BUILDING LANGUAGE-AGNOSTIC EDUCATIONAL MATERIALS FOR SOFTWARE ENGINEERING COURSES WITH LLM

Nikola Valchanov, Todorka Terzieva

Abstract. This study investigates the use of LLMs to create code examples in Java and Node.js from a single instructor-maintained C# codebase, with the goal of enhancing students' understanding of cross-language software engineering concepts. The paper presents a workflow for migrating examples of object-oriented design and architectural patterns, written in C# to Java and NodeJS through automated translation using LLMs. The resulting codebases acknowledge specific code structure and best practices for each respective language and framework.

Findings indicate that LLMs can reliably preserve architectural intent while adapting implementations to language-specific idioms, enabling instructors to maintain one codebase while offering students a multilingual perspective on core engineering principles.

Key Words: *Large Language Model (LLM), Artificial Intelligence (AI), education, Code Generation*

Large Language Models – Introduction

Large Language Models (LLMs) can be integrated in almost every aspect of a modern enterprise regardless of its field of operations. LLMs are increasingly integrated into software development where their potential to support multilingual code comprehension is used to build features and/or migrate existing codebases across versions of a given library, language or framework. While highly valuable in software engineering this LLM capability provides means to assist with building implementation examples showcasing general principles for solving a given set of problems across different languages, libraries and frameworks [1].

The introduction of Large Language Models (LLMs) started a global change that is affecting all industries. Each LLM is trained on large data sets, enabling them to perform reasoning, generation, and transformation tasks that previously required human expertise.

LLMs can be split into several categories based on their specialization. Some of the most notable categories are:

- General purpose;
- Code generation;
- Image and video processing;
- Audio and speech.

It is evident that combination of domain-specialized LLMs and orchestration of their interaction enables the software industry to build tools that incorporate data analysis, reasoning, engineering and content creation.

One of the most impacted industries by the emergence of LLMs are Information technologies (IT). LLMs handle coding, accelerate prototyping, enhance documentation, support architectural decision-making, and enable the use of entirely new workflows such as conversational debugging and automated refactoring. As a result, software engineering is shifting from manual code production toward AI-augmented development, where developers supervise, validate, and orchestrate machine-generated solutions.

Traditional development is centered on writing code line by line, guided by mental models of architecture, algorithms, and constraints. The introduction of LLMs in the process shifts the focus from writing code to prompt and context management. This leads to changes not only in the process, but in the requirements towards the engineers that operate the LLMs. When coding is left to the LLM, the quality of the output often depends mainly on the engineer's ability to structure information and maintain coherence across prompts and code generation iterations. All LLMs have context limits which means that the quality of the output in each interaction within a single conversation is directly dependent on how context information is structured and provided with the prompt.

Although disruptive, this shift does not eliminate the role of the software engineer as LLMs do not give deterministic solutions thus supervision and correction by a specialist is required at each iteration. At the same time, it opens a new set of problems related to how prompt context is structured and provided to the LLM. Tracking relationships and dependencies are directly correlated to the quality of the code the model produces.

Although code generation is mainly associated to the IT industry, it can be used in IT education as well. An example would be creating sets of small programs that illustrate specific concepts. Generating such solutions is achievable using standard tools available in the industry. Another example would be migrating an existing set of solutions to a different programming language.

This article presents an approach for building language-agnostic educational materials for software engineering courses using LLMs by translating solutions from C# to Node.js and Java.

Migrating Projects

Migration is a crucial process in software development that involves modernizing and adapting systems to ensure ongoing functionality and compatibility. This may include, but is not limited to, updating or restructuring existing code, improving its organization, making it easier to understand and maintain, transitioning to newer versions of programming languages to leverage new features and address potential security vulnerabilities, updating dependencies such as APIs and libraries to modernize the codebase, fix bugs and improve performance, and adapting to changes in underlying frameworks and platforms that applications depend on [5].

Let's consider using a project as source (source project) for the migration process and set a goal to generate a new codebase (target project) in a different language. The standard approach towards such migration would include (but in most cases will not be limited to) the following steps:

- Requirements and scope analysis;
- Architectural and design analysis;
- Migrate code.

Requirements and scope analysis step would include documenting the split of the solution into domains, layers and modules and their respective connections. This step should result in detailed description of the functionality of the project, the split of functional domains within it and how they are connected.

Architectural and design analysis step deals with how the code is structured in the source project and how it should be structured in the target project. This includes folder structure, classes, interfaces, hierarchies and abstractions. This step considers what the build model for the target project is and how, based on best practices, it can accommodate the original codebase. The result of this step should cover the class hierarchies and how they are structured in the project layers and modules.

Migrating the code is the actual migration step where the code is migrated from the source to the target project.

Although the steps above belong to the traditional approach, they are still useful when migrating projects with LLMs. In an LLM-centered codebase migration, the main change should not be the artifacts created at each step. Instead, the focus should be on how the information used to create those artifacts is collected.

