Introducing Computer Science Undergraduate Students to DevOps Technologies from Software Engineering Fundamentals

Edgar Sarmiento-Calisaya
esarmientoca@unsa.edu.pe
Universidad Nacional de San Agustín de Arequipa
Perú

Alvaro Mamani-Aliaga
amamaniali@unsa.edu.pe
Universidad Nacional de San Agustín de Arequipa
Perú

Julio Cesar Sampaio do Prado Leite
julioleite@ufba.br
Instituto de Computação
Universidade Federal da Bahia
Brazil

ABSTRACT

The fast adoption of collaborative software development by the industry allied with the demand for a short time to market has led to a dramatic change in IT roles. New practices, tools, and environments are available to support professionals in their day-to-day activities. In this context, the demand for software engineers with these skills continues to increase, specifically those related to Extreme Programming, Agile frameworks, CI/CD, and DevOps. To match Computer Science undergraduate students' skills with existing job offers, some universities have begun to include DevOps topics in their curriculums. However, due to the wide range of courses covered in Computer Science majors, it is particularly challenging to introduce DevOps within the context of Software Engineering fundamentals, i.e., connect abstract concepts to skills needed for software engineers in the industry. This paper investigates ways of introducing Computer Science students to industry-relevant practices and technologies early from two Software Engineering fundamentals courses. Student outcomes were extremely positive, providing insights into ways to introduce students to DevOps-related practices and technologies and bridge the gap between academia and industry.

CCS CONCEPTS

• **Social and professional topics → Computing education**.

KEYWORDS

DevOps, CI/CD, Industry, Academy, Software Engineering, Education, Tools, Environment

1 INTRODUCTION

In the last decades, Information Technology's roles have changed dramatically, the complexity of software systems has reached unpredictable levels as the adoption of practices, tools, and environments for the automation of Software Engineering tasks (e.g. development, testing, and release) has become a standard in the software industry [7]. Driven mainly by large software companies (e.g. Google or Amazon AWS) as the dominance of Agile frameworks and Extreme Programming (XP) [3] practices; DevOps [10] has become a new way of software development, release, and operation. As an extension of agile methods, DevOps enables a fluid collaboration between development, release, and operation teams by automating and integrating repetitive tasks and helping software engineers create customer and business value. Figure 1a shows an example of a software engineer Job offer in a large software company.

According to Henderson [19] and Winters et al. [40] in their book about Software Engineering at Google, "Software Engineering encompasses not just the act of writing code, but all of the tools and processes an organization uses to build and maintain that code over time", i.e., a "Software Engineer" not only writes programs but also applies best practices and uses tools and environments to solve problems and mainly *satisfy needs*.

Despite the importance of Continuous Integration and Continuous Delivery (CI/CD) and DevOps to the software industry [35] (Figure 1b), there is still a significant shortfall of Software Engineers skilled to meet the current market needs [17] [25] [14] [7] [2] [36]. With more and more companies demanding these skills from graduate students, it is critical for students to gain hands-on experience with these new practices and technologies.

To unify the education and workforce sides of computing, the Joint ACM and IEEE Task Force [23] report advocates "*a transition over time to a common language that stakeholders can utilize across education and workforce constituencies to understand and minimize the gap between education outputs (graduates) and the inputs required for a successful contribution to a global workforce in computing*". However, industry-relevant technologies cover technical and non-technical concerns, and DevOps is particularly challenging in education. Due to this fact, recent studies have investigated the current state of DevOps education (challenges and teaching methods), and how DevOps education should proceed in the future (recommendations) [32] [14][12]. On the other hand, agile and DevOps topics are being incorporated into many knowledge areas (KAs) of the newest public beta version (v4) of the Software Engineering Body of Knowledge (SWEBOK) [39]

Because of this, some universities have begun to adapt the content of their study programs to satisfy the market needs and match the skills of Computer Science [6] [34] [18] and Software Engineering [20] [2] [37] students with Software Engineer job offers. In some cases, new and advanced DevOps specialization courses (e.g. DevOps Engineering, Continuous Delivery and DevOps or DevOps Culture and Mindset) were created [31] [21]; while in other cases DevOps topics were included in existing ones (e.g. Cloud Computing). However, due to the wide range of courses covered in Computer Science majors, it is particularly challenging to introduce DevOps within the context of Software Engineering fundamentals courses, i.e., connect abstract concepts to technical and non-technical skills needed for software professionals in the industry.

This paper investigates ways of introducing Computer Science undergraduate students to DevOps practices and technologies early from two Software Engineering fundamentals courses – Software

Software Engineer Position

Basic Qualifications
- 1+ years of experience contributing to the system design or architecture (ARCHITECTURE, DESIGN PATTERNS, RELIABILITY and SCALING) of new and current systems.
- 2+ years of non-internship professional software development experience
- Programming experience with at least one software programming language.

Preferred Qualifications
- Knowledge of professional practices for the full software development life cycle, including CODING STANDARDS, CODE REVIEWS, SOURCE CONTROL MANAGEMENT, BUILD PROCESSES, TESTING, and OPERATIONS.
- Understanding of CI/CD and AGILE software engineering PRACTICES
- Background in support for large scale application implementations.
- Experience with distributed computing and enterprise-wide systems
- Excellent written and verbal communication, analytical and COLLABORATIVE problem-solving skills

a)

b)

ACM/IEEE Computing Curricula: Software Engineering Knowledge Area

REQUIREMENTS ENGINEERING: functional and non-functional requirements; requirements properties, elicitation, specification, validation and tracing.
SOFTWARE DESIGN: Design principles and paradigms; Structural and Behavioral models; Architecture styles and patterns; Relationships between requirements, design and code.
SOFTWARE CONSTRUCTION: Coding standars and practices; Integration strategies; Development context.
TOOLS AND ENVIRONMENT: Software configuration management and version control; Release management; Requirements analysis and design; Testing; Automatic build and Continuous integration; Tool integration.
VERIFICATION AND VALIDATION: Concepts; Inspections, reviews and audits; Testing Types; Testing Fundamentals; Defect Tracking.
SOFTWARE EVOLUTION: Pre-existing code (legacy); Evolution; Reengineering (e.g. refactoring); Reuse.
SOFTWARE RELIABILITY: Software reliability; System reliability and failure; Fault lifecycle; Software fault tolerance techniques and models.
SOFTWARE PROCESSES: Process models.
SOFTWARE PROJECT MANAGEMENT: Team; Effort stimation and Risk.
FORMAL METHODS: Formal specification and analysis techniques and languages.

c)

Figure 1: Example of a software engineer job offer in a large software company (a), skills and knowledge considered necessary to work under agile frameworks in the industry [7] (b), and the theoretical foundations of software engineering according to ACM/IEEE computing curricula [23] (c).

Engineering Fundamentals (Software Engineering I) and Contemporary Software Development (Software Engineering II) — as per the recommendations of the ACM/IEEE Computing Curricula [23] and the newest beta version of SWEBOK [39]. For this purpose, we describe the way the courses were organized and the practices and technologies that were used in hands-on assignments requiring teams to plan, design, develop, test, evolve, and release a software product in an iterative way during a two-semester software project. While DevOps is a set of ideas, practices, processes, and technologies – *culture* that allow development and operations teams to work together; CI/CD puts these DevOps ideals into practice. Therefore, the updated courses focused on CI/CD,

To evaluate our approach, we updated the *competencies* (knowledge + skills) that apply to these courses. The *student's outcomes* during a two-semester software project were extremely positive; providing insights into ways of introducing students to industry-relevant practices and technologies they have to apply or use if they want to work or set up their own software company, encouraging students collaboration and bridge the gap between academia and industry.

