

Emulating a Continuous Deployment Environment Using Locally Run GitHub Actions and a Containerized Deployment Server

Boris Milašinović, Juraj Dončević, Krešimir Fertalj

0000-0002-7889-3131

0000-0001-5221-6848

0000-0001-7469-6168

University of Zagreb Faculty of Electrical Engineering and Computing

Unska 3, 10000 Zagreb, Croatia

Email: {boris.milasinoVIC, juraj.doncevic, kresimir.fertalj}@fer.unizg.hr

Abstract—To align with standard IT practices, continuous practices are essential in software development courses. The lack of free cloud hosting solutions, or their unsuitability for academic content, hinders the acquisition of practical knowledge in continuous deployment and forces teachers to focus on on-premise solutions. The integration of the GitHub environment, consisting of a GitHub repository and GitHub Actions, with an on-premise server for deployment was the focus of the authors' previous work, which contributed a deployment workflow combining GitHub Actions and an on-premise server where students' applications are hosted on a shared server. This paper extends previous results by presenting a novel approach in which the deployment environment is simulated using Docker containers and combined with *nektos/act* for running GitHub Actions locally. This allows both teachers and students to emulate the complete workflow, including deployment, and experiment with workflows locally, thus saving GitHub build minutes.

I. INTRODUCTION

PERCEIVING the software developer as someone who only does development has become a historical relic in the IT industry. A modern developer should be involved not only in development but also in deployment and production operations. If they do not participate actively in deployment, they should at least be aware of how it works. The importance of this approach is also recognized in the academic context, which emphasizes that the delivery aspect is often missing, even though it should be a fundamental practice, continuously applied and not limited to a single course [1].

Ten years ago, the traditional curriculum focused only on development, design, and construction. In contrast to the relatively large number of studies on continuous integration, delivery, and deployment [2], very few papers addressed teaching DevOps, and the average university teacher lacked expertise in DevOps practices [3]. However, recent work shows that, despite the initial assumption, there is no overall statistically significant gap between higher education and industry regarding DevOps practices [4], and the number of papers has increased. For example, Garcia et al. [5] found 248 papers (within a five-year window) containing the keywords *devops* and *education*, narrowing them to 27 papers related to

DevOps teaching, categorizing teaching techniques, and identifying challenges. These findings show that the initial technical problems have been overcome, but financial constraints present in the early days [6] are still causing problems. Although initially very appealing and trendy, most commercial cloud providers have removed or reduced their free offerings, forcing teachers to focus on alternative PaaS options [7].

The need for a practical course in which continuous practices are not only taught theoretically or concluded at the continuous integration step has already been discussed in the authors' previous paper [8]. That paper presents a case study where continuous deployment is achieved using a combination of GitHub Actions and an on-premises server to address the lack of resources for cloud hosting solutions. While the solution is satisfactory and the deployment workflow has been used for the current academic year, the number of available GitHub Actions build minutes must be considered. Previously, GitHub Education build minutes were counted per repository, but they are now limited to 3000 minutes per month for the entire organization. Under normal circumstances, this limit is sufficient for 10 to 15 groups of students, but it requires careful planning. The identified problem, combined with the maturity of the *nektos/act* tool [9], which allows running GitHub Actions locally, inspired the authors to follow up on their previous paper with several novel elements. While *nektos/act* saves GitHub build minutes and helps address issues with incorrect or missing setup of variables and secrets, the mismatch between the local setup and the production server remains a challenge. This issue often requires frequent commits and pushes, leading to a trial-and-error strategy until all paths and configurations are corrected. By utilizing Docker, a replica – or more precisely, an emulation – of the production environment can be created on a student's local machine. This approach allows students not only to run actions locally to check if their code breaks during the build process, but also to see how their code would function on the production server.

The rest of the paper is organized as follows. The second section briefly summarizes the main ideas of the workflow pre-

sented in [8], describes modifications to the original workflow, and outlines the basic properties of *nektos/act* in the context of the modified workflow. Because a complete replica or twin of the production server would require a Linux virtual machine, a lightweight solution using Docker containers has been adopted. This approach requires different technical solutions, which are discussed in the third chapter. The fourth chapter discusses the pros and cons of the approach and provides a reflection on existing solutions, followed by the conclusion section.

