An Experience Report: Lessons in Progressive Project Design with LLM Support*

Chris Alvin
Computer Science Department
Furman University
Greenville, SC 29613
{chris.alvin}@furman.edu

Abstract

This paper explores the use of Large Language Models (LLMs) such as Claude for generating programming assignments in an advanced data structures course. While LLMs offer promising assistance in educational content creation, we find that effective use requires significant meta-programming and iterative refinement. Drawing from experience transitioning a course from Java to C# and implementing a compiler-like project across multiple assignments, we discuss both the benefits and challenges of LLM-assisted assignment creation. Our experiences suggest that while LLMs expedite code authorship and significantly accelerate the development of unit tests, they require substantial subject matter expertise and customization to produce appropriately scaffolded educational materials.

1 Introduction

The emergence of Large Language Models (LLMs) like Claude, ChatGPT, and others has sparked considerable interest in their application to educational contexts. These AI tools offer the potential to assist educators in developing course materials, creating assignments, and generating examples. However,

the effectiveness of these tools in creating programming assignments that are appropriately scoped, pedagogically sound, and technically accurate remains an open question.

In this paper, we consider the use of LLMs for creating a sequence of programming assignments in an advanced data structures course that transitioned from Java to C#. The course incorporates software engineering practices such as version control with Git/GitHub, design patterns, and unit testing, while building toward a cohesive project over the semester. Although LLMs can streamline the assignment creation process, the approach benefits from an interactive development cycle in which instructors can gradually refine the output to better align with their educational goals.

Initial interactions with LLM-assisted assignment generation revealed both promising capabilities and significant limitations. While LLMs could quickly generate assignment ideas and sample code, these outputs often required substantial refinement to match course learning objectives and student capabilities.

The gap between raw LLM output and pedagogically sound assignments highlighted the need for a structured approach to LLM-assisted content creation. This paper documents the twists and turns in developing 9 assignments that build toward elements of a compiler system for a very simple declarative language that we call DEC. Our general goal is to offer guidance to other educators seeking to incorporate similar approaches into their course development workflows. Lessons learned are highlighted throughout the text: ***lesson.***

2 Background: The Course

Our advanced data structures course serves as a linchpin in our liberal arts computer science curriculum and is typically limited to 15 students, allowing for close faculty-student interaction often in the form of code reviews. The course reviews linear data structures while introducing more complex structures like trees, hash tables, sets, and graphs; teaching complexity analysis; establishing sound software engineering practices; familiarizing students with Git and other collaboration tools; and emphasizing thorough testing methodologies. Students enter with foundational programming knowledge from CS1 (Python) and CS2 (OO programming), but this course represents their first significant exposure to a statically-typed language outside of Python.

The transition from Java to C# was motivated by industry feedback and student employment prospects, as job descriptions increasingly prefer C# and .NET over Java. This presented an opportunity to redesign the course assignments while maintaining core learning objectives related to advanced data structures, algorithm analysis, and software engineering practices. Through carefully designed assignments, students learn modular design, separation of

concerns, and design patterns that improve code maintainability.

The philosophy of the course is centered on a semester-long progressive project in which students build components that join to form a cohesive whole. This contrasts with the more common approach of discrete, topic-specific assignments. Our model requires careful scaffolding, as assignments do not precisely mirror current course content. Instead, they gradually increase in difficulty and expectations to build comprehensive understanding.

The course blends theoretical foundations with practical implementation and testing methodologies. As students study algorithm development and complexity analysis, they simultaneously implement and test these concepts in working code, reinforcing abstract principles through concrete application. This integrated approach extends beyond individual components to the system as a whole—students develop unit tests for separate modules while tackling the challenges of integration testing as their components interact within the larger project framework. By experiencing both the theoretical reasoning behind efficient algorithm design and the practical realities of building interconnected software systems, students develop a comprehensive skill set that prepares them for advanced coursework and professional software development, where both conceptual understanding and implementation expertise are equally valued.

