Beyond Coding: Lean Learning for Real-World Software Engineering

Chatley, Robert[a] Fraca, Estibaliz[a] Bailey, Jason[a] Procaccini, Ivan[a] Amin, Zaki[a]

Department of Computing, Imperial College London, United Kingdom[a]

Corresponding Author's Email: r.chatley@imperial.ac.uk

KEY WORDS: software engineering, lean learning, industry-relevant experience, engineering module design, curriculum design

ABSTRACT

Computer Science education often focusses heavily on programming, but modern software engineering demands a broader set of skills beyond writing code. We present a case study of a *Software Systems Engineering* module, designed to emphasise non-programming aspects of software development from early in the degree.

The core philosophy of the module is that experiencing the full software development and deployment lifecycle is more valuable in a limited timeframe than building sophisticated features requiring a lot of coding. Students start from a skeleton web application and focus on deployment, automated testing and quality checks, connection to a database, and integration of public APIs. Later, students explore concepts such as microservices and cloud scaling. This approach teaches the "engineering" side of software development, instead of focussing on writing lines of code.

Beyond knowledge of programming languages, software engineers must integrate diverse tools, libraries, and APIs. Working productively in a professional development team also requires familiarity with version control, continuous integration, automated quality checks, and cloud deployment. While many curricula would categorise these as advanced topics to be covered once programming has been mastered, we see these as threads that can be taught in parallel.

We believe this is particularly crucial for students in conversion master's degrees. These students come from non-computing backgrounds and have limited time to learn not only core computer science theory, but also industry-relevant engineering experience.

INTRODUCTION

With the growth of the technology sector, there is an increasing demand for software engineering expertise. Conversion degrees, particularly at the master's level, are very much in demand. These offer the opportunity for graduates in other fields to add technical computing skills to their existing discipline knowledge, putting them in a position to work in jobs where they develop new software products to advance the state of practice. It is also

common for career-changers to use conversion degrees to re-train in depth, to open up new future career directions (Li, 2025).

A key difficulty in designing an effective conversion master's programme is the short timeframe. Typically, students follow a one year course compared with three or four years for an undergraduate programme. Our own master's programme spans12 months, from October to September. Building both breadth and depth of skills and knowledge within this short time period is challenging. Traditionally, Computer Science programmes start with a focus on core skills in programming, algorithms and data structures. Undergraduate programmes may spend the whole of the first year on these fundamentals, before moving on to more applied courses and projects in later years (Chatley and Field, 2017).

The role of a modern software engineer comprises not only coding individual components but combining and integrating components into a larger system. Software engineers also need to be familiar with techniques for integrating components, quality assurance, agile methods, tools to support incremental and iterative delivery, databases, and designing systems deployed in the cloud (SFIA Foundation, 2022). This combination of topics, which might previously have been thought of as more of the remit of the "software architect" (Fowler, 2019), have become part the daily work of most practising software engineers.

Many works in the literature have looked at teaching these Software Engineering skills as part of a university degree. Pantoja Yépez et al. (2024) surveyed 56 case studies with a focus on skills related to "software architecture". Their study found that these skills were typically categorised as "advanced topics" and are typically developed later in a degree program, or even at the postgraduate level, once students have already built a solid foundation in coding and basic software development practices. In a one-year master's programme, we do not have the luxury of spending so much time on fundamentals before proceeding to these "advanced topics", but we still want to equip our students with a broad range of industry-relevant software engineering skills.

AIMS AND OBJECTIVES

Our aim was to design a learning experience that would prepare conversion students for industrial software engineering roles, exposing them to modern techniques and technologies, within the compressed timeframe of a one-year master's programme. A particular challenge here was that we could not assume any previous experience with coding and software development - students would learn introductory programming in another module within the programme. However, rather than wait until that module was completed, we wanted to find a way to start teaching broader software engineering skills from the start of the degree.

PROBLEM AND INTERVENTION

In the previous iteration of our conversion master's curriculum, we split the year into two main teaching terms. In the first term we concentrated on traditional computing

fundamentals, with a large emphasis on programming in C++. By the end of this first term, most students had become competent C++ programmers, but they had worked individually rather than collaboratively, and had written self-contained programs, rather than developing larger software systems integrating multiple components and technologies.