The zip file necessary for the demonstration of the concept can be downloaded from <https://doi.org/10.6084/m9.figshare.31999572>. Prerequisites to run the demo are Docker and *nektos/act*¹. The zip file contains a README file with instructions to run the demo, and figures that illustrate how the proposed solution works.

II. DEPLOYMENT WORKFLOW

Figure 1 shows the modified workflow from [8]. The workflow is triggered by changes to the main branch and can also be run manually using the GitHub interface. Most steps remain unchanged, with minor updates to action versions and adjustments in the initial steps for variable resolution using repository variables. The only exception is *appleboy/ssh-action*, which has been downgraded to v1.0.3 because later versions do not work on Windows. While this is not an issue for production on GitHub Actions, it is very important for local use. The initial version of the workflow² assumed that students would add repository secrets for the username and password and modify the *dotnet.yml* file in the *.github/workflows/* directory by entering their group number. This has now been changed to use repository variables for the group number, all of which are checked in the first step of the workflow.

The GitHub Education pack allows 3000 build minutes per month. While this limit was initially set per repository, it is now set at the organization level, meaning all students share the same limit. To use Actions minutes efficiently, students are advised to follow a modified GitHub Flow, where branches are created for feature development and quickly merged and deleted, but deployment occurs only from the main branch. With this approach, we have been able to manage courses with 10-15 groups without issues. The use of minutes among groups was not balanced, ranging from as little as 14 minutes for 5 workflow runs during the homework submission month to 134 minutes for 50 workflow runs, with an additional 78 minutes spent by Copilot on pull requests. Even if the group with the highest usage is used as a reference, it would still be feasible for 10 groups to remain below 3000 minutes per month. However, this assumption cannot be generalized, and it is not uncommon to see cases where all minutes are used very quickly³, for example, within two days [10].

The workflow from Figure 1 is divided into two jobs, which may be elegant to display in the diagram and to view in the

Actions logs, but it is inefficient in terms of Actions usage. Since Actions minutes are rounded up to the nearest minute⁴, having a separate short job is not a good approach, as even one second job will be billed as one minute, which can be very frustrating⁵. Therefore, for the sake of simplicity in the paper, the workflow will keep its structure as is, but a future version of the *dotnet.yml* file will be optimized so that *Setup variables* is not a separate job, but part of *Build and deploy*, in which *Check secrets* will fail (e.g., with `exit 1`) if variables are not set correctly.

A. Running Deployment Workflow Locally

Act [9] is a library that allows running GitHub Actions locally, which is especially useful in our scenario for both teachers and students. For teachers, every change to a file in *.github/workflows/* would require committing and pushing the change, resulting in many commits which later need to be squashed, consuming Actions minutes for a trial-and-error approach. The main cause of errors and repeated commits and pushes from students is related to URL paths in the web application. Locally, they run their application at the root of localhost, but on the server, the application have a base path which must be considered. Other common mistakes involve restoring third-party libraries or other git-ignored files, invalid configuration settings (such as a missing connection string in the *appsettings.json* file while using a secrets file locally), and similar issues. Finally, mistakes in variables setup occur frequently, but mainly in the initial phase when students are just starting to develop their homework.

During the initial setup, *Act* prompts the user to choose a GitHub Runner, offering micro, medium, and large Docker images⁶. In our case, with a .NET build pipeline, the micro image was insufficient, while the medium image ran the jobs correctly. GitHub Actions allow the definition of secrets and variables, which are available in workflow steps. The distinction is that secrets are masked with asterisks if printed in the workflow. *Act* supports defining secrets by entering them in the *.secrets* file or by passing the filename to the *act* command. The same approach is used for environment and configuration variables⁷, with default filenames *.env* and *.vars*, respectively. Running the original workflow locally but deploying to the real server would not be very useful if a student wants to see the effect of changes without affecting the current solution with a faulty deployment. Therefore, the workflow has been modified to use additional configuration variables for deployment, such as the server address and SSH port of the local deployment server. The complete YAML file for the deployment is available at <https://doi.org/10.6084/m9.figshare.31999572>, and the next section describes how to build the Docker image for the local deployment server.