3 Identifying the Project and Sequencing

The LLMs. We focused on a few models rather than conduct a comparative analysis. This focused approach identified effective prompt strategies and established a reliable workflow for assignment creation. We primarily used Claude (Sonnet 3.7) with a paid "Max" account and occasionally used a paid Copilot account.

Project framework ideation. Our project ideation process with Claude required several iterations before reaching productive outcomes. Initially, Claude produced generic data structure implementations rather than grasping the full context of our project constraints. Early responses offered frameworks that often included overly ambitious technical requirements, insufficient scaffolding, or assumed content-based expertise beyond what our instructors possessed (e.g., more than a surface understanding of biological concepts). This highlighted a crucial ***lesson***: effective LLM collaboration resembles working with students—you must lead incrementally toward your goal rather than attempting to encode all requirements in one comprehensive prompt. Each step demanded its own iteration cycle. We found that trying to design a complete assignment workflow within a single prompt (what we initially thought of as a form of 'meta-design' or pedagogical meta-programming) was impractical. Results were mixed (projects with fewer assignments than requested) or somewhat

standard (e.g., a database).

Successful outcomes emerged through persistent guidance and constraint refinement across multiple exchanges—a more time-intensive but conversational approach that better aligned solutions with our educational objectives. Several ideas arose that were interesting as multidisciplinary: Ecological Monitoring and Conservation System, Urban Transit Optimization, and Disaster Response and Resource Management System. Although not the most engaging choice for students, we selected a simple compiler system based on our familiarity with it and its effectiveness in undergraduate teaching. This suggests we can build better scaffolds toward advanced frameworks in future courses, moving students from basics to sophisticated applications.

Assignment sequencing. For first-time C# learners, we established a structured progression through two foundational assignments: (1) introduction to IDE environment, unit testing, and core C# language constructs; and (2) object-oriented paradigms implemented through familiar data structures. Our early LLM ideation sessions consistently produced 'first assignment' concepts that were either too broad or assumed unrealistic C# proficiency. Another ***lesson*** emerged: effective assignment sequencing requires predetermined progression planning with granular understanding of each pedagogical step. LLMs proved valuable when properly scoped.

3.1 The DEC Compiler Project

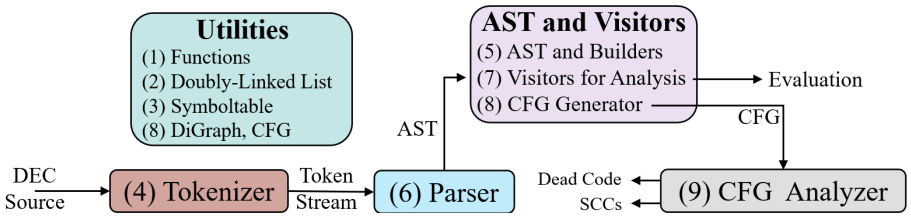

Figure 1: A quasi-dataflow figure describing the nine assignments.

The DEC (Declarative) language compiler project served as our course framework for applying data structures and algorithms in practical contexts. DEC features variables, arithmetic expressions, assignments, nested blocks, and return statements—simple enough for a semester project yet demonstrating key compiler and software engineering concepts. The compiler project progressed through nine interconnected assignments as shown in Figure 1.

1. **Utility Functions**: Students implemented basic utility functions in C# to gain familiarity with language syntax and testing frameworks.

2. **Doubly Linked List**: A generic class implemented with C# collection interfaces.
3. **Hierarchical Symbol Table**: A dictionary structure supporting nested variable scopes.
4. **Tokenizer**: A lexical analyzer converting source code to tokens.
5. **Abstract Syntax Tree (AST) and Builders**: The Builder pattern creates hierarchical program representations, reinforcing tree structures and enabling flexible AST construction.
6. **Parser**: A recursive-descent parser transforming tokens to an AST.
7. **AST Visitors**: Students implemented the Visitor pattern to create specialized visitors for unparsing, semantic analysis, and program evaluation while maintaining a clean AST hierarchy.
8. **Control Flow Graph (CFG) Generation**: Students created a directed graph representation of program execution paths using another AST visitor.
9. **Graph Analysis Algorithms**: Implementing BFS, DFS, and strongly connected components algorithms to analyze CFGs.