In the second term, we had the students form teams of 5-6 and work collaboratively on a project developing a software system to solve a particular problem. These team projects were typically set and supervised by one of the academic staff, who acted as both the customer and the assessor. While Project Based Learning (PBL) is traditionally seen as a good approach for teaching software engineering skills (Kokotsaki, Menzies and Wiggins, 2016; Souza, Moreira and Figueiredo, 2019), and indeed many teams did succeed in building software to meet their project brief, we perceived several problems:

1) There was great variety in the project briefs set by the different supervisors. This meant that different groups had very different technical challenges to solve and had very different learning experiences.
2) After the first term, the students were competent C++ programmers, but few of the group projects involved writing C++.
3) It was hard to deliver a lecture course that was relevant to all groups, covering topics that much of the class would find helpful for their projects.
4) Students felt quite unsupported and did a lot of self-study to learn new programming languages and technologies.
5) Student teams typically did not apply rigorous engineering practices around things like quality assurance, or scalable infrastructure, partly because of perceived time pressure, but also because they had not been taught how to apply them before they started their projects.

Given these problems, we decided to redesign the way software engineering skills are taught in this MSc programme, replacing the varied group projects module with a more structured module called *Software Systems Engineering (SSE)*. The new module runs across two terms, starting right at the beginning of the degree programme. The module design emphasises non-programming aspects of software development from early on.

We designed the module using *lean* techniques, following the pedagogical principles discussed by Parsons and MacCullum (2019). Our aim was to regulate the flow of learning, covering many different topics, but without overwhelming the students with theory before putting each concept into practice. Having successfully followed this approach in other modules (Chatley and Field, 2017), we divided the content into bite-sized chunks, each of which combined a little theory with a practical application, structured as a weekly cycle:

- Minimal theory – lectures with just enough background to support practical work
- Hands-on exercises – a focussed task (2-3 hours) applying a key practice
- Easily assessable submission – to enable a weekly rhythm, a concrete deliverable that was quick and easy for Teaching Assistants (TAs) to check - e.g. a link to a live website, a GitHub repository, or a short document with screenshots.

The topics covered in the module are detailed in Table 1. We aimed for a broad coverage, rather than going deep into a smaller number of topics. Each topic was supplemented with an exercise. Students were offered lab sessions where they were encouraged to complete the exercise asking for help if needed. For most cases, students worked in pairs.

Topic / skill	**Exercise Description**
Web Sites and Web Servers	Students create a simple static web site using GitHub pages. They are initially encouraged to create a GitHub account and commit/push small changes
Web Applications	Students create a simple dynamic web application using Python and Flask. They can then manually deploy their application to the public internet via a bespoke PaaS
Build Pipelines	Students use GitHub Actions to create a build pipeline to automate the deployment of their web application
Continuous Delivery	Students implement unit tests and other checks for their code, then include them in their build pipelines
Using APIs	Students investigate a publicly available data API and extend their application to consume its data
Databases	Students are introduced to relational databases and learn to query and update data with SQL
Cloud	Students deploy their web applications to the cloud (in our case using Microsoft Azure)
Containers	Students are introduced to Docker and using containers to deploy applications to the cloud
Microservices	Students learn the concepts of Microservices to develop a system of multiple coordinating components

Table 1. Topics delivered in the Software Systems Engineering module

This new design format for the module focuses on the students experiencing the full software development and lifecycle, which is considered more valuable than building sophisticated features requiring a lot of coding in the mentioned limited timeframe. By replacing the more free-form projects with this structured programme, we know that each student has covered the same technical topics, we can support their learning better through lectures and tutorials, and we can scaffold their learning to incrementally introduce new theory and tools as they build more sophisticated systems over the duration of the course.

OUTCOMES

We collected some qualitative data to analyse whether the lean-learning approach we followed was appropriate for this context. Students were sent an anonymous survey one month after the end of the module. The goal of the survey was to gather the students' opinion on the following questions:

- Do they feel that they are industry ready after completing the module?
- Do they deem it acceptable to teach "advanced" topics earlier in their degree, particularly to those without prior coding experience?
- Did the "*broad rather than deep, and practical rather than theoretical*" module philosophy work for them?

In their responses to the first question, students listed topics that they felt prepared them the most for industry, such as "*[…] considerations of cost, speed, reliability; practical cloud computing exposure; CI/CD and automation, for efficient development*" and "*[…] getting very comfortable with git, testing, and automatic checks when deploying*". Students also highlighted how SSE topics had come up in job descriptions and interviews: "*I actually had a question about GitHub actions in one of my job interviews […]*" and "*[…] lots of the things we utilised in SSE (idea of containers ci/cd etc) appear in job requirement*". The general feeling was that the course did provide a good coverage of industry-relevant topics.

In considering the teaching of "advanced" topics earlier in their degree, although the students rightly acknowledged their limited coding experience ("*As conversion students, we do not master any of the programming languages like Python, Java, or html […]*"), they found that learning about common professional tools and patterns *by doing* helped them attain a global intuition of "what to do and why" on approaching a software engineering project: "*[…] learning about GitHub actions was definitely useful, not only in the practical sense but also in terms of better understanding […] during development*".