2 CHALLENGES AND VISION

We believe that both Software Engineers and DevOps professionals are aimed at developing and releasing software; however, a DevOps professional is more focused on automating development and release processes. Thus, a DevOps Engineer begins his/her career as a Software Engineer while a Software Engineer usually automates repetitive tasks.

When incorporating industry-relevant practices and technologies into Software Engineering fundamentals courses, we identified the following challenges:

- **Theory and Practice**: Most software engineering courses are still focused on theoretical foundations and abstract concepts [17] (as in Figure 1c), even though one of the main objectives of computer science majors is to train students to enter the software development companies as their first job. As such, teaching theoretical foundations in combination with industry practices and technologies is a must.
- **Teaching Approach**: In line with *Sommerville's* notion of introductory software engineering, we believe that students should apply the abstract concepts, practices, and technologies in software projects relevant to them, e.g. a *project-based* approach where students (more than two) can collaborate

when planning, designing, developing, testing, evolving, or delivering a more relatable software product.

- **Technology Availability**: Technology currently in the market is fragmented for development, testing, deployment, platform configuration, and people. The tools and environments used to cover software engineering fundamentals depend on the ability and industry experience of the instructors to integrate them all. For instance, *Jenkins*, *GitHub Action*s or *Amazon CodePipeline* could be used to set up a CI/CD environment.
- **Wide DevOps Skills**: Existing technology is based on different programming languages and executing platforms. As such, using them in combination requires instructors to have significant technical and non-technical skills. For instance, *SonarLint*, *SonarQube* or *SonarCloud* could be used to automate code reviews and deliver clean code.
- **Customization**: Computer scientists focus on solving problems, coding, programming languages, algorithms, computational theory, and models. However, a computer scientist may also focus on extending, refactoring, or redesigning current tools to adapt them to team needs and culture. As such, instructors should choose technologies that can be easily integrated into CI/CD pipelines, extended through *plugins*, or set up in distributed environments, preferably, free and open source.
- **Configuration**: DevOps tools are complex, numerous, and hard to set up, making it difficult to provide an appropriate learning environment. As such, instructors should focus on concepts and choose proper tools only as an illustration.
- **Human Aspects**: A *project-based* teaching approach provides a social context that encourages teamwork among students. As such, instructors must choose proper practices and technologies to facilitate communication, coordination, and collaboration between team members.

As a consequence, we posit that Software Engineering courses should provide concepts for software development, testing, and release through a project-based teaching strategy, which must be focused on relevant software products for students, and serve as the basis for the development of the laboratories. To create (simulate) awareness of the expectations of the industry, students must be challenged by real-life software engineering problems, and apply the right practices and proper tools to solve them collaboratively. That is, students must plan, design, develop, test, evolve, and release a software product in an iterative way and supported by a CI/CD pipeline.

3 COURSE OVERVIEW

Today, job positions and software development in the industry are characterized by keywords [7] such as Agile, Kanban [1], Scrum [38], Test-driven Development (TDD), Behavior-driven Development (BDD), Coding Standards&Conventions, Clean Code [29], SOLID Principles&Clean Architecture [28], Issue Tracking, Domain-driven Design (DDD) [11], Refactoring [16], Version Control, Change Control, Microservices [26] [30], Design Patterns, CI/CD, DevOps, Tool-support among many others. Figure 1a shows an example of a software engineer Job offer in a large software company.

Even though software development and release have evolved significantly in the last decade, neither the completely revised and updated version (3.0) of the SWEBOK [5] nor the Computer Science Curriculum of the Joint ACM and IEEE Task Force [23] incorporated topics related to DevOps in the software engineering knowledge areas. Thus, there are insufficient studies [17] [12] investigating ways of weaving DevOps into software engineering courses or incorporating industry-relevant practices and technologies together with the theoretical foundations of software engineering (Figure 1c). Thus, XP, CI/CD, or DevOps-related practices and technologies still receive minor attention in undergraduate computer science education [17].

In the following subsections, we will present our strategy towards implementing the requirement expressed by the last paragraph of Section 2, by detailing how the courses would be structured (Content, Organization, and Grading).

3.1 Content

In the context of an undergraduate program of *Computer Science*[1] in the Latin American region, we have updated two Software Engineering courses by incorporating industry-relevant practices and technologies together with the theoretical foundations. Students take these courses in the fifth and sixth semesters of their junior projects, allowing them to apply the acquired skills to senior projects (eighth to tenth semesters). Previously, students attended during their first and second-year courses about *Programming* (C/C++ and OO programming), *Computing Fundamentals* (Operating Systems, Linux and Command Line Interfaces), *Database Management* (Relational Databases, SQL and NoSQL) and *Platform-based Development* (Java/Python; Web, Services and Mobile Applications).

A *course* is divided into 3 *learning units* (as in Figure 2); and a learning unit is composed of a set of *competencies* (knowledge + skills + dispositions) [15], *topics* and industry practices and technologies suggestions to be used in laboratory assignments. To test the acquired competencies, there are partial evaluations and continuous ones at the end of each unit. During continuous evaluations, students gain hands-on experience with industry practices and technologies. We describe the two updated courses in the following:

- **Software Engineering Fundamentals (Software Engineering I)**: This course introduces the fundamentals of software engineering. That is, the processes, activities, tasks, and techniques related to Requirements Engineering, Software Design&Architecture, and Software Construction.
- **Contemporary Software Development (Software Engineering II)**: This course extends the ideas of software design and construction from the introduction of software engineering tools and environments, quality and testing techniques, and the challenges in large-scale legacy systems. That is a broader vision of Software Engineering from the point of view of projects.

These courses are mandatory for the Computer Science students in our university. The fourth year in Computer Science proposes another mandatory course named *Cloud Computing*, which introduces *infrastructure as code*, *containerization*, *service orchestration*

