

Eliminating Setup Friction in PHP Web Development Labs Using Docker and DevContainers

Elham Eskandarnia Elham

`emsi1350@hotmail.com`

Arab Open University

Research Article

Keywords:

Posted Date: August 18th, 2026

DOI: <https://doi.org/10.21203/rs.3.rs-10356699/v1>

License: © ⓘ This work is licensed under a Creative Commons Attribution 4.0 International License.

[Read Full License](#)

Additional Declarations: No competing interests reported.

Abstract

The first obstacle for students in a web development course is to set up a working PHP development environment. This has nothing to do with the real goal of the course: backend programming and client-server interaction. This paper describes a multi-semester case study with three cohorts and two universities in Bahrain. The case study compares a traditional XAMPP-based laboratory setup with an alternative based on Docker and DevContainers. The traditional setup had six constant friction points: differences in PHP versions, PDO/php.ini misconfigurations, inconsistencies in database credentials, IDE integration barriers, Mac vs. Windows inconsistencies, and a cyclical reinstallation spiral, which in some cases led to students losing a week's worth of completed work. The problems related to the database were especially disruptive; there are twelve laboratory sessions in the course, eight of which require a functioning database connection to be established in week four, so one unresolved configuration error could potentially put two-thirds of the rest of the semester in jeopardy. The move from a traditional setup to a containerized environment decreased setup time from more than two hours to less than ten minutes, representing a reduction of over ninety percent. This decrease persisted regardless of whether the container environment was added at the beginning of the semester or in the middle of it as a corrective action. The container-based cohort of students carried out the laboratory activities in a timely manner and some of them self-reported that they did not need help anymore with their environment. This paper argues, based on cognitive load theory, that the advantages of containerization in education are better reflected through both the time efficiency and the lowering of extraneous cognitive load, which may threaten the entire semester on account of a single point of failure. Limitations and directions for future research are also discussed such as the need for a validated instrument to assess cognitive load.

Introduction

Setting up the development environment is one of the first things students face in a web development course, and it has nothing to do with what they are actually supposed to learn. Before writing a single line of PHP, a working stack must be installed: a web server, PHP, a database, and the drivers that connect them. In our lab this was done with XAMPP for many years. XAMPP is sold as a simple all-in-one installer, but this was not what we observed across several semesters of teaching it.

The most recurring problem was the PDO driver. PDO is not enabled by default, regardless of whether the database is MySQL or SQL Server, and students must manually edit php.ini to uncomment the correct line. It sounds simple. In practice, many students get it wrong: the wrong line gets uncommented, the file is saved incorrectly, or the path does not match their own PHP installation. What follows is rarely a fix. Students delete the entire XAMPP installation and reinstall it instead. By this stage the student usually already has an htdocs folder containing weeks of their own work, so reinstalling means either remembering to back up htdocs first or losing what was built over the previous four weeks. On its own, this pattern could consume two hours or more of a single lab session, for what is, in principle, a one-line fix.

The PHP version is not consistent across students either. XAMPP bundles a specific release, and students often end up on different versions depending on when they followed the installation tutorial. Database connectivity is a second bottleneck: XAMPP's MySQL service ships with no root password by default, while tools such as PhpStorm require one before a connection can be made, so the student has to edit the configuration again to make both sides match. Registering the PHP interpreter inside PhpStorm or VS Code is its own struggle, since it assumes knowledge of local file paths that most beginners simply do not have. Then there is Windows versus Mac. Instructions written for one platform frequently do not work on the other, often with no clear error message telling the student why.

None of this is related to what the student is actually supposed to learn, which is backend programming and how the client and server communicate. Yet it eats a large part of class time, and, more importantly, the student's mental capacity, before the lesson even begins. Cognitive load theory (Sweller, 1994) explains why this matters: working memory is limited, and effort spent on anything unrelated to the learning goal counts as extraneous load, which eats into the capacity left for germane load, the effort that actually builds understanding. Environment setup fits this definition exactly. It is frustrating, effortful, and has nothing to do with the course material itself.

Docker, together with the DevContainer specification supported by VS Code, removes almost all of this extraneous load. The student opens a folder, and everything, correct PHP version, PDO already enabled, database password already set, IDE already connected – is built automatically and in the same way, regardless of whether the machine is running Windows or Mac.

This paper presents a multi-semester case study of replacing an XAMPP-based lab setup with a Docker and DevContainer environment in a web development course that teaches PHP, Apache, and database integration. The contribution of this paper has three parts.