We finally asked students to describe how they felt about the overall effectiveness of the module. The students commented saying that the module "*provided a nice contrast to some of the deeply theoretical modules we cover*" and that "*it was nice to do anything with the new topic in the seminars. Then, with foundations laid, it was easy to self-learn and go deeper*". Some of them still found some of the more technical topics difficult to tackle: *"[…] first time being introduced to these concepts from an implementation perspective is always hard*"; "*I still don't really know how to use Docker*". While this suggests that our balance of broad coverage vs in-depth mastery will need some refining, providing students with a lightweight foundation where they can freely choose to go deeper through self-study seems positively received.

DISCUSSION AND CONCLUSION

The course structure, following the lean-learning weekly-cycle approach described in the previous sections, creates momentum through weekly assignment deadlines. By

concentrating on practical application, staff can spend less time on transmission of content, and more time to one-on-one and small group interactions in labs. Other case studies in the literature, such as Anslow and Maurer (2015), show the benefit of such weekly cycles combining theory and practice. However, in their case, Anslow and Maurer focussed on project management and process, while we wanted to cover a wide range of technical topics.

Students incrementally build knowledge and skills week-by-week. We observed that within just six hours of tuition, students progressed from having no web development experience to deploying a functional application on the public internet. This rapid advancement offers students a tangible sense of achievement while reinforcing a grasp on fundamental engineering concepts.

The survey completed by students after the course shows that they feel they have acquired skills that are demanded in industry. The lean methodology of the course, described as *broad rather than deep* and *practical rather than theoretical*, structured in small chunks, seems to have been effective in gradually growing skills in a practical way from the beginning of the module.

However, there is still a sense that, without a depth of technical knowledge, and particularly programming skills, some of the topics covered have not been *deeply* understood. Due to the rapid spread of AI tools supporting software development, many students were able to build "working" applications without truly understanding what they had done or how the system they built worked internally.

Given the rapid pace of change in the technology sector, particularly after the advent of AI, we have open questions on whether it is more effective to teach fewer topics in depth down to the low-level fundamentals, or to focus instead more on higher-level concerns such as data governance, security, cost awareness, energy usage and sustainability, and delegate the lower-level coding tasks to an AI. These are perhaps the main concerns of future software engineers, as their daily work moves beyond coding.

REFERENCES

Anslow C. and Maurer, F. (2015). 'An Experience Report at Teaching a Group Based Agile Software Development Project Course', *In: Proceedings of the 46th ACM Technical Symposium on Computer Science Education (SIGCSE '15).* ACM, New York, NY, USA, pp. 500–505. https://doi.org/10.1145/2676723.2677284

Chatley, R. and Field, T. (2017) 'Lean learning: Applying lean techniques to improve software engineering education', *In: 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, pp. 117–126. https://dl.acm.org/doi/10.1109/ICSE-SEET.2017.5

Kokotsaki, D., Menzies, V., and Wiggins, A. (2016). 'Project-based learning: A review of the literature. Improving Schools', **19**(3), 267-277. https://doi.org/10.1177/1365480216659733

Fowler, M. (2019) *Software Architecture Guide*. Available at: https://martinfowler.com/architecture/ (Accessed: 7 May 2025).

Li, Z. (2025) 'Fostering qualified software engineers via software engineering conversion programmes', *In: Proceedings of the 37th IEEE Conference on Software Engineering Education and Training (CSEE&T 2025).*

Pantoja Yépez, W., Hurtado Alegría, J., Bandi, A. and Kiwelekar, A. (2024) 'Training software architects suiting software industry needs: A literature review.' *Education and Information Technologies.* **29(**9), pp. 10931-10994 https://dl.acm.org/doi/10.1007/s10639-023-12149-x

Parsons, D., MacCallum, K. (eds) (2019) *Agile and Lean Concepts for Teaching and Learning*. Springer, Singapore. https://link.springer.com/book/10.1007/978-981-13-2751-3

SFIA Foundation (2022) SFIA View: SFIA 9 Software Engineering Competencies. *Available at: https://sfia-online.org/en/framework/sfia-9 (Accessed: 7 May 2025).*

Souza, M., Moreira, R. and Figueiredo, E. (2019). 'Students Perception on the use of Project-Based Learning in Software Engineering Education'. *In: Proceedings of the XXXIII Brazilian Symposium on Software Engineering (SBES '19)*. Association for Computing Machinery, New York, NY, USA, 537–546. https://doi.org/10.1145/3350768.3352457