⁴<https://docs.github.com/en/billing/reference/actions-runner-pricing>

⁵<https://github.com/orgs/community/discussions/8726>

⁶<https://nektosact.com/usage/runners.html>

⁷<https://docs.github.com/en/actions/how-tos/write-workflows/choose-what-workflows-do/use-variables>

¹<https://nektosact.com/installation/index.html>

²<https://figshare.com/s/1d930df955658ed2b11e>

³<https://github.com/community/Global-Campus-Teachers/discussions/475>

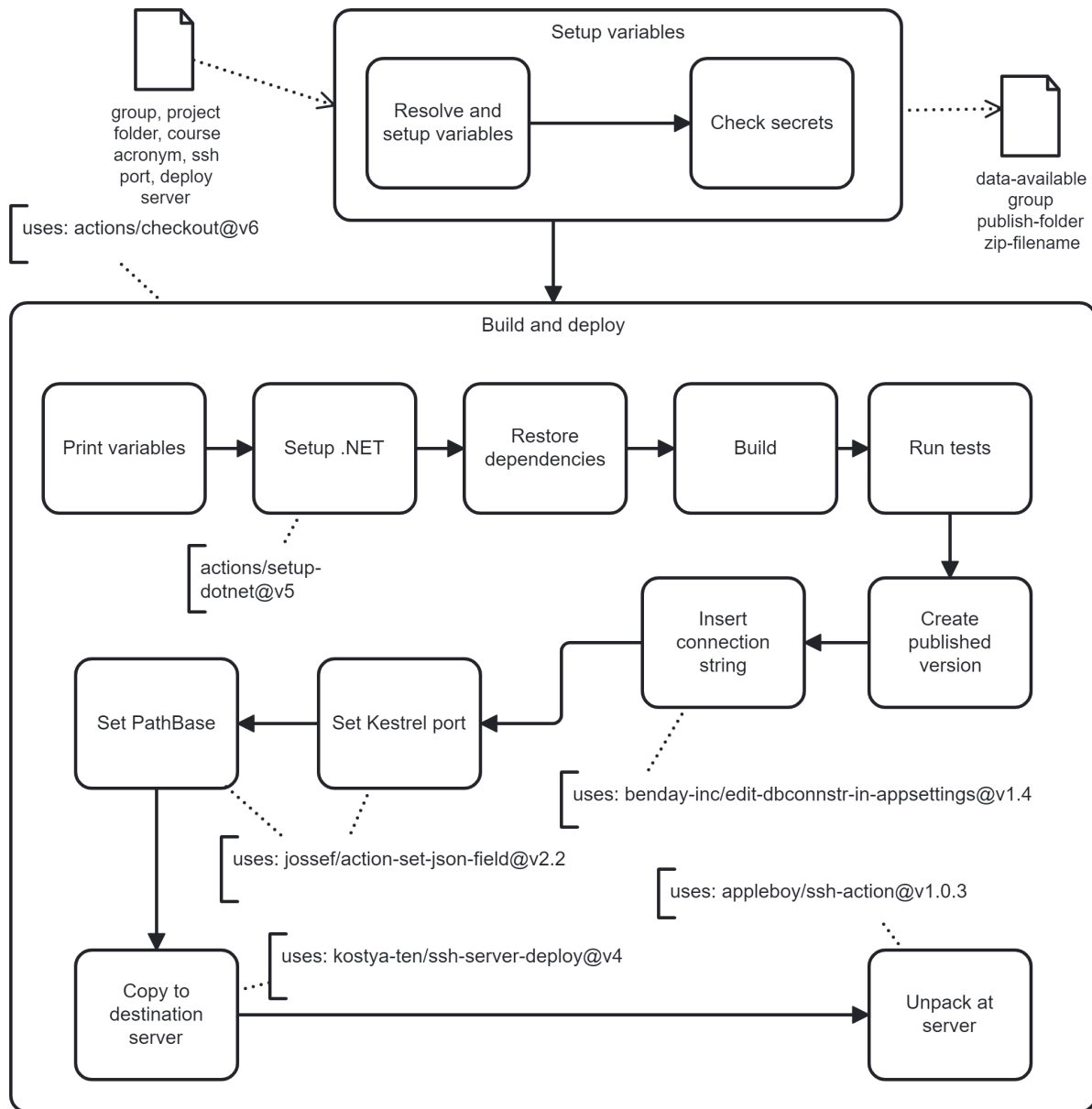


Fig. 1. Deployment Workflow

III. TECHNICAL DETAILS

Creating an identical replica of the server would require creating a Linux virtual machine, which presents several challenges. Different operating systems use different types of virtualizers, necessitating the creation of multiple definitions for common hypervisors, with the hope that all of them would support VHDX or another common virtual disk image. Even if this were successful, distributing a large virtual disk would be highly inconvenient. An alternative would be to require students to install the necessary Linux distribution themselves, distributing only setup scripts. This approach is feasible and does not suffer from heterogeneous environments, but it still

requires students to install a hypervisor, which is less common and demands more memory and disk space compared to the lightweight solution of using Docker containers.

Files required to initialize the deployment container are shown in Listing 1 and available in `deployment-server` folder of the provided demo⁸. The folder contains Docker configuration files, and files needed to initialize container users and folders (e.g., `passwords.txt`), define permissions, and specify running processes.

The `docker-compose.yml` file is straightforward, defining the container name, hostname, and exposed ports and map-

⁸<https://doi.org/10.6084/m9.figshare.31999572>

Listing 1 Required Files for the Container Setup

```
certs
  server.crt
  server.key
docker-compose.yml
docker-entrypoint.sh
Dockerfile
nginx-main.conf
passwords.txt
restart-or-start-service.sh
supervisord.conf
```

pings (i.e., 2222 instead of 22 for SSH, and 443 for HTTPS). Additionally, it passes the expected application filename and course acronym to the `Dockerfile`.

The largest and most complex file is the `Dockerfile`, which uses input from the other files and builds the image based on their values. Algorithm 1 describes the steps of the build process. Because containers are not full virtual machines, some changes specific to containers are required; these are explained in the following subsections.

Algorithm 1 Container image building steps

Require: Ubuntu base image compatible with .NET, e.g., 24.04

- 1: Resolve the arguments (application DLL name, folder, and services names prefix)
 - 2: Install `ssh`, `nginx`, `supervisor`, `incron`, `.NET`, etc.
 - 3: Set locales