1. It documents six recurring failure points observed in the traditional lab setup across more than three semesters: PHP version mismatch, PDO/php.ini misconfiguration, database password mismatch, IDE integration problems, Mac vs. Windows inconsistency, and the reinstall cycle that follows from these.
2. It reports an instructor-observed drop in setup time, from more than two hours under the traditional approach to under ten minutes with the container-based approach, and this holds across multiple cohorts of the same course.
3. It frames this reduction through cognitive load theory, arguing that the benefit of containerization in the classroom is not really the time saved, but the cognitive capacity given back for actual learning.

The rest of the paper is organized as follows. Section 2 reviews related work on containerization in computer science education and cognitive load theory in programming instruction. Section 3 describes the six friction points in detail. Section 4 presents the Docker and DevContainer solution and maps each part of it to the problem it solves. Section 5 describes the methodology and teaching context. Section 6 reports the results. Section 7 discusses the findings in light of cognitive load theory, and Section 8 concludes with limitations and future work.

Literature Review

This review moves in two steps: first, where Docker and DevContainer setups have already been tried in computer science classrooms, and what gaps remain; then, cognitive load theory itself, the lens used later in the paper to explain why the real payoff of containers is not just the time saved.

Containerization in Computer Science Education

Before Docker, the usual way to give every student or developer the same environment was a virtual machine (VM). A VM runs a full guest operating system on top of the host. That solves the consistency problem, but at a real cost: VMs are heavy to download, slow to start, and need far more disk and memory than the actual application requires (Potdar et al., 2020). Student laptops already have limited resources. Distributing a VM image for every lab, in a classroom setting, is not very different from the original problem — it just shifts from "installing software" to "downloading and running a multi-gigabyte image."

Docker takes a different approach: instead of a full guest operating system, containers share the host machine's kernel and only package the application and its dependencies (Mouat, 2015; Nickoloff & Kuenzli, 2019). This makes containers much lighter and faster to start than VMs, while still giving the same guarantee of consistency across machines. This is the reason Docker, rather than VMs, was the natural choice for the environment described later in this paper.

The idea of using containers to solve the "works on my machine" problem is not new in industry, and Docker in particular has been widely adopted for this reason (Mouat, 2015; Nickoloff & Kuenzli, 2019). In research and education contexts, containers have also been used mainly to support reproducibility: Boettiger (Boettiger, 2015) describes how Docker can preserve not just code but also the exact library and system dependencies needed to replicate a result, and a similar argument is made for reproducible data science workflows more generally (Nüst et al., 2020). Cito et al. (Cito et al., 2017) apply this idea specifically to software and web engineering research, showing that containers can remove a large part of environment-related noise when an experiment is repeated by a different researcher on a different machine.

Most of this work, though, is about research reproducibility, not the daily experience of a student setting up an environment for the first time. The closest match to the present paper comes from computer science education itself. Valstar et al. (Valstar et al., 2020) used DevContainers to standardize the development environment for students in an advanced data structures course, and found the method worked across operating systems without the setup problems that a remote server-based approach typically brings. Liu et al. (Liu et al., 2022) standardized student programming environments at scale using Docker containers, provisioning one container per student automatically through GitHub Codespaces.

Both of these studies confirm that containerization solves the technical side of the setup problem, but neither connects this benefit explicitly to learning theory; the improvement is described mostly in terms of consistency and reduced support burden for the teaching staff, not in terms of what happens to the student's cognitive resources once the friction is removed. This gap is the motivation for looking at cognitive load theory next.

Cognitive Load Theory

Cognitive load theory, originally proposed by Sweller (Sweller, 1994), is built on the idea that human working memory is limited, and instructional design should take this limit into account. The theory normally distinguishes three types of load: intrinsic load, which comes from the difficulty of the material itself; extraneous load, which comes from how the material or task is presented and has nothing to do with the learning goal; and germane load, which is the effort the learner spends actually building schema and understanding. The general principle is that extraneous load should be reduced as much as possible, since working memory that is occupied by it is not available for germane load.

Cognitive Load in Programming Education and the Setup Gap

Programming is already recognised as a domain with high intrinsic load (Garner, 2002). A large body of work applies cognitive load theory specifically to teaching introductory programming, looking at how worked examples, visualization, or scaffolding tools can reduce the extraneous load that comes from syntax and notation (Berssanette & Francisco, 2022). Duran et al. (Duran et al., 2022) cover this literature broadly in a recent review, and show that cognitive load theory has become one of the most cited theoretical frameworks in computing education research.