[1] https://fips.unsa.edu.pe/cienciadelacomputacion/

Course	Competency	Area/Unit	Topic	Practice	Tool or Environment
Sofware Engineering Fundamentals	Bound the system from its context	Requireements Engineering	Software Requirements and Software Engineering	UML Use Case Diagram and DDD Ubiquitous Language.	StarUML.
			System and Context		
	Identify and document functional and non-functional requirements.		Requirements Elicitation	DDD Ubiquitous Language and User Story.	
	Describe requirements through scenario or goal based techniques.		Requirements Specificaction	Descriptions or BDD Specifications	[NFR Framework, Cucumber]
	Inspect requirements specifications by provided Checklists.		Requirements Analysis	User Story, Use Case or BDD Checklists.	
	Validate the requirements by creating web or mobile app prototypes.				Justinmind (or Proto.io, Marvel, Figma, Canva)
	Manage requirements by prioritizing, assessing impact of changes, tracing requirements to tasks (work items), tracking them and team communicaton and coordination.		Requirements Management and Documentation	Kanban/Scrum Board	Trello, Miro or Github Project and Slack
	Present the domain of a software system by dividing it up into aggregates, modules, bounded contexts or sub-domains, and using a modeling language.	Software Design	Design Principles	DDD Entities, Vos, Services, Factories, Aggregates, Modules, Bounded Contexts and the UML Class Diagram	StarUML.
			Design Paradigms		
			Structural and Behavioral Models		
	Design the system architecture (subsystems, components or module) by considering architectural patterns and design practices.		Software Architecture: Fundamentals and Documentation	Layers, microservices or event-driven patterns; DDD&Layers; UML Package/Component Diagram.	StarUML
			Architecture Styles and Patterns		
	Generate automatically source code from design models (structural) according to Layers/Restful/Event-driven frameworks		Module, Package, Component, Library and Framework	DDD&Layers; and MVC, Restful and ORM frameworks	StarUML extensions
	Apply consistent coding styles that contribute to readability, maintainability and reusability of the software.	Software Construction	Coding Styles	Monolith, Cookbook, Things, Error/Exception Handling, Trinity and Restful	IDE, SonarLint (IDE extension) and GitHub
	Demonstrate and fix common coding bugs, smells and vulnerabilities by performing code reviews on a chosen language and coding standard.		Coding Standards and Conventions	Code Reviews and Clean Code	
	Apply software development practices and approaches that contribute to adaptability, robustness and reliability of complex software.		Integration Strategies, Doubles & TDD	*To be done in next subject (Integration Test)*	
			Development Practices	SOLID Principles	IDE
			Development Approaches	DDD and Clean Architecture	IDE
Contemporary Software Development	Manage project repositories in a unique platform and using version and change control tools.	Tools and Environments	Software configuration management and version control	Multi-branch repository	Git and GitHub integration
	Perform automatic packaging, publishing, release (deploy) and running of a software system in a central repository and an online environment (host).		Release management	Release and containerization	Nexus Repo., Docker and Kubernetes
			Requirements and Design Tools	Kanban/Scrum Board	Trello or Github Project, StartUML
	Perform automatic static analysis of source code for bugs, code smells and vulnerabilities.		Testing tools	Code Reviews and Source Code Analysis	SonarQube
	Build a project by using automatic build systems.		Automatic Build	System Builders and Dependency Management	Java (Maven, Gradle), Python (PIP, PyBuilder), C/C++/C# (Cmake, NuGet, MSBuild), JS (NPM,Webpack)
	Construct a simple CI/CD Pipeline to automatically integrate and deploy a software		Continuous Integration	CI/CD Pipelines (build, unit testing and quality control)	Jenkins
	Measure the complexity (readability, testability and maintainability) of a program by developing a Control Flow Graph of the code.	Verification and Validation (V&V)	V&V: Inspections, reviews and audits	Cyclomatic Complexity and Cognitive Complexity	SonarLint
			Testing: process, techniques, levels, strategies and plan		
	Design, implement and execute test cases of a software component using techniques of white box to measure code coverage.		Test Cases and Test Cases Generation Methods	Testing Based on Program Code Coverage, xUnit Framework, TDD and Test Doubles	xUnit and Mocking: Java (Junit, Mockito), Python (unittest) C++ (Gtest, GMock), Javascript (Jest)
	Design, implement and execute test cases for a system under test using techniques of black box to measure quality metrics in terms of		Test Automation: xUnit; Test Doubles (Integration Test); Functional, Performance, Security and Usability Tests; Regression Test&TDD	Equivalence Partitioning and Boundary Value Analysis	Selenium Web Driver
	Perform automatic performance (load, volume and stress) testing.			Simulating Workloads and users	Apache JMeter
	Perform automatic security (penetration) testing.			OWASP and Vulnerabilities	OWASP ZAP
	Track defects lifecycle using issue tracking and notification tools, and performing regression		Issue Tracking	Issue lifecycle	Github Issues (or Jira) and Slack
	Refactorize a legacy software by performing static analysis to remove bugs, code smells and vulnerabilities, or add new	Software Evolution	Legacy Systems	Refactoring, regression testing, xUnit and TDD	SonarQube and SonarLint
			Reengineering		
	Analyze (potential security, performance and reliability weaknesses) a given software architecture and migrate from Monolith to Microservices or Event-driven Architectures.		Reengineering Techniques: Refactoring and Redesign	From Monolith to Microservices or Event-driven Architectures using DDD Bounded Contexts or Modules	SonarQube and CISQ Standard (ISO 5055) [and IBM Mono2Micro]
	Analyze (potential reusability weaknesses) a given software and improve its reusability by design patterns or software product lines.		Software Reuse	Design Patterns	SonarQube
			Reuse techniques: Design Patterns and Software Product Lines		

Figure 2: Software engineering courses, competencies, knowledge areas, topics, practices, and tools&environments.

and *cloud infrastructures* and practices. This course complements the previous software development courses by introducing tools, environments, and platforms (e.g. *Chef, Ansible, Docker, Kubernetes, and Amazon AWS*) for automatic and continuous software release (deployment).

3.2 Organization

Following a *project-based approach*, during and at the end of each course, students are required to release a *software project* (web application) in *teams* as a final practical exam, and evidence that the concepts, practices and tools studied were applied during their development or evolution. The software project must be available as a *GitHub* repository (with a README) and a Kanban board in a task tracking and management tool like *Trello*, *Miro* or *GitHub Projects*.

To achieve this goal, a course is organized as a mix of weekly lecture (2hrs) and laboratory (4hrs) sessions developed during 17 weeks. *Lectures* present the concepts required to develop and evolve a software project. The remaining of the course is an interleaving between lecture and laboratory sessions. *Laboratories* introduce the practices and tools that must be applied to the team project; and serve as *project follow-up* (aimed at having a close monitoring of the work done for each group member and helping solve any encountered impediments [4]) and *checkpoint* (where each group presents the advances regarding the projects objectives [4]) sessions. Checkpoints take place at the end of a learning unit, and in the last checkpoint, teams make a demo of the project developed or evolved.

The development of the projects allows students to acquire some relevant competencies (as in Figure 2) for the industry and reinforce theoretical concepts, however, each course seeks specific objectives:

- **Software Engineering Fundamentals**: Students specify functional requirements (FR) and non-functional requirements (NFR); design a domain model and system architecture; construct a software product with a high-quality source code or clean code – *easy to read, understand, maintain, evolve and reuse*; as manage (prioritize, trace and track) their development and assess the consistency (change impact) between the generated artifacts (software requirements, domain model, architecture and source code).
- **Contemporary Software Development**: Students manage (add, update, and merge changes) code repositories, construct a simple CI/CD pipeline (build, unit testing, and quality control), revise the existing pipeline to integrate other relevant steps (functional testing, performance testing, security testing, and automatic release) and track (lifecycle) defects; to automatically integrate and deploy a software system in an online platform.

3.3 Grading

The grading of each course is organized around 3 main milestones and deliveries (according to learning units). While the 3 partial evaluations (2 theoretical exams and 1 development project) count for 50% of the final grade, the other half is composed of continuous evaluations - laboratory assignments (50%).

To satisfy the expectations of the academy (students must have a strong understanding of the fundamentals, including but not limited to an ability to write programs, as well as an ability to embrace practices and tools to satisfy needs) and the industry (students must have a high degree of critical thinking to choose the properly practices and tools and work as a team in a highly automated development environment when resolving problems), we have organized the theoretical exams, final practical exam and laboratories according to the competencies described in Figure 2.