This progression introduced increasingly complex data structures (lists, trees, graphs) and algorithms (traversal, search, analysis) while building toward a partial implementation of a compiler.

4 Developing the Assignments

Our assignments have three essential components: source code, unit test code, and a description document with background information and requirements. We communicated these specifications to Claude as part of our prompt engineering strategy. We instructed the model to withhold artifact generation until requested, ensuring alignment with our pedagogical framework and assignment structure.

4.1 Source Code

For each assignment, we started by instructing Claude to generate source code based on our loose description of the current assignment, the overall system, and the sequence of assignments. To ensure accessibility of code constructs for students new to C#, we established specific code generation requirements: avoid lambda expressions when possible (if used, always provide explanatory comments); include comprehensive docstring comments for all methods detailing parameters, return types, and exceptions thrown; and use explicit datatypes instead of `var` in loops to enhance code clarity.

Table 1: A comparison of control flow patterns: LLM-generated sequential continue pattern (left) versus preferred traditional if-else structure (right).

```
while (condition) {
  if (condition1)
    { <case 1 code> continue; }
  if (condition2)
    { <case 2 code> continue; }
  if (condition3)
    { <case 3 code> continue; }
  <case 4 code>
}
```

```
while (condition) {
  if (condition1)
    { <case 1 code> }
  else if (condition2)
    { <case 2 code> }
  else if (condition3)
    { <case 3 code> }
  else { <case 4 code> }
}
```

A ***lesson*** we quickly learned was that LLMs excel at generating data structures, which is logical given the numerous implementations and the probability-based nature of LLMs. Thus, we completed several assignments (2, 3, 5, 9) quickly.

For traditional data structures and even non-traditional data structures, the code and implementations were generally reasonable for our course. However, we sometimes found that code generated by LLMs included constructs or patterns that either exceeded student capabilities or contradicted course norms. As an example consider Table 1, LLM-generated code often employed continue statements in nested conditions where a traditional if-else structure would be more appropriate for students. Despite providing explicit instructions, the LLM frequently generated code with `continue` statements. This behavior persisted across prompts, suggesting difficulty controlling stylistic consistency without structural scaffolding. Our solution was to provide a structured template (see Table 1) for the LLM to follow. This reveals another ***lesson***: for certain programming constructs, we must provide explicit structural guidance to LLMs, much as we would for students learning proper coding practices.

The LLM also occasionally employed language features or design patterns not covered in the course, requiring instructor intervention. A significant implementation challenge encountered during our project was the development of an appropriate parser for the DEC language. Despite designing a grammar that could be unambiguously represented in Backus-Naur Form without left recursion—a deliberate choice given that our course emphasizes data structures and implementation methodology rather than language complexity—we faced persistent difficulties with automated code generation.

The large language model consistently produced parser implementations that employed a loop-based and count-based, greedy matching strategy, wherein each opening delimiter (such as 'begin' or '{') would be matched with its corresponding closing delimiter. This approach persisted despite explicit instruc-

tions to generate a recursive descent parser with hierarchical helper methods structured to process distinct components of the token stream. In the end, we implemented a recursive descent parser ourselves.

The recursive complexity of the AST structure exceeded the model's ability to generate appropriate parsing logic, suggesting current language models struggle with deeply nested recursive programming constructs. It is thus a ***lesson*** that there may be instances in which AI assistance requires supplementary human expertise. In the end, we implemented a recursive descent parser ourselves.

Justification. Generated code provides a consistent metric for gauging assignment difficulty and accessibility for students. It establishes a baseline complexity level that can be adjusted to match student capabilities, ensuring assignments are appropriately challenging without being overwhelming. While this approach may make students more likely to use LLMs themselves, our firsthand experience helps us identify characteristic structures and patterns typically employed by these tools. Though this advantage may diminish as LLMs evolve, the pedagogical insights gained remain valuable.