One thing gets little attention in this literature: the extraneous load that comes before any programming even starts, introduced by environment setup itself. Almost all of the cognitive load work in programming education assumes the student already has a working environment, and looks only at the load introduced once they begin writing or reading code. The six friction points in Section 3 of this paper—PHP version mismatch, PDO configuration, password mismatch, IDE integration, platform inconsistency, and the resulting reinstall cycle—fit the definition of extraneous load almost exactly. They have nothing to do with the intrinsic difficulty of backend programming, yet they occupy the same limited working memory before the lesson can even begin. Removing this pre-lesson load, using container technology already shown to work in a classroom (Liu et al., 2022; Valstar et al., 2020), is therefore not a mere convenience. It is a measure that supports learning directly, in the sense cognitive load theory actually defines it.

The Problem: Traditional Lab Setup Friction

This section describes six recurring friction points observed across more than three semesters of teaching this course using the traditional XAMPP-based lab setup. These are not isolated incidents reported once or twice; they are patterns that repeat, semester after semester, regardless of the specific cohort.

PHP Version Mismatch

The version of PHP bundled with XAMPP changes over time, and since students often follow a tutorial or installation guide found online rather than the one provided by the instructor, different students in the same cohort frequently end up running different PHP versions. This causes code that works for one student to fail for another, for reasons unrelated to the code itself.

PDO Driver Configuration

This was the single most disruptive problem. It was made worse by the fact that many students ended up with more than one PHP installation on the same machine: the one bundled inside XAMPP, and, at the university where PhpStorm's educational license activated correctly, a separate standalone PHP installation used together with PhpStorm's built-in web server. Many students preferred that second option once they found it worked more reliably than configuring Apache through XAMPP. At the other university, where the PhpStorm license did not activate (see Section 3.4), this alternative was not available. Students had no choice but to debug XAMPP and Apache directly, with no working fallback.

Each PHP installation has its own `php.ini` file, and PDO is not enabled by default in either of them, regardless of whether the course uses MySQL or SQL Server. The student must manually edit `php.ini` to uncomment the correct extension line, but where two installations were present, it became common for students to edit the wrong file: the change is made correctly, but to the `php.ini` that is not actually being read by whichever server, XAMPP's Apache or PhpStorm's built-in server, they are currently running. The result looks identical from the student's side; the PDO error persists, even though they are confident the configuration was already fixed.

This confusion ended the same way regardless of which installation was at fault. Rather than identifying which `php.ini` was actually in use, students treat the environment as broken beyond repair and delete and reinstall XAMPP, even when XAMPP itself was not the source of the problem. By this stage the student usually already has an `htdocs` folder containing weeks of their own work, so reinstalling means either remembering to back up `htdocs` first or losing what was built over the previous weeks. The cost could run to two hours or more of a lab session. In several cases, the actual issue, identifying the correct `php.ini`, was never found by the student at all.

Database Credential Mismatch

XAMPP's MySQL service ships, by default, with no root password, while tools such as PhpStorm require a password to be set before a connection can be established. Students had to edit the MySQL configuration manually to introduce a password and then make sure the same password was entered correctly on the IDE side. A mismatch here produces a connection error with no obvious explanation to a beginner.

IDE Integration Bottleneck

Registering the PHP interpreter inside an IDE such as PhpStorm, NetBeans, or VS Code is a separate step, and it assumes the student knows the local file path of their own PHP installation. Beginners rarely do. Many do not even know where XAMPP was installed by the setup wizard, and locating that path becomes, on its own, a recurring source of support requests. One thing made this worse. PhpStorm's free educational license requires activation with a student email, and it did not recognize the email domain of one of the two universities involved. Some students could not activate the license they expected to use, and had to switch to NetBeans or VS Code mid-course.

Cross-Platform Inconsistency (Mac vs. Windows)

Instructions written for one operating system frequently did not transfer to the other. Default install paths, the way services start and stop, even where `php.ini` sits, all differ between Windows and Mac. One set of written instructions could not reliably cover both. Troubleshooting often meant the instructor had to ask which operating system the student was using before any advice could be given at all.

The Reinstall Cycle

The previous five points added up to a recurring pattern. Rather than diagnosing a specific misconfiguration, students treat a broken environment as something to discard and rebuild from scratch. That is understandable. None of the underlying problems were visible or well explained by the error messages XAMPP produces. But it meant the same category of failure resurfaced repeatedly across a semester, each time costing a similar amount of lab time.

