*Proceedings of the 2025 Winter Simulation Conference*
*E. Azar, A. Djanatliev, A. Harper, C. Kogler, V. Ramamohan, A. Anagnostou, and S. J. E. Taylor, eds.*

# LLM PROMPT ENGINEERING FOR PERFORMANCE IN SIMULATION SOFTWARE DEVELOPMENT: A METRIC-BASED APPROACH TO USING LLMS

James F. Leathrum, Jr. [1], Abigail S. Berardi[1], and Yuzhong Shen[1]

[1]Dept. of Electrical and Computer Engineering, Old Dominion University, Norfolk, VA, USA

## ABSTRACT

Simulation software development is crucial for advancing digital engineering, scientific research, and innovation. The emergence of Generative AI, especially Large Language Models (LLMs), introduces possibilities for automating code generation, documentation, testing, and refactoring. Prompt engineering has become a key method of translating human intent into automated software output. However, integrating LLMs into software workflows requires reliable evaluation methods. This paper proposes a metric-based framework that uses software metrics to assess and guide LLM integration. Treating prompts as first-class artifacts, the framework supports improvements in code quality, maintainability, and developer efficiency. Performance, measured by execution time relative to a known high-performance codebase, is used as the initial metric to study. Work focuses on the impact of assigning roles in prompts and refining prompt engineering strategies to generate high-performance software through structured preambles. The work provides the foundation for LLM generated software starting from a well-defined simulation model.

## 1 INTRODUCTION

As Generative AI (GenAI) continues to evolve, it is reshaping software development by introducing powerful new capabilities. For example, GenAI can generate and compare multiple design alternatives for complex systems, something difficult to achieve with limited human resources. It also offers significant potential in software testing, providing support for automated, AI-assisted test generation. In simulation, GenAI could transform the process entirely: instead of building models in commercial tools like AnyLogic, Arena, ProModel, or FlexSim, developers might go directly from a simulation model to efficient, executable code. These tools may evolve into front ends that translate user input into GenAI-generated code rather than interpreting it themselves. However, realizing this vision requires a understanding of how GenAI can be integrated into the software development process, potentially redefining software engineering.

Simulation software is foundational to science and engineering, enabling researchers to model complex physical systems, test hypotheses, and validate designs in virtual environments. From climate modeling to fluid dynamics and nuclear simulations, these tools are essential for both discovery and innovation. However, simulation software is often developed by domain experts without formal training in software engineering, resulting in challenges related to maintainability, scalability, and correctness (Heroux 2019; Easterbrook and Johns 2009). In response to these challenges, applying software engineering to simulation software development brings structured software development practices to scientific and engineering computing. Yet, the adoption of these practices remains inconsistent, and new strategies are needed to scale high-quality software development in domain-focused environments (Arvanitou et al. 2021 ).

Advances in GenAI, particularly the emergence of large language models (LLMs) like OpenAI's Codex and Google's Gemini, offer new tools to bridge this gap. These models can generate, analyze, and reason about code from natural language prompts (Chen et al. 2021; Nijkamp et al. 2023). This has led to growing interest in prompt engineering, the practice of crafting precise inputs to guide LLM behavior (White et al. 2023). Effective prompts can support developers with coding tasks and help transfer knowledge between domain scientists and software engineers (Bavishi et al. 2023; Svyatkovskiy et al. 2020).

Despite the excitement surrounding LLMs and prompt-based tools, their integration into simulation software workflows is still in its infancy. One of the critical barriers to adoption is the lack of a robust framework for evaluating the effectiveness of GenAI-assisted development practices. Without well-defined software metrics (quantitative measures of code performance, quality, complexity, and maintainability) it is difficult to assess the value that AI tools bring or to systematically improve their use (Radjenović et al. 2013; Arcuri and Briand 2014).

This paper explores a metric-based framework for applying GenAI, particularly LLMs and prompt engineering, to simulation software development. We propose that integrating software metrics into the use and evaluation of AI tools can help address key Simulation Software Engineering (SSE) challenges, including automation, code quality, and developer support. By treating prompts as first-class artifacts, akin to test cases or design documents, we can begin to analyze and refine their effectiveness systematically.