We assume that the basic set of data items to support an LLM to properly migrate code is as follows:

- Project structure;

- Project layering;
- Modules;
- Code file relationships;
- Actual code to migrate.

The list above contains the context that is included in the basic manual code migration process. These artifacts can serve as foundation for creating:

- Project architecture layout;
- Class-to-module and class-to-layer association;
- Migration plan.

The list above should serve as foundation for the context needed for migrating a project by an LMM. Some of the most significant problems that this context addresses are:

- Ability to segment context into smaller interactions within a conversation – LLM context window often doesn't fit the whole codebase;
- Ability to identify relationships – the LLM often would duplicates classes due to its inability to identify relationships within the codebase;
- Proper file system and code structure – the LLM doesn't structure the code properly due to its inability to identify how classes map to layers and modules.

Code Migration Tool

For the implementation of a code migrating tool that handles the case described above, we need to plan the following modules:

- Code structure extractor;
- Migration service.

The platform running the LLM locally for this implementation is Ollama. The model supporting the migration tool is qwen2.5-coder: 14b. The Qwen2.5-Coder: 14B model is part of Alibaba's specialized programming series, optimized for code generation, correction, and reasoning. The 14-billion-parameter version offers an excellent balance between performance and hardware requirements, performing remarkably well on complex tasks that smaller models (such as the 7B) sometimes fail to handle [2]. Choosing this model is a strategic decision for developers looking for a balance between high performance and the ability to run locally on standard hardware [10]. Some main reasons to choose this particular model:

- Understanding entire projects: With its 128K token context window, the model can analyze not only individual functions but also entire files or small codebases at once, making it ideal for complex debugging tasks.
- Broad language support: Trained on 5.5 trillion tokens, it handles over 92 programming languages, including rarer or more specific ones.
- Versatility (General Knowledge): Although specialized for code, the model retains high levels of general logic and mathematical skills (GSM8k, MATH), which helps it better understand complex business requirements, not just syntax.
- Freedom and Privacy: Using it via Ollama allows full control—the code never leaves the local machine, which is critical for the security of sensitive projects.

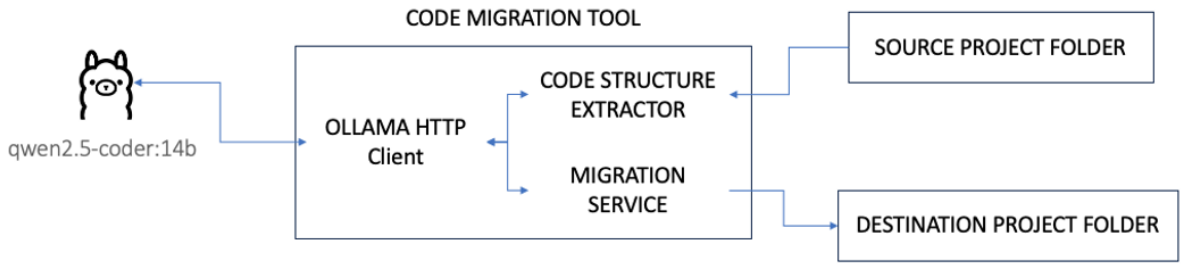

Figure 1. Code Migration Tool Architecture

The code structure extractor analyzes the codebase. It identifies the relative path of each file from the project root and extracts its contents. It also examines each file to identify all referenced namespaces, as well as the classes defined within the file and their corresponding namespaces.

The information extracted for each file is structured into JSON format and supplied to the LLM as context. Within this conversation the LLM is prompted to analyze the provided structure and separate all code files into layers and modules. The context data of each file is then enriched with layer and module information.

Once all information is in place the JSON-serialized file data is sent to the LLM as context for building a migration plan split into steps. Each step contains a step summary, a prompt describing the step and a list of files that should be within the migration scope of the step.

Once the migration plan is complete the tool is ready to process the migration.

The Migration service orchestrates the migration process. It iterates through the steps of the migration plan. For each step the migration service builds a prompt with context containing:

- All files in-scope of the step;
  - File Path;

  - Module and Layer;
  - Relationships with other files;
  - File Content.
- The Migration Plan summary;
- The Migration Step;
- The format of the output (JSON).
  - Path in the new project structure;
  - Content.

The JSON result from each step contains a JSON document with file content and path in the new project structure. The tool populates the new project root accordingly.

Building Educational Materials

The proposed method of migrating codebases enables trainers to build language-agnostic educational materials that are not tightly bound to a single programming language.

The proposed approach consists of the following steps:

1. Trainer builds a software engineering course consisting of abstract concepts and general problems discussed in each respective topic;
2. The trainer implements (in C#) a set of solutions to the problems associated with the topics of the course;
3. Using the tool presented in this article the trainer generates additional training materials (solutions to the problems from 2.) in different programming languages using LLMs.