Taken together, these six points describe a setup process that is fragile, inconsistent across students, and disconnected from the actual subject matter of the course. They motivate the container-based solution presented in Section 4.

Proposed Solution: Container-Based Environment

We describe the environment first: what replaces XAMPP, and how each component was configured. Then we map it back, point by point, to the six friction points from Section 3, so it is clear which design choice solves which problem.

Environment Setup with Docker and DevContainers

The proposed solution replaces the manual XAMPP-based setup described in Section 3 with a declarative environment built using Docker and DevContainers. Instead of installing PHP, Apache, and a database server directly on the host machine, the entire stack is defined once, inside a Dockerfile, and built into a container image that runs identically regardless of the host operating system.

A custom Dockerfile extends the official php:8.2-apache base image. PDO and the other database extensions (pdo_mysql, mysqli, or the appropriate driver for SQL Server) are installed and enabled inside the image itself, using the PHP extension installer utility, so the student never has to open or edit php.ini manually. Apache modules such as rewrite and headers are also enabled at build time. A short excerpt is shown below:

```
FROM php:8.2-apache
```

```
RUN install-php-extensions pdo_mysql mysqli intl mbstring zip
```

```
RUN a2enmod rewrite headers
```

EXPOSE 80

A devcontainer.json file complements the Dockerfile, and lets Visual Studio Code recognize and build this environment automatically when a student opens the project folder. It sets out which ports get forwarded (port 80 for the web server, port 3306 for the database) and which extensions install inside the container, such as PHP IntelliSense and a PHP debugger. No Docker knowledge is required at this point. VS Code sees the .devcontainer folder on its own and offers to reopen the project inside the container with a single click.

Database services are specified in a standalone docker-compose.yml file orchestrating the PHP/Apache application, MySQL or SQL Server container, and, when applicable, a database management interface such as phpMyAdmin. This file already has the root password, database name, and user credentials set up so that the student does not have to set up or remember any passwords. The same credentials are already set up on the database side and, where applicable, in the application's connection code.

Mapping Solutions to Friction Points

Table 1

summarises how each component of the proposed environment addresses one of the six friction points described in Section 3. Mapping of container-based solution components to the friction points of Section 3

Friction point	Container-based resolution
PHP version mismatch	PHP version pinned in Dockerfile FROM line; identical for every student
PDO/php.ini misconfiguration	Driver installed and enabled at image build time; no manual editing required
Database credential mismatch	Username and password pre-set in docker-compose.yml; same on both sides
IDE integration bottleneck	devcontainer.json configures VS Code automatically; no manual path entry
Mac vs. Windows inconsistency	Container runs identically on any host with Docker installed
Reinstall cycle	Environment is disposable and rebuilt in minutes, with student work outside the container, on the mounted volume, never lost

The last row of Table 1 is worth emphasising. Because the `www` folder, where the student's actual PHP files live, is mounted as a volume into the container rather than copied inside it, the student's work survives even if the container itself is deleted and rebuilt. This directly removes the consequence described in Section 3.2, where deleting and reinstalling XAMPP could mean losing a week of work stored inside `htdocs`; with the container approach, rebuilding the environment and losing the work are no longer the same event.

Methodology

Two things need setting out before the results: the three teaching contexts where observation were collected, and how the time and friction data reported in Section 6 were actually gathered.

The case study draws on three cohort across two academic year and two university, all covering the same core technical content—PHP, Apache, and a relational database—but differing in which environment was used and when.

The first cohort was the Client-Server Programming course at a British university, offered at Level 5 to a mixed group of approximately 60 cybersecurity and Software Engineering student. This was the first time the course was taught, and the lab setup followed what the existing curriculum specify: student either installed the software stack on their own machine or use the university computer lab. This arrangement caused severe disruption, particularly once student began to prefer working on their own laptop rather than the lab machine, a preference the existing curriculum had not accounted for. The lab session in week 1, dedicated to initial environment installation, and the lab session in week 4, when a database had

to be added to the existing setup, were both effectively lost to troubleshooting rather than to the intended lab content.

The second cohort was the same Client-Server Programming course at the same British university, taught the following year to a new cohort of approximately 80 Cybersecurity and Software Engineering students. Based on the experience of the previous year, the Docker and DevContainer environment described in Section 4 was used from week 1 onward, without any traditional setup phase.