Each *laboratory* is precisely specified, so it lets students know exactly the competency being achieved, what they have to do, and which practices and tools should be used. For instance, the laboratory of "*Domain Modeling*" says that students have to "*Present the domain of a software system by dividing it up into aggregates, modules, bounded contexts or sub-domains and using a modeling language*" by: *1) identifying the entities, value objects (VOs) and aggregates; 2) dividing complex models into its aggregates, modules, bounded contexts or sub-domains - UML Packages; 3) modeling the domain as a UML Class Diagram; and 4) using the StarUML tool.*

The deliveries of each course according to the learning unit are the following:

- **Software Engineering Fundamentals**: A Software Requirements Specification (SRS) Document, an Architecture Definition Document, and a High-Quality Code Repository (free of bugs, vulnerabilities, and smells).
- **Contemporary Software Development**: A Multi-Branch Code Repository and a Simple CI/CD pipeline (build); a CI/CD pipeline (build + unit testing + quality control); and a CI/CD pipeline (build + unit testing + quality control + functional testing + performance testing + security testing [+ docker containerization]) + Issue Tracking.

4 IMPLEMENTATION

Around 40 students are attending these courses every semester, which are divided into 8-10 teams of 3-6 students each. In the first project follow-up session (1st week), the instructor briefly described the projects (to be developed or evolved) that must be submitted by each team and had 17 weeks for their development or evolution. The performance on these projects determines their final grade. Two projects were proposed:

- **Wiki for Call for Papers**: A software to collect information about computer science events (workshops, conferences, or journals), that allows to register, publish, and call for papers of upcoming events (e.g., *WikiCFP* or *Research.com*).
- **Event Manager**: A software for managing specific events on computer science, that allows registering an event and its different editions (by year), publishing the program (sessions and papers), and downloading the accepted papers (e.g., *conf.researchr.org/series/icse*).

Each team chooses a programming language (*Java*, *C#*, *Python* or *Javascript*) and works as an agile team, where members try to play the following roles and responsibilities:

- **A Product Owner/Customer** understands the requirements of the project, traces requirements to specific tasks (backlog items) in the product backlog through the *Trello* tool and a Kanban/Scrum board, sets the priority of tasks, assigns tasks to development team members, plans the releases by selecting the tasks to be added to the current iteration/sprint_backlog, reviews/helps their artifacts once a release is delivered and informs to the Team Leader and Scrum Master.

- **A Scrum Master** acts as a coach for the teams, facilitates the communication and coordination of the team members - task lists and meetings, ensures that the tasks are performed accordingly and timely, facilitates the communication and coordination of the team members, and oversees the iteration/sprint planning and release through the *Trello* tool and *Agile* practices like a *Kanban/Scrum board*. He/She also removes hurdles affecting project progress and team productivity. This role is carried out by the course instructor.
- **Development Team Members** are software engineers, who complete their tasks within the assigned deadlines (and incorporating agile practices like code refactoring) and participate in meetings about general aspects of the project such as the definition of requirements (domain model, architecture and code repository), assignment of tasks and responsibilities, as well as review meetings. A software engineer can act as a requirements engineer, designer, architect, developer, tester, DevOps engineer depending on the task assigned in the *Kanban board*. They also clone the team (*GitHub*) repository in their own local copy (*Git*) of the project, add/update changes to the team repository (*GitHub*), and update the status of their tasks (*Trello*).
  - **A Team Leader** is a more experienced software engineer on the team, which merges the changes in code repositories (*GitHub*), performs the CI/CD pipeline after a release is delivered and tracks (*Trello* or *GitHub Issues*) the issues (improvement or bug) after tests/reviews failed or a new requirement/feature is requested.

To achieve course competencies, students must be taught the practical applications of theoretical foundations. As such, we show more about Agile/DevOps workflows and encourage students to release their final projects. We have organized the laboratories as a sequence of steps carried out over the subject of the semester, which are designed to ensure that students produce the intermediary artifacts necessary to release the final product. Figure 3 depicts an overview of the sequence of laboratories as a workflow, where some industry practices and tools (e.g. *Clean Code*, *DDD*, *SonarQube* and *Jenkins*) incorporated by students are shown in boxes with dotted corners, the artifacts produced/updated by the students (e.g., *Kanban Board* or *Source Code*) are shown in regular boxes.

We were careful to choose the most popular tools in the industry and those that allow its customization and mainly its extension. And, hands-on experience was initially provided by demonstrations (e.g. CI/CD pipeline creation in *Jenkins*) made by the instructors, and later students were guided to setup their computers and provided with tools documentation. The laboratory assignments are described in the following.

Software Engineering Fundamentals (as in Fig. 3a):

- Lab 01 – Identifying Requirements: Bound the system from its context (*UML Use Case Diagrams and the StarUML tool*); and Identify FRs and NFRs (*DDD Ubiquitous Language and User Stories*).
- Lab 02 – Describing Requirements: Specify requirements through scenarios or goal-based techniques and templates (*Use Case (UC) Descriptions, BDD Specifications and the NFR Framework*).
- Lab 03 – Analyzing Requirements: Inspect (Requirements Templates – Writing Guidelines and Checklists) and Validate (Web or mobile app prototypes and the *Justinmind* or *Figma* tools) requirements to detect discrepancies, errors, and omissions (DEOs).
- Lab 04 – Managing Requirements: Prioritize, Trace (requirements to tasks), and Track their development (*Kanban board, the Trello and Slack tools*).
- Lab 05 – Domain Modeling: Present the domain of a software system by dividing it up into DDD modules, bounded contexts, or sub-domains (*DDD Entities, VOs, Services, Factories, Aggregates, Modules, Bounded Contexts or Sub-domains; UML Class Diagram; and the StarUML tool*).
- Lab 06 – System Architecture: Design the system architecture (subsystems, components or modules) by considering architectural patterns (*Layers, Microservices or Event-driven*), DDD&Layers (*presentation, application, domain, and infrastructure)* and the UML Component/Package Diagram (the StarUML tool).
- Lab 07 - Model-driven Development: Use the domain model (*Entities, VOs, Services, Aggregates, Modules, Bounded Contexts or Sub-domains*) of on an ongoing basis to guide the development (*presentation, application, domain, and infrastructure)* of an application, and generate its source code (MVC and ORM frameworks).
- Lab 08 – Coding Styles: Apply consistent programming styles [27] (*Monolith, Cookbook, Pipeline, Things, Error/Exception Handling, Persistent-Tables, Lazy-Rivers, Trinity and Restful*).
- Lab 09 – Clean Code: Demonstrate bugs, code smells and vulnerabilities, and fix them (*Coding Standards and Conventions, Clean Code and SonarLint IDE extension*).
- Lab 10 - SOLID Principles: Apply principles of object-oriented programming when building software that is easier to scale and maintain (*Clean Code and SOLID*).