4.2 Unit Tests and Description Document

Additionally, we prompted the model to "Generate corresponding xUnit tests for each implemented method, with test classes named according to the class being tested. Use `[Theory]`, when applicable, to mitigate redundancy of test code." (The `[Theory]` attribute in xUnit helps verify code behavior across multiple data sets using `[InlineData]` to parameterize test cases.) Unit tests generated by LLMs, while comprehensive in many respects, sometimes failed to compensate for logic in string-based returns (often from `ToString`). In one instance, the LLM did not compensate for rational numbers being output as integers (i.e., 3/1 is output as 3). In two assignments, we had to manually configure some tests; this level of performance was satisfactory and exceeded our expectations given the complexity of the test cases.

To ensure that the LLM generates edge cases, we found again that iteration is key. A ***lesson*** we learned involved prompting the LLM with reflective questions such as 'Did I miss any test cases?' immediately following initial test generation, which consistently led to improved coverage.

Once the parser assignment was complete, we could engage in more thorough integration testing. We first ensured all new and existing unit tests passed successfully. We then instructed the LLM to generate integration tests specifically designed to parse input programs into ASTs and perform any subsequent processing. This approach verified parser capabilities within the system while maintaining standards across test types.

Description document. After completing satisfactory source code and

test suites, we instructed the LLM to generate our assignment description document. To maintain consistency with course standards, we provided three exemplar documents for reference. Our requirements specified that Claude should: incorporate illustrative examples to clarify assignment concepts; adhere to our established document structure with standard section headings ("Background," "What You Need To Do," "Testing", etc.); and present method specifications in tabular format detailing method names, input/output parameters, and functional descriptions.

Each of our description documents typically spans 3-4 pages, containing comprehensive background information, illustrative examples, and code-related instructions. The structure of these assignments constrains the source code interfaces while leaving implementation details to student discretion.

We believe that description documents are not just about code instruction; philosophically, they act as sections in a textbook. Reading the description several times for comprehension is an integral part of the learning process. This multilayered approach to documentation supports students in developing both technical implementation skills and deeper conceptual understanding.

We quickly discovered that LLMs struggled to develop description documents consistent with our course's established style, despite being provided with sample documents. The AI-generated content offered minimal explanation of topics, overly succinct examples, and exhibited a structural preference for bulleted lists rather than the paragraph-form narrative we typically employ. Particularly in the DEC assignment sequence, Claude failed to provide sufficient background information in a manner accessible to our student audience.

The fundamental ***lesson*** we learned: LLMs work best with interactive, small-scale requests rather than generating entire assignments at once. This micro-approach produced better results that matched our style, standards, and expectations.

5 Recommendations

Based on our experiences with LLM-assisted assignment development, we offer these recommendations to educators looking to effectively leverage these tools. Our experiences revealed that LLMs can accelerate content creation, but successful implementation requires strategic guidance and subject expertise.

1. **Limit prompt complexity and iterate rather than dictate**: Overly elaborate prompts often produced suboptimal results. A more effective approach was to begin with simpler requests and iteratively refine them based on the LLM responses. Adopt an iterative approach in which constraints are gradually introduced and refined. Overall, do not expect to generate complete assignments from a macro-perspective.

2. **Review for pedagogical alignment**: The generated code sometimes defied course norms or included constructs beyond the students' capabilities. We found offering pattern-based alternatives (e.g., "convert to use `if-else` instead of `continue`") effective.
3. **Verify unit tests**: After having an LLM generate unit tests, explicitly ask it to identify any missing test cases. This simple follow-up consistently revealed overlooked scenarios and improved test coverage.
4. **Guide pattern implementation**: While LLMs effectively generate code implementing design patterns, instructors should dictate which patterns to use and how they apply to align with course objectives.
5. **Provide system context**: We achieved better results by establishing overall system context (our DEC project) and then requesting specific components rather than asking for standalone assignments.
6. **Maintain technical oversight**: Double-check all generated code and tests for technical accuracy, especially when implementing specialized algorithms or data structures.

6 Related Works

Research on LLMs in computing education has primarily focused on student-facing tools for programming assistance, with limited exploration of instructor-led content creation workflows. Our work addresses this gap by examining how educators can leverage LLMs for progressive assignment design.

