GitLab is used as user interface and a hub for all automation
Grafana	Dashboards for observability	Grafana is used to monitor the state of the oTree cluster.
Kubernetes	Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications.	Kubernetes powers the cluster where oTree applications are deployed.
Prometheus	Prometheus collects observability metrics	Prometheus is the data source for the observability dashboards.
Walsh, D	Article on how to use the <code>--privileged</code> flag with container engines.	Arguments on why not to use privileged containers in the CI pipeline.

Table 5: Literature Table for Technical Sources

## Eidesstattliche Erklärung

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.

Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn die Erklärung nicht erteilt wird.

Berlin, 30. März 2023

Jasper Anders