Contemporary Software Development (as in Fig. 3b):

- Lab 01 – Software Repositories: Clone the team (*GitHub*) repository in their own local repository (*Git*), Create branches (master, development, and 1 per member or feature to be released in the current iteration/sprint) from the master branch, and add/update/merge changes to the master branch in the team repository.
- Lab 02 - CI/CD Pipelines: Construct a CI/CD Pipeline to automatically integrate and deploy a software system *(Jenkins or GitHub Actions*).
- Lab 03 – Automatic Build: Build a project by using automatic build systems (*Maven, Gradle, CMake, PyBuilder or Webpack*).
- Lab 04 - Static Analysis of Source Code: Perform team project analysis to make explicit bugs, code smells, and vulnerabilities (*SonarQube* tool).

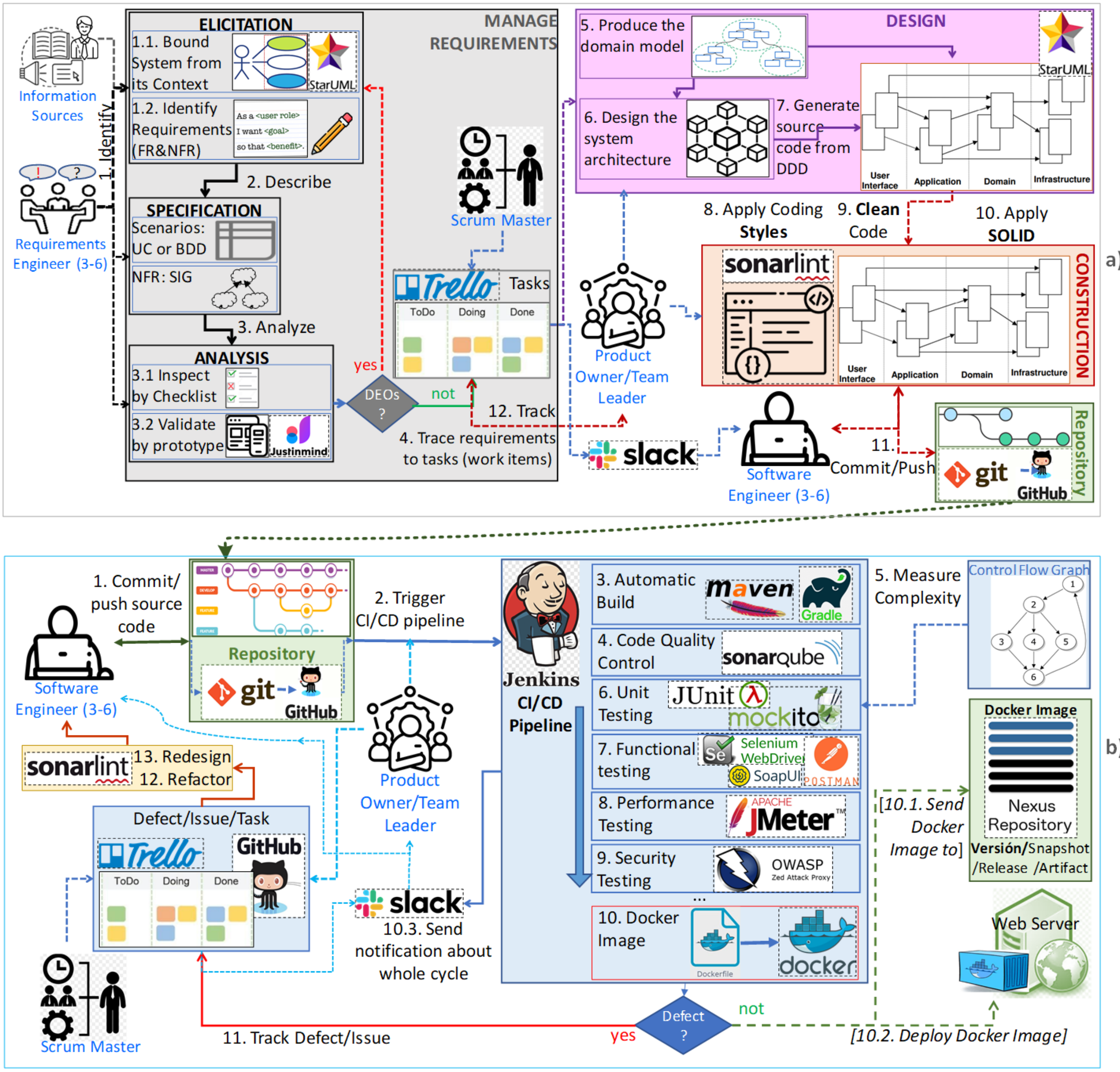

Figure 3: Assignments overview in software engineering fundamentals (a) and contemporary software development courses.

- Lab 05 – Measuring Complexity: Measure the readability, testability and maintainability of a program by developing a Control Flow Graph of the code (*Cyclomatic and Cognitive Complexity and SonarLint*).
- Lab 06 - Unit [and Integration] Testing: Design and Implement unit tests using a *xUnit* framework (*JUnit, GTest, unittest or Jest*) and the *TDD* approach.
- Lab 07 – Functional Testing: Design and Implement functional tests (*Selenium Web Driver, SoapUI or Postman* tools).
- Lab 08 - Performance Testing: Perform load, volume, and stress testing (*JMeter* tool).
- Lab 09 - Security Testing: Perform penetration testing (*OWASP ZAP* tool).
- Lab 10 – *Continuous Release: Deploy a software system as a container (Docker) in an online environment (Kubernetes).* This is optional because it will be done in a Cloud Computing course.
- Lab 11 - Issue Tracking: Track defects when detected (*Trello, GitHub Issues or Jira*) and Perform regression testing.

- Lab 12 – Refactoring: Identify bugs, code smells, vulnerabilities or new requirements/features, fix/add them, and apply regression testing (*SonarLint IDE* extension, xUnit framework, and TDD).
- Lab 13 – Redesign: Identify code fragments that are candidates to be replaced by *Design Patterns*.
- Lab 14 - *From Monolith to Microservices: Split the code within a Monolithic Application by dividing the code separately like Modules (DDD Modules, Bounded Contexts, and Anti-corruption Layer pattern) which will be migrated to Microservices and deployed as containers (Docker) in an online environment (Kubernetes).* This is optional because it will be done in a Cloud Computing course.

CI/CD requires *Trunk-based development* or Git Workflows[2] as branching strategies, i.e., all development occurs at the "master" branch of the repository, not on branches [19] [40]. Committing changes to the master branch triggers the CI/CD pipeline. This helps identify integration problems early and minimizes the amount of merging work needed. It keeps the master branch in a deployable state; and makes it much easier and faster to push out security fixes. However, for teams that are new to CI/CD, committing changes directly to master while keeping it deployable can be challenging before you have had time to develop a robust test suite.[3] Besides this fact, it is challenging to manage what is being included in each release and provide ongoing support for multiple team members. To oversee the changes being committed by the team members and encourage team members to take part in the team project; we adopted a *multi-branch strategy*.

Additionally, the instructors introduce the students to other tools available in the market, mainly tools and frameworks for automatic building, unit testing, and integration testing of *C++, C#/Python/PHP/Javascript* applications.

To expose students to the Agile/DevOps nature of their projects – teamwork, the laboratory sessions were also regular control points (follow-up or checkpoint) of their projects.

5 RESULTS AND LESSONS LEARNED

We report our last year's experiences (Period: 2022) of applying the teaching approach as well as our observations of conducted work, learning progress, and ***student outcomes*** (measuring the competencies): *unsatisfactory, partially satisfactory, satisfactory* and *excellent*. The assessment rubrics and the collected data are available as supplementary material.