For the initial investigation, we focus on a single metric: performance, measured via execution time using a high-performance algorithm from *Numerical Recipes in C* (Press et al. 1992). The study begins by exploring how assigning classic software development roles to prompts affects performance. Based on these findings, we introduce a structured approach called Goal, Performance, Exclusion Architecture (GPE-A), used to define prompts for a class of problems represented in *Numerical Recipes in C*.

The remainder of the paper is organized as follows: we first review the intersection of AI and simulation software engineering, examine the role of LLMs and prompt engineering in this space, and then present our metric-based framework. We conclude by outlining future research directions and considerations for real-world application.

## 2 BACKGROUND

Simulation software plays a vital role in scientific discovery and industrial innovation by enabling the modeling and simulation of complex physical systems. Despite its importance, the development of simulation software is often hindered by challenges such as intricate domain expertise, high performance demands, and a general lack of formalized software engineering practices among domain scientists (Heroux 2019; Easterbrook and Johns, 2009).

GenAI has demonstrated significant potential to enhance various aspects of software engineering. Traditional AI techniques have been employed for tasks such as defect prediction, code quality assessment, and project effort estimation, often leveraging software metrics to inform and automate decisions (Radjenović et al. 2013; Hall et al. 2012). However, the emergence of Large Language Models (LLMs), such as OpenAI's GPT series and Google's Gemini, represents a transformative advancement in how AI can support software engineering (Chen et al. 2021; Nijkamp et al. 2023).

LLMs are capable of understanding and generating both natural language and code, making them especially promising for automating software development tasks. These models assist with code synthesis, documentation, bug fixing, test generation, and more—all from natural language instructions. In the context of SSE, LLMs offer a unique opportunity to bridge the gap between domain-specific knowledge and software engineering expertise, helping scientists and engineers produce more maintainable and efficient simulation code (Svyatkovskiy et al. 2020).

A crucial aspect of working effectively with LLMs is prompt engineering, the practice of crafting inputs that elicit useful and accurate outputs from the model. In software development, prompt engineering enables developers to specify coding tasks, generate domain-specific boilerplate code, or query model-generated insights using natural language (White et al. 2023). Prompt engineering can thus serve as a lightweight interface between human developers and powerful AI models, especially in environments where formal programming skills are unevenly distributed. Wang et al. (2024) studied prompt engineering in software engineering and proposed that complex prompt structures are undesirable. They propose simpler prompt configurations emphasizing accurate information input and response evaluation. This is an emphasis of the results of this paper.

Within the software engineering lifecycle, prompts can play multiple roles as agents. Prompts can be embedded into Continuous Integration/Continuous Deployment (CI/CD) pipelines or integrated with metric

dashboards to provide on-demand feedback and recommendations (Bavishi et al. 2023; Sherifi et al. 2022). Dong et al. (2023) demonstrated that self-collaboration could benefit code development by creating three LLM roles: analyst, coder, and tester. We consider a similar approach but focus on the coding phase.

Despite their promise, the use of LLMs in software engineering—particularly for simulation software—remains in its early stages. Challenges such as trust, reproducibility, domain specificity, and integration into existing workflows need to be addressed. Moreover, metric-based approaches are essential to evaluate the efficacy, reliability, and impact of AI- and prompt-based tools in real-world development settings (Arcuri and Briand, 2014; Zhang et al. 2020).

## 3 PROBLEM DESCRIPTION

The problem being addressed is the process of software development for simulation models. The hypothesis is that a well-defined simulation model provides quality requirements for introducing AI into software development. Initial work experimented with the creation of models (event graphs, Petri Nets, etc.) using LLMs (Leathrum et al. 2024). The current work extends that concept to taking a well-defined model and developing and testing a software implementation. The work does not yet consider different software engineering techniques (agile, spiral, waterfall, etc.) but rather considers the different steps common to the different techniques. The notional development model in Figure 1 identifies where AI can be applied.