 - 4: Create directories for `ssh`, `incron`, `nginx`, build artifacts, and application folders
 - 5: Set SSH authentication
 - 6: Copy required files to the image
 - 7: Validate the password file
 - 8: Create the static part of `nginx.conf`
 - 9: **for all** `passwords` in `passwords.txt` **do**
 - 10: Create a user account and its web-application directory
 - 11: Set ownership and permissions on the application directory
 - 12: Append a reverse proxy location to the `nginx` configuration
 - 13: Append a process definition to `supervisord`
 - 14: Append a restart rule to `incron`
 - 15: **end for**
 - 16: Validate the `nginx` configuration
 - 17: Remove temporary build files
 - 18: Expose ports 22 and 443
 - 19: Set the entrypoint script and start `supervisord` by default
-

A. Users, Folders, and Permissions

Although from a student's perspective only their group would be sufficient, a complete setup with all groups has been implemented. The benefit of this approach is more apparent from an infrastructure standpoint, as the same scripts can be used to set up a new server in case of reinstallation or hardware failure, or even if we choose to use a Docker container instead of a physical server. All usernames share the same prefix (the course acronym) followed by the group number formatted to two digits for ordering purposes. With this approach, the `passwords.txt` file determines the number of users. Before building the image, a student should modify this file, as the template contains only a few lines with demo passwords. For each user, a folder `/var/www/course-acronym/XX` is created, and ownership is assigned to the user. Because .NET processes are registered to run under `www-data`, additional permissions must be set using `setfacl` command. This cannot be done at the image level and should be performed after the container is created, which is why `docker-entrypoint.sh` exists.

B. Running Multiple Processes

Although the official Docker documentation⁹ recommends one process per container as best practice, it also permits running multiple processes using Supervisor. The real, physical server runs not only `nginx` but also an SSH server, which allows students to connect to the server to check logs and, although not intended, possibly make manual changes. It also runs one .NET service per group and `incron` to detect changes upon new deployment and restart the service. Unlike on Windows, where the restart occurs automatically when web application files are changed, on Linux the service for the web application must be restarted manually, which is accomplished using `incron` and `systemctl` (on the physical server) or `supervisorctl` (in the container). Restarting the service on the real server using `systemctl` always starts or restarts the service regardless of its current state, which is not the case with `supervisorctl`, so the `restart-or-start-service.sh` script is required.