Figure 4a shows the performance of 10 teams in the ***Software Engineering Fundamentals*** course. 9 teams managed their software requirements by using scenario-based formats and tracking them in a *Kanban board* through the *Trello* tool – EXCELLENT. All the teams defined their software architectures by the *layers pattern* and *DDD* recommendations (no *microservices* but using *aggregates and modules*) – SATISFACTORY. All the teams applied at least 5 coding styles (*Cookbook, Things, Error/Exception Handling, Persistent-Tables, Trinity, and Restful*) — EXCELLENT. 6 teams applied the coding standards provided by the programming language (*Java, Python or Javascript conventions*) and *Clean Code* recommendations – EXCELLENT. 5 teams applied at least 3 SOLID principles (*Single Responsibility, Interface Segregation, and Dependency Inversion*) – EXCELLENT. 4 teams applied at least 5 *DDD* practices (*Ubiquitous Language, Domain Entities&VOs, Factories, Repositories, and Modules*) – EXCELLENT; 2 teams applied satisfactorily at least 4 *DDD* practices – SATISFACTORY; and 4 teams applied at least 3 *DDD* practices – PARTIALLY SATISFACTORY.

Figure 4b shows the performance of 9 teams in the ***Contemporary Software Development*** course. Only 3 teams managed (add, update, and merge) to work with *multi-branch* repositories – EXCELLENT; the main difficulty is related to merging conflicting code in different branches. All the teams got to build a CI/CD pipeline on *Jenkins* – EXCELLENT. 6 teams automatically built their systems – EXCELLENT; the main difficulty is the lack of experience in the chosen programming language. 4 teams managed the integration between source code analysis (*SonarQube*) and *Jenkins* – EXCELLENT; 5 teams performed the analysis outside the CI/CD pipeline – UNSATISFACTORY. 7 teams performed unit tests into their projects and through the CI/CD pipeline – EXCELLENT. All the teams automated functional tests, but did not integrate into the CI/CD pipeline – SATISFACTORY. 1 team ran and integrated performance tests into the CI/CD pipeline – EXCELLENT; the other teams ran locally - SATISFACTORY. 8 teams ran security tests locally – SATISFACTORY. 8 teams managed defects found by CI/CD cycles through *GitHub Issues* – EXCELLENT. Only 1 team released its application as a Docker Container, the other teams found difficulties mainly due to lack of time.

Overall, 100% of the students achieved *SATISFACTORY* or *EXCELLENT* outcomes in Software Engineering Fundamentals skills; and 67% of the students achieved *SATISFACTORY* outcomes in Contemporary Software Development skills.

Student reactions to the assignments involving industry tools were generally positive and they perceived the laboratories beneficial. And, all teams worked collaboratively through a *Kanban board* and a team repository. However, we believe the following ***lessons*** can be taken for how to improve them going forward:

Preparation: Students must develop, test, evolve, and release a software product through a CI/CD pipeline as a final project. However, the level and depth of the prerequisite Platform-based Development course were insufficient. Therefore, Instructors of this course should prioritize web development and the use of frameworks for development and testing.

Tools Documentation and Examples: Part of the work done by the students to develop the final project was done outside of the course hours due to the limited time assigned to the course. Therefore, the instructor must prioritize the demonstration of some tools (e.g. Jenkins), examples (CI/CD pipeline creation), and documentation (e.g. GitHub documentation) to help students.

Technology Integration: Different tools (mostly open source and free) were studied, however, the students were challenged when they needed to integrate the different tools. Therefore, the instructor should prioritize the creation of a basic CI/CD pipeline, i.e., a pipeline mainly integrated with a build, quality control, and unit testing tools.

Course Content: CI/CD puts DevOps ideals into practice. Therefore, the updated courses focused on technical concepts like CI/CD;

[2] https://www.atlassian.com/git/tutorials/comparing-workflows
[3] https://www.jetbrains.com/teamcity/ci-cd-guide/concepts/trunk-based-development/

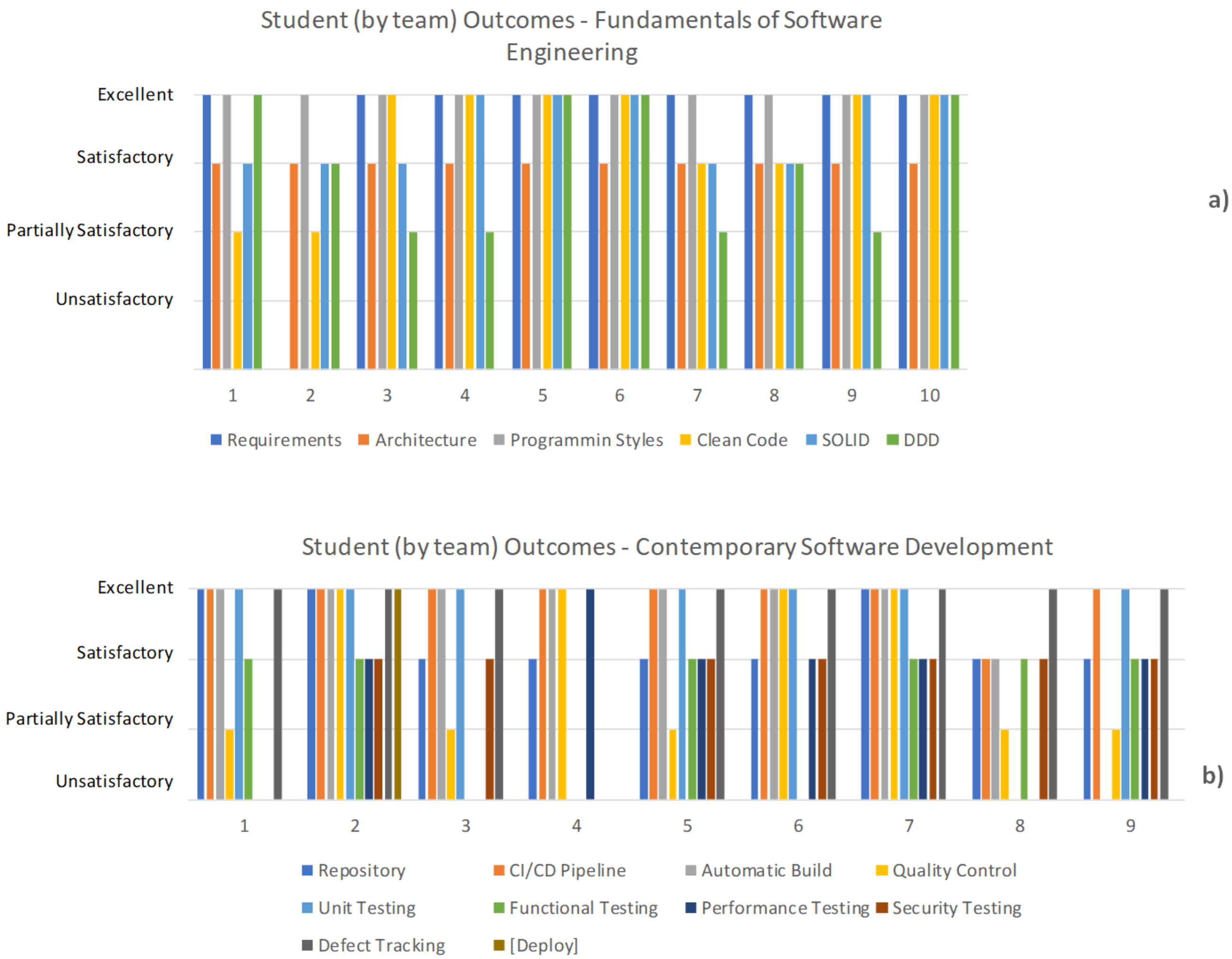

Figure 4: Measuring student outcomes in software engineering fundamentals (a) and contemporary software development (b) courses – Period: 2022.