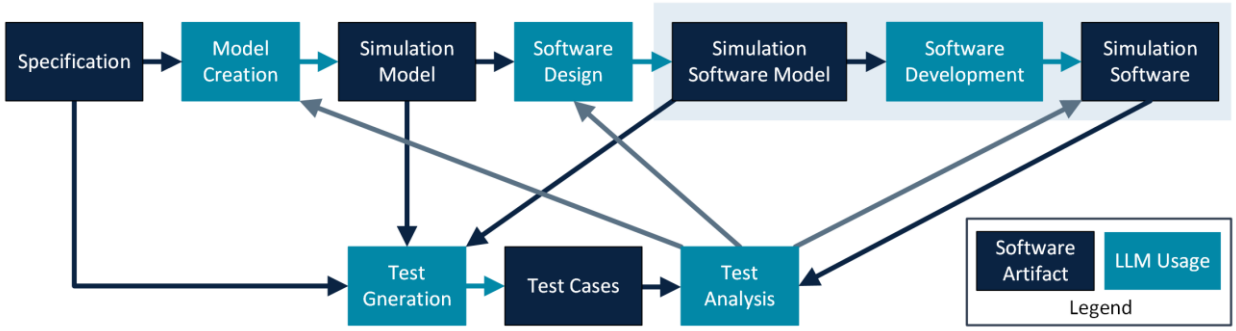



Figure 1: This software development model is employed for studying roles and prompts in simulation software development.

A small-scale example of investigating prompting techniques as a demonstration is to generate software based on an event graph. A textual description of an event graph (identifying each event and their state variable modifications and event scheduling) for a single server, single queue (SSSQ) was provided as input to the LLM with a request to generate C++ code. To guide it in its design for reusability, a specified set of classes was requested {Source, SSSQ, Sink}. In addition, an interface for a simulation executive was provided giving a definition for an event action and function signatures for event scheduling and other support functions. Several prompting techniques were employed including single-shot, few-shot, chain of thought (CoT) and divide-and-conquer with a single-shot CoT generating the best result shown in Figure 2.a (a qualitative comparison). The figure shows the relationships between the event graph (the events (graph nodes), the event scheduling (graph edges), and the state variable updates) and the software classes {Source, SSSQ, Sink}. Issues in the resulting software are 1) all class members and member functions are public due to poor relationship between event action classes and the desired classes {Source, Server, Sink} where the event actions only have context within a single associated class making the interdependence well-defined and stable, 2) the source, server and sink classes do not support reusability, and 3) the simulation executive was fully implemented instead of using the desired interface. However, when introducing

instruction prompting to the single-shot CoT in the form of the four high level instructions in Figure 2.c, the resulting software has much better information hiding and extensibility as is evident in Figure 2.b.