The third cohort consisted of about 80 students at a comparable level enrolled in a web development course at a different American university (Year 3). This cohort used a conventional, locally installed configuration at the start of the semester. By week five, the number of setup-related issues—which were exacerbated in this cohort by the PhpStorm license issue detailed in Section 3.4 and the subsequent need to deploy to an AWS server provided by the university—had reached a level deemed unsustainable for the duration of the course. Midway through the semester, the Docker and DevContainer environment was brought in as a remedial step, starting week 6. This cohort gives a direct within-semester comparison between the traditional and container-based approach, using the same student, the same instructor, and the same course material. The improvement beyond that point was significant, and it was immediate.

Time estimate report in Section 6 are based on instructor observation across these three cohort, not on a formally instrumented or controlled experiment. No timer was used to measure individual students; instead, the estimate reflect what was consistently observed during scheduled lab sessions regarding how long it typically took a student to reach a working environment under each approach. Informal feedback volunteered by students during and after the affected lab session is also reported, where relevant, in Section 6.

Students in all three cohorts used a variety of integrated development environments: some used NetBeans for PHP, some used PhpStorm, and some, especially in the Docker-based cohort, used Visual Studio Code, which integrates directly with the DevContainer specification. Since PhpStorm’s built-in PHP server allowed students to run their code without configuring Apache or XAMPP at all, it was actually the preferred option for many students. This is also why, as described in Section 3.2, a separate standalone PHP installation alongside the one bundled in XAMPP was so common. However, some students were forced to switch to NetBeans or VS Code regardless of their preference because PhpStorm’s free educational license required activation through a student email, and this activation did not work reliably for one of the two universities involved. Because of this diversity, the IDE integration bottleneck mentioned in Section 3.4 was, if anything, seen across a greater variety of tools than what would have been revealed by a single IDE course.

Results

A consistent pattern holds across all three cohorts, regardless of university, program, or course title. When the Docker and DevContainer environment was used, whether from week one, as in the second

British cohort, or introduced mid-semester, as in the American cohort, setup-related disruption to lab activities essentially disappeared. Where the traditional setup ran for the whole semester, as in the first British cohort, the opposite was true. Setup dominated.

In the first British cohort, both week 1 and week 4 lab sessions were effectively lost to troubleshooting instead of the intended content. The week 4 loss mattered more. Weeks 1 to 3 cover environment setup, input/output logic, and session handling, none of which need a working database connection, as shown in Fig. 1. Everything after that does. Database setup and form handling (weeks 4–5), four weeks of CRUD operations (weeks 6–9), an API exercise built on stored data (week 11), and the final project submission (week 12) all depend directly on the connection established in week 4. Eight of the twelve lab sessions, two-thirds of the semester, are technically un-completable without it, whatever the state of the student's PHP logic. So when the connection failed, for the PDO, port, or password reasons described in Section 3.2 and Section 3.3, the damage was not contained to week 4 – it propagated through the rest of the semester, since every later session and the final project inherited the same unresolved problem.

Semester structure showing which lab sessions depend on the database connection established in week 4

Across these sessions, the instructor's observation was consistent: reaching a working environment under the traditional approach took two hours or more, when it was reached at all within the scheduled lab time. Under the container-based approach used in the other two cohorts, the same outcome was reached in under ten minutes every time – a reduction of more than ninety percent in setup time, and correspondingly, ninety percent more of the lab session left for the actual task rather than configuration. Figure 2 shows this comparison.

Approximate time to reach a working development environment, traditional setup (XAMPP) versus container-based setup (Docker)

In the American cohort, the same pattern held for the first five weeks, compounded by the PhpStorm licensing issue described in Section 3.4. Then, from week 6 onward, once Docker and DevContainers came in, it stopped. Students completed lab activities on schedule, and more than once a student volunteered, unprompted, that they no longer needed help getting their environment running. The second British cohort showed the same thing from week one: lab activity proceeded on schedule, and the instructor stopped fielding the configuration questions that had dominated the first cohort's early weeks.

These are instructor observations, not data from a formal instrument, and should be read that way. Still, the pattern held across two universities in Bahrain and three different cohorts, and the American cohort's within-semester before-and-after comparison strengthens it further. Taken together, this supports the container-based environment, rather than some other change in circumstance, as the actual cause of the difference observed.

Discussion

The results from Section 6 are interpreted here through the cognitive load lens introduced in Section 2.2, then placed against the closest related work, Valstar et al. (Valstar et al., 2020) and Liu et al. (Liu et al., 2022), to show what this case study adds beyond what they already show.

Setup Friction as Extraneous Load

The results presented in Section 6 can be read in two ways. The first, more obvious reading is that a container-based environment saves time: more than ninety percent of the time previously spent reaching a working setup is recovered, and this time can instead be spent on the lab activity itself. This reading is correct, but it is not, by itself, the most important one for an educational audience.