C. nginx

The configuration for `nginx` is essentially the same as on the real server, with the main difference being related to the server certificates. The real server is publicly accessible and can use Let's Encrypt¹⁰ or similar tools to generate certificates, which is not possible for those running locally. The choice is whether to create a self-signed certificate, which will generate a browser warning until accepted, or to run over HTTP. The former approach, despite the certificate warning, is preferable as it avoids mixing HTTP and HTTPS. Part of the configuration is static, while the rest is generated based on the group number, using the same concept as in [8], e.g. where the port is calculated as $50000 + 100 \cdot groupnumber$.

⁹https://docs.docker.com/engine/containers/multi-service_container

¹⁰<https://letsencrypt.org/>

D. SSH

The part of the `Dockerfile` related to SSH setup enables SSH login and disables it for the root user as a precaution. If each container generated its own host keys, deleting a container and creating another from the same image would result in a different fingerprint. SSH would then reject the connection until the stored fingerprint is deleted, which is inconvenient. Therefore, the build process creates keys at the image level, ensuring that as long as the image remains the same, every container instance has the same host key, regardless of the variables used to instantiate the image.

IV. DISCUSSION AND RELATED WORK

Students need to deploy and manage their applications without interfering with others, while being organized in teams with permissions set accordingly [7]. The proposed solution satisfies these requirements because each group has its own folder, service, and permissions set on the server. They can not only continuously deploy their solution independently of others' settings and permissions, but also test it locally before pushing commits to GitHub, reducing both time and the possibility of error. The approach in which the deployment workflow is already created and the homework solution template is provided to students, leaving them only to enter variables, could lead students to approach continuous practices superficially. However, this method minimizes setup risk and aligns with [11], which considers such aspects as implementation details and encourages students to understand the concepts by putting them into practice. Students better understand concepts if they can experience them in practice [12].

Paez and de la Fuente [11] used GitLab, Docker, Heroku, and Kubernetes, but the Kubernetes service was paid. Another similar setup [13] used Docker and Kubernetes with Google Cloud Platform education credits. Generally, Docker is widely used, either as a necessary step before deployment or as a solution for assignment evaluation because it provides a sandbox for potentially dangerous programs [14]. In our approach, packaging students' applications into Docker containers would give us two options, neither of which is better than the presented solution. Either we would need to configure a reverse proxy to work with the containers, which is more complicated and memory-demanding compared to the current solution, or we would have to download containers locally for manual review, losing the ability to review applications from anywhere, even on tablets.

Although there are many solutions for continuous practices, both in the cloud and on-premises, several studies [15], [16] note the dominance of GitHub Actions. In just 18 months, GitHub Actions replaced Travis as the dominant tool [15], [17], receiving a positive reception among developers [18]. One reason for this is its support for reusable actions [19]. Therefore, choosing the GitHub ecosystem (GitHub repository, GitHub Actions, GitHub Classroom) is straightforward, though not without risks. While a study by Kinsman et al. showed that a typical Action does not require ongoing

maintenance [18], research by Decan et al. [20] shows that Actions are continuously updated, leading to many workflows using outdated versions. Other risks include the possibility that some extensions may be removed or even used in supply chain attacks [21], but the same problem can occur in any packaging ecosystem.

Act was introduced by its developer Casey Lee in 2020¹¹ and has proven to be an important tool for running various analyses of GitHub workflows, such as detecting code smells [22], creating reproducible benchmarks [23], and identifying issues and enhancing the efficiency of CI/CD pipelines, although it does not fully support all features of GitHub Actions [24]. The benefit of using *Act* for a try-and-fix approach during workflow creation is clear, as it saves build minutes, speeds up the process, and does not require the numerous commits that would otherwise need to be squashed to keep the main trunk clear.