however, some important technologies related to continuous delivery were not developed in-depth. Thus, the Cloud Computing course should prioritize infrastructure as code, containerization, service orchestration&observability, and cloud infrastructures. Finally, we believe that students must apply the acquired skills in Software Engineering and Cloud Computing courses to senior projects in a new course named "***Software Project***", which must include topics related to software planning, development, testing, release, deployment, configuration, and operation.

Teamwork: Collaboration is not natural for students. To have greater evidence of the participation of all team members, the instructor should individually monitor the students through the traceability (matrix or graph) between the number of tasks created and completed on the Kanban board and the number of commits/merges/rebases in the team repository.

Finally, we agree with the observation made by Winters in [24], "*Software Engineering presents a difficult challenge for learning in an academic setting*". DevOps is particularly challenging because it covers technical concepts, such as pipeline automation, and non-technical ones, such as teamwork (roles and responsibilities) and project management. Non-technical concepts will be more authentic and more relevant if and when our learners experience collaborative and long-term software engineering projects in *vivo* rather than in the classroom. Following Winter's suggestions, we focused primarily on technical concepts that are needed by a majority of new-grad hires, and that either are novel for those who are trained primarily as programmers.

6 RELATED WORK

Agile, CI/CD and DevOps have been topics of significant interest to academic and industry research [17] [25] [14] [7] [36] [12] [39]. Thus, some universities have begun to adapt the content of their study programs in order to satisfy the market needs and match the technical and non-technical skills of Computer Science students with Software Engineer job offers [21].

A few studies have investigated ways of including industry-relevant practices and technologies into existing software engineering fundamentals courses in undergraduate programs of computer science [6] [34] [18] and software engineering [20] [2] [37]. Other studies have reported experiences of including DevOps topics in new and advanced software engineering courses [31] [21] (e.g. DevOps Engineering, Continuous Delivery and DevOps or DevOps Culture and Mindset); and they have focused on continuous deployment topics and tools. On the other hand, more elaborated proposals, frameworks, and recommendations are focused on the design and implementation of DevOps concepts, topics, and tools into graduate (masters) programs or extensions of software engineering or computer science [22] [9] [20].

In the context of undergraduate programs of software engineering; DevOps topics have been included in several courses such as requirements engineering, project management, design&architecture, construction, testing, configuration management, release management and operations management. That is, they selected the concepts related to the software development&release cycle. However, due to the wide range of courses covered in Computer Science programs, it is particularly challenging to introduce DevOps within the context of software engineering fundamentals courses, i.e., connect abstract concepts to technical and non-technical skills needed for software engineers in the industry.

Due to these facts, some research about industry and academy gap [4] [37] [39] provide interesting advice for future adaptations of undergraduate programs of computer science and software engineering, and how these new topics should be taught [17] [13].

7 CONCLUSION

The objective of our proposal is to prepare undergraduate computer science students for agile (Kanban and Scrum), Extreme Programming, and DevOps-related practices (e.g. Test-driven Development, Coding Standards, Continuous Integration, and Refactoring) and tools. Our proposal profited from previous experiences with project-based software engineering at PUC-Rio [8, 33], and on the understanding of the new context posed by industry to IT professionals. This understanding led us to investigate ways of weaving new industry practices, tools, and environments into two existing software engineering fundamentals subjects, which were updated to incorporate and connect abstract software engineering concepts to industry-relevant practices and technologies. As such, students engaged in teams to develop, evolve, and release software products through a CI/CD pipeline.

To achieve the software engineering courses competencies, we have used a project-based teaching strategy and elaborated a sequence of laboratory assignments, which were carried out over the courses. Our initial results reveal that students acquired competencies (knowledge and skills) to: 1) work in teams through the Kanban and Scrum practices; 2) use team repositories to control version and changes; 3) construct CI/CD pipelines by integrating industry tools to build, asses code quality and test; and 4) develop and evolve a software project by defining a schedule, roles and responsibilities – tasks in a fixed period; and 5) track allocated tasks.

Informally, students' reactions have primarily been positive, however, we believe that the main limitation is related to release and operation management topics. That is, we have not covered properly some topics like containerization or infrastructure as code. However, this is not considered harmful because the software engineering fundamentals courses will be complemented by the Cloud Computing course. Another limitation is related to the strategy we use to assess teamwork; it does not analyze the consistency between the tasks assigned to students and their activity in the team repository. Finally, non-technical concepts of DevOps are particularly challenging; thus, we focused on technical concepts that are needed by a majority of new-grad hires.

Despite these limitations and others mentioned in the previous section, the students found the courses motivating. As evidence, we currently have students researching topics related to automated Software Engineering, DevOps, and Cloud to develop their undergraduate theses; in the past most opted for artificial intelligence or computer graphics. Another result that could be related to the updated courses is that most of our students now get jobs more quickly in the Latin American industry. In the past, most students opted for a master's degree to build confidence and apply to large software companies.

In the future, our intention is to 1) reevaluate our technology stack to determine whether a different set of tools provides better results (e.g. work with Jenkins and GitHub Actions and compare them); 2) improve our evaluation method for teamwork; 3) work in coordination with the cloud computing course to efficiently cover topics such as infrastructure as code, containerization, service orchestration and cloud-based architectures (microservices and event-driven); 4) update/renew a course of a software project to include topics related to software planning, development, testing, release, deployment, configuration, and operation; and 5) prepare an infrastructure to follow up the students after graduation, by devising surveys to be responded by past students and their managers, as to have a feedback of the strategy in place.

8 DATA AVAILABILITY

The authors confirm that the data supporting the findings of this study are available within the article and its supplementary materials.

REFERENCES

[1] Muhammad Ovais Ahmad, Jouni Markkula, and Markku Oivo. 2013. Kanban in software development: a systematic literature review. In *2013 39th Euromicro conference on software engineering and advanced applications*. IEEE, 9–16.

[2] Isaque Alves and Carla Rocha. 2021. Qualifying software engineers undergraduates in devops-challenges of introducing technical and non-technical concepts in a project-oriented course. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 144–153.

[3] Kent Beck. 1999. Embracing change with extreme programming. *Computer*, 32, 10, 70–77.

[4] Evgeny Bobrov, Antonio Bucchiarone, Alfredo Capozucca, Nicolas Guelfi, Manuel Mazzara, and Sergey Masyagin. 2020. Teaching devops in academia and industry: reflections and vision. In *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment: Second International Workshop, DEVOPS 2019, Château de Villebrumier, France, May 6–8, 2019, Revised Selected Papers 2*. Springer, 1–14.