The second reading is the one this paper wants to emphasize: what happens to the student's cognitive resources during the two hours, or more, that a failed setup typically consumes. That time is not simply "lost." According to cognitive load theory (Sweller, 1994), it is actively occupied by extraneous load: the effort of guessing which `php.ini` is the right one, of remembering whether the MySQL password was already changed, of figuring out why PhpStorm will not connect when, as far as the student can tell, everything was configured correctly. None of this builds any understanding of backend programming or client-server interaction, the actual learning goal of the course. It gets worse. The failure points in Section 3 often produce identical symptoms from different causes, as the dual PHP installation problem in Section 6 shows, so the student cannot even reliably learn from debugging it. The same error message could mean three different things, and a beginner has no way to know which.

Eight of twelve lab sessions depend on the database connection, as Fig. 1 illustrates, and this makes the problem specific to this course's structure rather than to environment setup problems in general. A student who fails to resolve the connection in week 4 does not simply fall behind on database content. The week 5 form, the week 6 to 9 CRUD operations, the week 11 API exercise, and the final project all become unreachable, since each one requires the same broken connection to function. This is not extraneous load confined to one session. It compounds. The student re-attempts the same unresolved configuration problem across multiple weeks, each time at the expense of the germane load that should be going toward that week's new material instead.

Viewed this way, the contribution of a container-based environment to education is not best described as convenience. It is better described as the removal of a recurring, compounding source of extraneous load that, in this course's particular structure, has the potential to compromise most of the remaining semester from a single unresolved configuration error.

Relation to Prior Work

Valstar et al. (Valstar et al., 2020) and Liu et al. (Liu et al., 2022) both show that DevContainer and Docker-based environments can be deployed successfully in a real computer science course, and both report that the approach removes the cross-platform setup problem that a more traditional approach, or even a remote server-based one, would still have. The present case study confirms this again, across two

universities in Bahrain and three cohorts. That is not an isolated result tied to one institution or one course.

This paper adds two things beyond confirmation. First, neither Valstar et al. nor Liu et al. offers a within-semester comparison. Their evaluations describe a single, pre-existing Docker-based course, not a course that switched from a traditional format to a container-based one partway through, as the American cohort did here. That intra-semester switch is arguably more convincing than a comparison across different courses, since the students, the instructor, and the course material all stay fixed while only the environment changes.

Second, and more importantly, neither prior study connects these benefits to cognitive load theory or to any other learning theory. Their contribution is framed around consistency, onboarding time, and less support work for teaching staff – real and useful points, but they stop short of explaining why that consistency matters pedagogically, beyond convenience. This paper argues, building on the gap identified in Section 2.3, that the cognitive load literature in programming education has largely ignored pre-lesson setup load, and that the real value of removing this friction is the cognitive capacity it gives back to the student, not just the time saved. That is why the cascading effect from Section 6 matters. A single unresolved configuration issue in week 4 can jeopardize eight of twelve lab sessions. This is not a logistical inconvenience. It is a direct threat to the germane load the course is meant to build, week by week.

Conclusion and Future Work

This paper presents a multi-semester case study of replacing a traditional, XAMPP-based lab environment with a Docker and DevContainer-based alternative, across three cohorts and two universities in Bahrain. Six recurring friction points showed up in the traditional setup: PHP version mismatch, PDO/php.ini misconfiguration, database credential mismatch, IDE integration bottleneck, cross-platform inconsistency between Mac and Windows, and a recurring reinstall cycle that, in several cases, cost students a week of their own work. These points were not independent of each other. The database-related problems, once they hit in week 4, threatened eight of the remaining twelve lab sessions, since most of the course's later content depends on that connection.

Switching to a container-based alternative cut setup time from over two hours to under ten minutes, a reduction of more than ninety percent. This held whether the container environment was in place from week one or introduced mid-semester as a fix. Students in the container-based cohorts finished lab activities on schedule, and more than once said, unprompted, that they no longer needed help with their environment.

Using cognitive load theory (Sweller, 1994), this study argues that the benefit of containerization here goes beyond time saved. The two or more hours a poor setup often demanded were not neutral lost time. That effort went into something entirely unrelated to backend programming or client-server

interaction, the actual learning goals of the course. Removing that burden frees cognitive capacity for the material itself, which should translate into a deeper understanding of the course content.