V. CONCLUSION

Using continuous deployment in software development courses aligns with current trends, where students are required to be familiar not only with the development phase of the software lifecycle but also with deployment and operations. As the use of cloud solutions for deployment can be financially constrained, teachers need to balance available options, such as combining the GitHub ecosystem which includes GitHub Actions with 3000 free build minutes per month and an on-premises server for deploying students' web applications.

The presented solution builds on a successful application of this approach by adding the ability to run GitHub Actions locally and deploy to a local deployment server. This method allows both teachers and students to experiment with available actions and create various deployment workflows without worrying about using up free build minutes. By emulating a deployment server with Docker containers, students can fully test their solutions before pushing changes to GitHub, without concern that their deployment will cause problems in production, thus avoiding unnecessary commits and saving build minutes. Furthermore, this approach makes it possible to quickly replace a physical server with a Docker container setup. The proposed solution is lightweight and easy to adapt to specific needs. If the programming language or application type changes, the build step concept remains the same, and only parts of the build scripts need to be updated to use the required packages, while the deployment process stays consistent.

To paraphrase common Docker phrases about promoting local environments to production, the proposed solution *brings production to your home directory* by using Docker to replicate or emulate production on a student's local machine, combined with locally run GitHub Actions using *nektos/act*.

ACKNOWLEDGMENT

During the preparation of this work, the authors used *Codex GPT-5.4* to assist in writing the container configuration files

¹¹<https://cplee.org>

and shell scripts. The prompts were based on the current server configuration and guided by the steps of the novel idea presented in the paper. The generated files have been thoroughly examined. The text has been proofread using *InstaText*. The authors take full responsibility for the content of this publication.