[5] Pierre Bourque, Richard E. Fairley, and IEEE Computer Society. 2014. *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*. (3rd ed.). IEEE Computer Society Press, Washington, DC, USA. ISBN: 0769551661.

[6] Robert Chatley and Ivan Procaccini. 2020. Threading devops practices through a university software engineering programme. In *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 1–5.

[7] Héctor Cornide-Reyes, Fabián Riquelme, Rene Noel, Rodolfo Villarroel, Cristian Cechinel, Patricio Letelier, and Roberto Munoz. 2021. Key skills to work with agile frameworks in software engineering: chilean perspectives. *IEEE Access*, 9, 84724–84738.

[8] Lyrene Fernandes da Silva, Julio Cesar Sampaio do Prado Leite, and Karin Koogan. 2004. Ensino de engenharia de software: relato de experiências. In *Workshop de Educação em Informática (WEI)*. Salvador, Brazil, 994–1005.

[9] Yuri Demchenko, Zhiming Zhao, Jayachander Surbiryala, Spiros Koulouzis, Zeshun Shi, Xiaofeng Liao, and Jelena Gordiyenko. 2019. Teaching devops and cloud based software engineering in university curricula. In *2019 15th International Conference on eScience (eScience)*. IEEE, 548–552.

[10] Christof Ebert and Lorin Hochstein. 2022. Devops in practice. *IEEE Software*, 40, 1, 29–36.

[11] Eric Evans and Eric J Evans. 2004. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

[12] Samuel Ferino, Marcelo Fernandes, Elder Cirilo, Lucas Agnez, Bruno Batista, Uirá Kulesza, Eduardo Aranha, and Christoph Treude. 2023. Overcoming challenges in devops education through teaching methods. In *Proceedings of the 45th International Conference on Software Engineering: Software Engineering Education and Training* (ICSE-SEET '23). IEEE Press, Melbourne, Australia, 166–178. isbn: 9798350322590. doi: 10.1109/ICSE-SEET58685.2023.00022.

[13] Samuel Ferino, Marcelo Fernandes, Anny Fernandes, Uirá Kulesza, Eduardo Aranha, and Christoph Treude. 2021. Analyzing devops teaching strategies: an initial study. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering*, 180–185.

[14] Marcelo Fernandes, Samuel Ferino, Uirá Kulesza, and Eduardo Aranha. 2020. Challenges and recommendations in devops education: a systematic literature review. In *Proceedings of the XXXIV Brazilian Symposium on Software Engineering*, 648–657.

[15] CC2020 Task Force. 2020. *Computing Curricula 2020: Paradigms for Global Computing Education*. Association for Computing Machinery, New York, NY, USA. isbn: 9781450390590.

[16] Martin Fowler. 2018. *Refactoring*. Addison-Wesley Professional.

[17] Vahid Garousi, Gorkem Giray, Eray Tuzun, Cagatay Catal, and Michael Felderer. 2019. Closing the gap between software engineering education and industrial needs. *IEEE software*, 37, 2, 68–77.

[18] Ryan Gniadek, Margaret Ellis, Godmar Back, and Kirk Cameron. 2022. Integrating devops to enhance student experience in an undergraduate research project. In *2022 ASEE Annual Conference & Exposition*.

[19] Fergus Henderson. 2017. Software engineering at google. *CoRR*, abs/1702.01715. http://arxiv.org/abs/1702.01715 arXiv: 1702.01715.

[20] Mark Hills. 2020. Introducing devops techniques in a software construction class. In *2020 IEEE 32nd Conference on Software Engineering Education and Training (CSEE&T)*. IEEE, 1–5.

[21] Richard Hobeck, Ingo Weber, Len Bass, and Hasan Yasar. 2021. Teaching devops: a tale of two universities. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E*, 26–31.

[22] Rachel A Kaczka Jennings and Gerald Gannod. 2019. Devops-preparing students for professional practice. In *2019 IEEE Frontiers in Education Conference (FIE)*. IEEE, 1–5.

[23] Association for Computing Machinery Joint Task Force on Computing Curricula and IEEE Computer Society. 2013. *Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science*. Association for Computing Machinery, New York, NY, USA. isbn: 9781450323093.

[24] Association for Computing Machinery Joint Task Force on Computing Curricula and IEEE Computer Society. 2023. Computer science curricula 2023: version gamma. Retrieved Sept. 30, 2023 from https://csed.acm.org/wp-content/uploads/2023/09/Version-Gamma.pdf.

[25] Kati Kuusinen and Sofus Albertsen. 2019. Industry-academy collaboration in teaching devops and continuous delivery to software engineering students: towards improved industrial relevance in higher education. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, 23–27.

[26] Xabier Larrucea, Izaskun Santamaria, Ricardo Colomo-Palacios, and Christof Ebert. 2018. Microservices. *IEEE Software*, 35, 3, 96–100.

[27] Cristina Videira Lopes. 2020. *Exercises in programming style*. Chapman and Hall/CRC.

[28] Robert C Martin. 2017. Clean architecture. (2017).

[29] Robert C Martin. 2009. *Clean code: a handbook of agile software craftsmanship*. Pearson Education.

[30] Sam Newman. 2019. *Monolith to microservices: evolutionary patterns to transform your monolith*. O'Reilly Media.

[31] Nicolás Paez and Hernán de la Fuente. 2022. Software engineering education meets devops: an experience report. In *2022 IEEE Biennial Congress of Argentina (ARGENCON)*. IEEE, 1–7.

[32] Candy Pang, Abram Hindle, and Denilson Barbosa. 2020. Understanding devops education with grounded theory. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training*, 107–118.

[33] Roxana Lisette Quintanilla Portugal, Priscila Engiel, Joanna Pivatelli, and Julio Cesar Sampaio do Prado Leite. 2016. Facing the challenges of teaching requirements engineering. In *Proceedings of the 38th International Conference on Software Engineering Companion* (ICSE '16). Association for Computing Machinery, Austin, Texas, 461–470. isbn: 9781450342056. doi: 10.1145/2889160.2889200.

[34] Miloš Radenković, Snežana Popović, and Svetlana Mitrović. 2022. Project based learning for devops: school of computing experiences. In *E-business technologies conference proceedings* number 1. Vol. 2, 127–131.

[35] Sander Rossel. 2017. *Continuous Integration, Delivery, and Deployment: Reliable and faster software releases with automating builds, tests, and deployment*. Packt Publishing Ltd.

[36] Miguel Ángel Sánchez-Cifo, Pablo Bermejo, and Elena Navarro. 2023. Devops: is there a gap between education and industry? *Journal of Software: Evolution and Process*, e2534.

[37] Walter Schilling. 2022. Wip: integrating modern development practices into a software engineering curriculum. In *2022 ASEE Annual Conference & Exposition*.

[38] Ken Schwaber and Mike Beedle. 2002. *Agile software development with scrum. Series in agile software development*. Vol. 1. Prentice Hall Upper Saddle River.

[39] IEEE Computer Society. 2022. Swebok evolution. Retrieved Sept. 30, 2023 from https://www.computer.org/volunteering/boards-and-committees/professional-educational-activities/software-engineering-committee/swebok-evolution.

[40] Titus Winters, Tom Manshreck, and Hyrum Wright. 2020. *Software engineering at google: Lessons learned from programming over time*. O'Reilly Media.