This study has real limitations, and they should shape how the results are read. The observations come from instructor experience across three cohorts, not a controlled experiment. No individual student-level time data was collected, and the time estimates, while consistent across semesters, stayed approximate. Student feedback was volunteered rather than gathered through a structured survey or interview. Future work should address this with a proper instrument – a short post-lab questionnaire measuring perceived cognitive load directly (Duran et al., 2022) – alongside the kind of time-based observation reported here.

This case study could grow in a few directions. A systematic comparison of perceived cognitive strain, using a validated instrument, between traditional and container-based cohorts would strengthen the claim made in Section 7. Deployment is one open thread. The AWS server used by the American cohort, mentioned briefly in Section 5, falls outside the scope of this paper, but it is a natural extension, since deployment brings its own environmental friction that container technology can address just as well. And since DevContainers integrate directly with cloud-based development environments such as GitHub Codespaces, as Liu et al. (Liu et al., 2022) show, a future version of this course could drop the dependence on student hardware entirely. That would matter most at institutions where student devices vary widely in specification or operating system.

Declarations

Competing Interests:

- The authors declare no competing interests.

Ethics Approval and Consent to Participate:

- Formal review by an institutional ethics committee or IRB was waived by the Scientific Research Council at Arab Open University, the body responsible for research ethics oversight at the institution. The Council granted this waiver because the study drew solely on retrospective, anonymized, aggregate instructional records, namely lab completion times and recurring technical friction points, with no individually identifiable student data collected or reported, and no experimental intervention applied to any participant beyond normal course delivery. On the same grounds, the Council waived the requirement for informed consent from individual students, since no student was identifiable in the data and no procedure outside standard course activities was carried out. The study was conducted in accordance with the research ethics guidelines of Arab Open University governing the use of routine, anonymized instructional records for retrospective pedagogical research.

Funding:

Not applicable.

Author Contribution

The paper has solo author

Data Availability

The code and configuration files supporting this case study are available at:

<https://github.com/elhamum/php>

References

1. Berssanette, J. H., & Francisco, A. C. de. (2022). Cognitive load theory in the context of teaching and learning computer programming: A systematic literature review. *IEEE Transactions on Education*, 65(3), 440–449. <https://doi.org/10.1109/TE.2021.3127215>
2. Boettiger, C. (2015). An introduction to docker for reproducible research. *ACM SIGOPS Operating Systems Review*, 49(1), 71–79. <https://doi.org/10.1145/2723872.2723882>
3. Cito, J., Ferme, V., & Gall, H. C. (2017). Using docker containers to improve reproducibility in software and web engineering research. *International Conference on Web Engineering*, 609–612. https://doi.org/10.1007/978-3-319-38791-8_58
4. Duran, R., Zavgorodniaia, A., & Sorva, J. (2022). Cognitive load theory in computing education research: A review. *ACM Transactions on Computing Education*, 22(4), 1–27. <https://doi.org/10.1145/3483843>
5. Garner, S. (2002). Reducing the cognitive load on novice programmers. *Proceedings of ED-MEDIA 2002–World Conference on Educational Multimedia, Hypermedia and Telecommunications*, 578–583.
6. Liu, J., Bowden, T., Sun, Y., & Sahami, M. (2022). Standardizing students' programming environments with docker containers. *Proceedings of the 27th ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE 2022)*. <https://doi.org/10.1145/3502717.3532164>
7. Mouat, A. (2015). *Using docker: Developing and deploying software with containers*. O'Reilly Media, Inc.
8. Nickoloff, J., & Kuenzli, S. (2019). *Docker in action*. Simon; Schuster.
9. Nüst, D., Sochat, V., Marwick, B., Eglen, S. J., Head, T., Hirst, T., & Evans, B. D. (2020). Ten simple rules for writing dockerfiles for reproducible data science. *PLOS Computational Biology*, 16(11), e1008316. <https://doi.org/10.1371/journal.pcbi.1008316>

10. Potdar, A. M., Narayan, D., Kengond, S., & Mulla, M. M. (2020). Performance evaluation of docker container and virtual machine. *Procedia Computer Science*, 171, 1419–1428.
<https://doi.org/10.1016/j.procs.2020.04.152>
11. Sweller, J. (1994). Cognitive load theory, learning difficulty, and instructional design. *Learning and Instruction*, 4(4), 295–312. [https://doi.org/10.1016/0959-4752\(94\)90003-5](https://doi.org/10.1016/0959-4752(94)90003-5)
12. Valstar, S., Griswold, W. G., & Porter, L. (2020). Using DevContainers to standardize student development environments: An experience report. *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, 1–4.
<https://doi.org/10.1145/3341525.3387424>