Most of the existing work examines the student use of LLMs. CodeHelp provides LLM-powered programming assistance with instructor oversight [3], while Zheng et al. demonstrate that scaffolded LLM collaboration improves students' computational thinking [6]. However, pedagogical concerns persist. Wu et al. report negative correlations between LLM use and learning outcomes [5], and Herman argues that AI tools should not replace fundamental conceptual understanding [1]. These findings support our approach of using LLMs for assignment creation while restricting student access during coursework.

Closer to our work, Hoq et al. develop a collaborative framework where instructors co-create programming problems with LLMs through iterative refinement [2]—mirroring our conversational approach. Their misconception-driven approach parallels our discovery that structured, step-by-step prompting yields better LLM results than attempting comprehensive single interactions.

Although Sharma et al.'s systematic review highlights LLMs' potential for reducing instructor workload, they emphasize that benefits require careful pedagogical alignment and domain expertise [4]—precisely what our iterative refinement process provides. Despite growing interest, no existing work system-

atically examines instructor workflows for creating cohesive, multi-assignment projects using LLMs. Our contribution fills this gap by documenting practical strategies for prompt engineering, iterative development, and pedagogical alignment in a real-world course redesign context.

7 Conclusions

Our exploration of LLMs for programming assignment development reveals both promising capabilities and important limitations. LLMs significantly accelerated our course transformation from Java to C# and the development of our nine-part compiler project, though with varying effectiveness across components. Our experience suggests that LLMs currently excel at producing standard data structure implementations and testing suites but struggle with generating deeply recursive or semantically nuanced code without substantial instructor oversight. Recognizing these boundaries helped us design better prompts and development workflows.

The most valuable insight from our work is that effective LLM collaboration requires conversation rather than commands. Successful assignment development emerged through gradual improvements across multiple exchanges, like how instructors might guide students through complex topics. This iterative approach provided proper support while staying aligned with course goals.

While these redesigned assignments await classroom implementation, this report documents our complete development process and lessons learned from the creation of LLM-assisted assignments. Future course offerings will incorporate and evaluate these materials. We offer this as a foundational guide for educators who pursue similar LLM-assisted design workflows.

LLM-assisted assignment development represents a valuable addition to educational content creation when approached with appropriate expectations, subject matter expertise, and systematic interaction. By understanding both the capabilities and limitations of these tools, educators can leverage them to create more robust, engaging, and pedagogically sound programming assignments.

References

[1] Felienne Hermans. "Why to use LLMs in computer science education?" In: *Personal blog* (2025). URL: `https://www.felienne.com/archives/8367`.

[2] Muntasir Hoq et al. *Facilitating Instructors–LLM Collaboration for Problem Design in Introductory Programming Classrooms*. arXiv preprint. 2024. arXiv: `2504.01259`. URL: `https://arxiv.org/abs/2504.01259`.

[3] Mark Liffiton et al. "CodeHelp: Using Large Language Models with Guardrails for Scalable Support in Programming Classes". In: *Proceedings of the 2023 Koli Calling International Conference on Computing Education Research*. ACM, 2023. DOI: 10.1145/3631802.3631830. URL: https://dl.acm.org/doi/10.1145/3631802.3631830.

[4] R. Sharma et al. "Use of AI-Driven Code Generation Models in Teaching and Learning Programming: A Systematic Review". In: *Proceedings of the 2024 ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 2024. DOI: 10.1145/3626252.3630958. URL: https://dl.acm.org/doi/10.1145/3626252.3630958.

[5] Qianou Wu et al. "The Impact of Large Language Models on Programming Education and Student Learning Outcomes". In: *Applied Sciences* (2024). URL: https://www.mdpi.com/2076-3417/14/10/4115.

[6] Lu Zheng et al. "LLM-based collaborative programming: impact on students' computational thinking and self-efficacy". In: *Humanities and Social Sciences Communications* 12.149 (2025). URL: https://www.nature.com/articles/s41599-025-04471-1.