The result is a course with a set of topics presented in language agnostic manner. Each topic is supported by a set of practical implementations in a range of programming languages, where most of the codebases are generated by LLMs.

The benefit from the above is that students are exposed to a wider range of technologies while discussing fundamental concepts. At the same time the amount of invested effort for migrating the examples from the original implementation to new languages is minimal.

A practical example would be the topic of “Data types and variables” in one of the most fundamental courses in software engineering – “Introduction to Programming”.

The trainer presents the concept of data types and variables through abstract visualizations during the theoretical component of the training. This portion of the

training materials is usually in the form of presentations and text-based problem definitions.

For the practical component of the training, the educational materials are solutions to the text-based problems. Let's consider the following example:

Problem: Create a program that stores a person's basic information in variables and prints it to the console. Define variables for:

First name (string)

Age (integer)

Height in meters (floating-point number)

Is student (boolean)

Assign values directly in the code.

Print all values to the console in a readable format.

For the purposes of this article, let's consider a case where the trainer creates a solution in C#. If left at that – the course becomes tightly bound to a single programming language.

Using the presented tool, the trainer can generate additional educational materials using LLMs that are derived from the existing C# solutions. This way the practical component of the course allows students to see solutions in a range of programming languages, identify differences and engage in discussions. This enables students to build a broader and technology-agnostic understanding of the discussed theoretical topic.

Results

For the experiment a set of solutions (console applications) to problems related to object-oriented programming were translated form C# to Node.js and Java.

The domain areas that the set of problems covered were as follows:

- Basic Data Structures;
- Basic inheritance;
- Complex example (Entities, Repositories, Services, Views).

The results are described in the tables below:

Table 1. Test results

Problem	Java	Node.js
LinkedList – 130 lines	2 success, 0 failed	2 success, 0 failed

Queue – 117 lines	2 success, 0 failed	2 success, 0 failed
Square -> Rectangle – 93 lines	3 success, 0 failed	3 success, 0 failed
Shape -> Square -> Rectangle – 120 lines	4 success, 0 failed	4 success, 0 failed
Console Phone Book (Entities, Repositories, Views) – 1684 lines	17 success, 2 failed	19 success, 0 failed

In the complex example the migration completed successfully with syntax errors related to translating C# generics to both Java and Node.js. The issues were resolved by re-running the migration step and providing the compilation error.

The results from this study indicate that LLMs can easily assist with the migration of simple to moderately complex examples for fundamental software engineering courses.

Automating the migration of this type of software programs is a step towards automated migration of full course materials across a wide range of software engineering languages and platforms. The ability of LLMs to migrate solutions, maintaining best practices for the target platform of the migration can strongly benefit students and by widening the range of technologies they're exposed to. This is especially valid for fundamental theoretical topics, especially in disciplines like introduction to programming, object-oriented programming and data structures.

References

[1] G. Lim, *Ollama Crash Course: Build Local LLM powered Apps*, Amazon, 2025, ASIN: B0DXB4SWG9

[2] L. Chapman, *Ollama in Action*, Amazon, 2025, ASIN: B0F9452Z3J

[3] M. Krishnamurthy, *Learning Qwen from Qwen, Leomohan Publications*, 2025, ASIN: B0FM6Y8GY4

[4] Y. Wang, A Multi-Perspective Investigation into Code Migration for Large Language Models, *Applied and Computational Engineering*, 2026, 135–145, DOI: 10.54254/2755-2721/2026.AS32228, ISSN: 2755-2721

[5] B. Mateus, M. Martinez, Ch. Kolski, Learning migration models for supporting incremental language migrations of software applications, *Information and Software Technology*, 2023, Vol. 153, ISSN: 0950-5849, https://doi.org/10.1016/j.infsof.2022.107082

[6] K., Ketan, Large Language Models for Code Generation, Medium, 2023, available: https://blog.fabrichq.ai/large-language-models-for-code-generation-f95f93fe7de4

[7] O. Mendelevitch, Large Language Models for Code Generation – Part 1, Vectara, 2023, https://www.vectara.com/blog/large-language-models-llms-for-code-generation-part-1

[8] J. Baierl, Applications of Large Language Models in Education: Literature Review and Case Study, UCLA, 2023, available: https://escholarship.org/uc/item/6kf0r28s

[9] M. Heller, LLMs and the rise of the AI code generators, InfoWorld, 2023, https://www.infoworld.com/article/2338500/llms-and-the-rise-of-the-ai-code-generators.html

[10] Qwen2.5-Coder Technical Report, 2024, available: https://arxiv.org/pdf/2409.12186

Nikola Valchanov[1,*], Todorka Terzieva[1]

[1] Paisii Hilendarski University of Plovdiv,

Faculty of Mathematics and Informatics,

236 Bulgaria Blvd., 4027 Plovdiv, Bulgaria

Corresponding author: nvalchanov@uni-plovdiv.bg