Figures

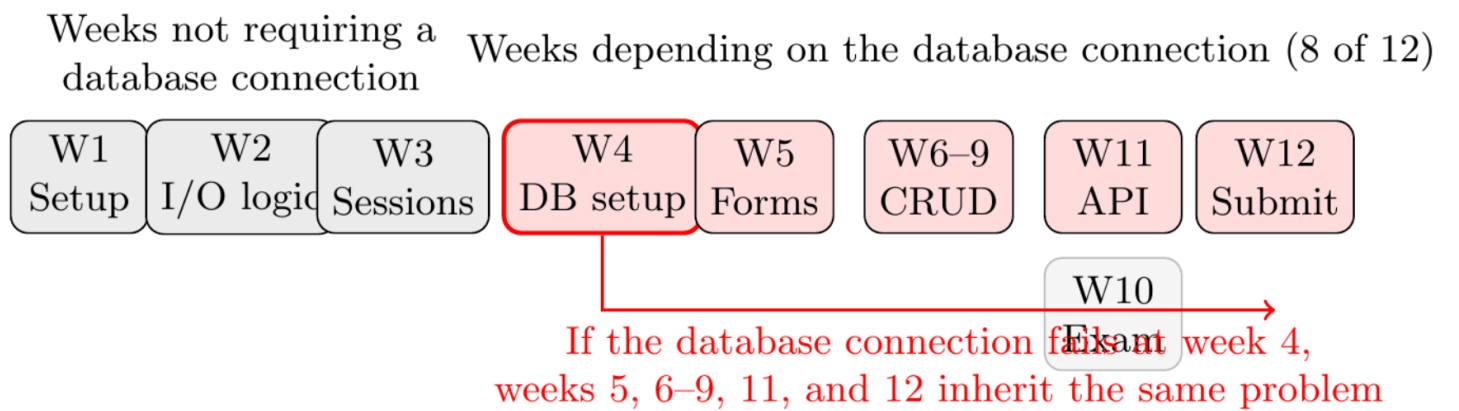


Figure 1

Unnumbered image in the Result section.

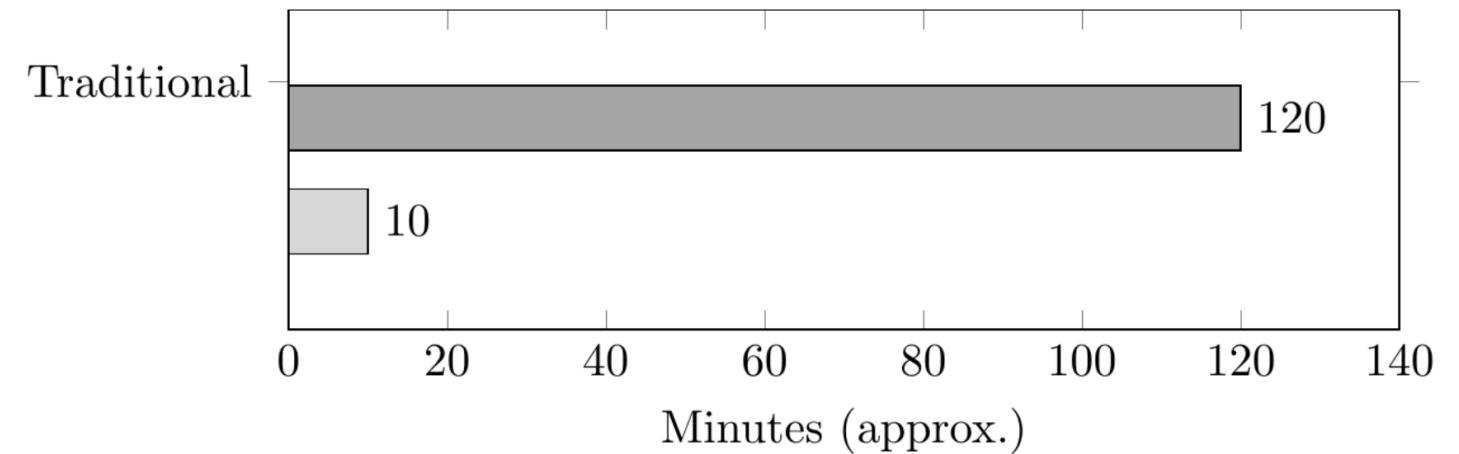


Figure 2

Unnumbered image in the Result section.