REFERENCES

- [1] R. Chatley and I. Procaccini, "Threading devops practices through a university software engineering programme," in *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*, 2020. doi: 10.1109/CSEET49119.2020.9206211 pp. 1–5.
- [2] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017. doi: 10.1109/ACCESS.2017.2685629
- [3] H. B. Christensen, "Teaching devops and cloud computing using a cognitive apprenticeship and story-telling approach," in *Proceedings of the 2016 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITICSE '16. New York, NY, USA: Association for Computing Machinery, 2016. doi: 10.1145/2899415.2899426. ISBN 9781450342315 p. 174–179.
- [4] M. Á. Sánchez-Cifo, P. Bermejo, and E. Navarro, "Devops: Is there a gap between education and industry?" *Journal of Software: Evolution and Process*, vol. 35, no. 12, p. e2534, 2023. doi: <https://doi.org/10.1002/smr.2534>
- [5] P. Garcia, J. Ferreira, M. Gonçalves, T. Carneiro, E. Figueiredo, and I. Pereira, "Current devops teaching techniques: A systematic literature review," in *Anais do XXXVIII Simpósio Brasileiro de Engenharia de Software*. Porto Alegre, RS, Brasil: SBC, 2024. doi: 10.5753/sbes.2024.3503. ISSN 2833-0633 pp. 389–398.
- [6] R. T. Mason, W. Masters, and A. Stark, "Teaching agile development with devops in a software engineering and database technologies practicum," in *Proceedings of the 3rd International Conference on Higher Education Advances*. Universitat Politècnica València, 6 2017. doi: 10.4995/HEAD17.2017.5607. ISBN 9788490485903
- [7] R. Chatley, J. Bailey, I. Procaccini, Z. Amin, E. Fraca, R. Ambati, A. Dharmadikari, T. Hughes, A. Mittal, A. Purohit, and S. Wadhvani, "Creating a student-friendly paas platform: Experiences with tsuru in software engineering education," in *Proceedings of the 33rd ACM International Conference on the Foundations of Software Engineering*, ser. FSE Companion '25. New York, NY, USA: Association for Computing Machinery, 2025. doi: 10.1145/3696630.3727239. ISBN 9798400712760 p. 817–822.
- [8] B. Milašinović, J. Dončević, and K. Fertalj, "Implementing continuous deployment with github actions and on-premises servers in software development education," in *Proceedings of 20th Iberian Conference on Information Systems and Technologies (CISTI 2025)*, A. Rocha, F. G. Peñalvo, C. J. Costa, and R. Gonçalves, Eds. Cham: Springer Nature Switzerland, 2026. ISBN 978-3-032-10929-3 pp. 231–242.
- [9] C. Lee, "act: Run your github actions locally," 2026, accessed: 2026-04-12. [Online]. Available: <https://github.com/nektos/act>
- [10] Y.-C. Tu, V. Terragni, E. Tempero, A. Shakil, A. Meads, N. Giacaman, A. Fowler, and K. Blincoe, "Github in the classroom: Lessons learnt," in *Proceedings of the 24th Australasian Computing Education Conference*, ser. ACE '22. New York, NY, USA: Association for Computing Machinery, 2022. doi: 10.1145/3511861.3511879. ISBN 9781450396431 p. 163–172.
- [11] N. Paez and H. de la Fuente, "Software engineering education meets devops: an experience report," in *2022 IEEE Biennial Congress of Argentina (ARGENCON)*, 2022. doi: 10.1109/ARGENCON55245.2022.9940102 pp. 1–7.
- [12] B. P. Eddy, N. Wilde, N. A. Cooper, B. Mishra, V. S. Gamboa, K. M. Shah, A. M. Deleon, and N. A. Shields, "A pilot study on introducing continuous integration and delivery into undergraduate software engineering courses," in *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, 2017. doi: 10.1109/CSEET.2017.18 pp. 47–56.
- [13] M. Hills, "Introducing devops techniques in a software construction class," in *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*, 2020. doi: 10.1109/CSEET49119.2020.9206183 pp. 1–5.
- [14] F. Špaček, R. Sohlich, and T. Dulík, "Docker as platform for assignments evaluation," *Procedia Engineering*, vol. 100, pp. 1665–1671, 2015. doi: <https://doi.org/10.1016/j.proeng.2015.01.541> 25th DAAAM International Symposium on Intelligent Manufacturing and Automation, 2014.
- [15] M. Golzadeh, A. Decan, and T. Mens, "On the rise and fall of ci services in github," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2022. doi: 10.1109/SANER53432.2022.00084 pp. 662–672.
- [16] M. Wessel, J. Vargovich, M. A. Gerosa, and C. Treude, "Github actions: The impact on the pull request process," *Empirical Software Engineering*, vol. 28, no. 6, p. 131, 11 2023. doi: 10.1007/s10664-023-10369-w
- [17] M. Wessel, T. Mens, A. Decan, and P. R. Mazrae, *The GitHub Development Workflow Automation Ecosystems*. Springer, Cham, 2023, pp. 183–214. ISBN 978-3-031-36060-2
- [18] T. Kinsman, M. Wessel, M. A. Gerosa, and C. Treude, "How do software developers use github actions to automate their workflows?" in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. Institute of Electrical and Electronics Engineers Inc., 2021. doi: 10.1109/MSR52588.2021.00054 pp. 420–431.
- [19] P. Rostami Mazrae, T. Mens, M. Golzadeh, and A. Decan, "On the usage, co-usage and migration of ci/cd tools: A qualitative analysis," *Empirical Softw. Engg.*, vol. 28, no. 2, Mar. 2023. doi: 10.1007/s10664-022-10285-5
- [20] A. Decan, T. Mens, and H. Onori Delicheck, "On the out-datedness of workflows in the github actions ecosystem," *Journal of Systems and Software*, vol. 206, p. 111827, 2023. doi: <https://doi.org/10.1016/j.jss.2023.111827>
- [21] Y. Kubo, F. Kanei, M. Akiyama, T. Wakai, and T. Mori, "Action required: A mixed-methods study of security practices in github actions," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, 2026. doi: 10.14722/ndss.2026.240483. ISBN 979-8-9919276-8-0
- [22] A. Khatami, C. Willekens, and A. Zaidman, "Catching smells in the act: A github actions workflow investigation," in *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, 2024. doi: 10.1109/SCAM63643.2024.00015 pp. 47–58.
- [23] N. Saavedra, A. Silva, and M. Monperrus, "Gitbug-actions: Building reproducible bug-fix benchmarks with github actions," in *2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2024. doi: 10.1145/3639478.3640023 pp. 1–5.
- [24] L. Zheng, S. Li, X. Huang, J. Huang, B. Lin, J. Chen, and J. Xuan, "Why do github actions workflows fail? an empirical study," *ACM Trans. Softw. Eng. Methodol.*, Jul. 2025. doi: 10.1145/3749371