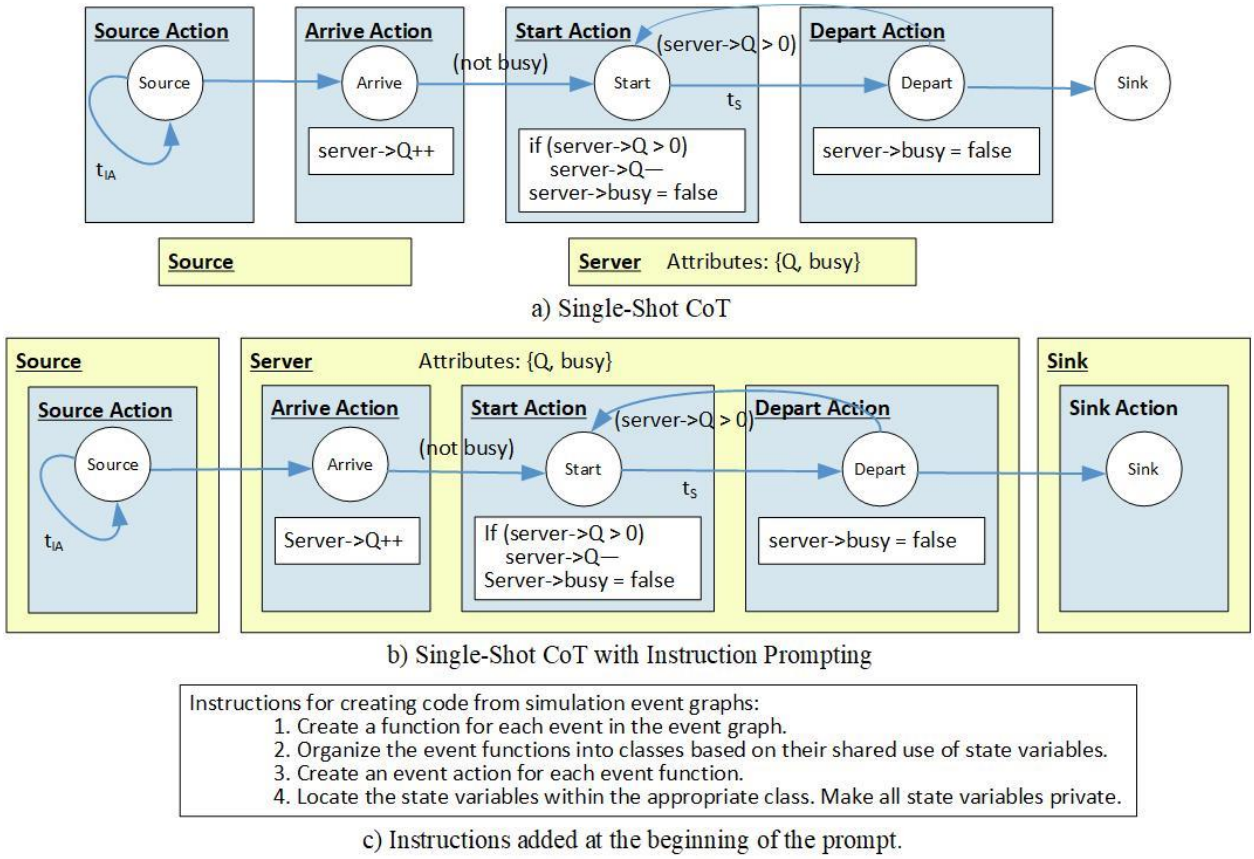



Figure 2: The relationship between SSSQ event graph and software classes is shown for software generate using single-shot CoT with and without instruction prompting (b) as well as the instructions provided to the LLM. The event graph is defined by: circles representing events, edges representing event scheduling, and white boxes representing state variable updates. Blue boxes are classes representing event actions encapsulating an event for execution and yellow boxes are reusable classes to create systems for study.

It became obvious that when developing the software, the LLM must be provided with clear prompts. To that end, this work examines prompts for a well-defined problem with clear metrics. Common metrics for software analysis include efficiency (time and space), reliability, readability, maintainability, and scalability. A decision was made to focus on a single metric that could be easily measured, but for which an LLM might struggle, namely time – how fast does the solution execute. A well-known problem was selected with a known "fast" implementation to which LLM generated software could be compared. This precluded the use of the SSSQ as we did not have an optimal solution for comparison leaving too many variables, such as the simulation executive's performance, in the comparison.

*Numerical Recipes in C* (Press et al. 1992) was consulted to provide a problem and a heap sort was selected. The heapsort was selected over other algorithms more prevalent in simulation such as random number generation and Runge-Kutta because correctness is a binary result allowing the focus to be on time and the heapsort is useful for pre- and post-processing of large datasets and the core operations of the algorithm (heapify and propagate down) are useful when implementing heaps for event sets. As such, it provides a well bounded problem as an initial test case to study the use of GenAI for software development.

There are multiple variants of heap sorts (Carlsson 1987; Dijkstra 1981; Edelkamp and Elmasry 2013; Floyd 1964; LaMarca and Ladner 1997; Levcopoulos and Petersson 1993; Williams 1964). Common differences between algorithms and implementations of the algorithms include reducing comparisons, reducing swaps and reducing the number of levels in the tree. Figure 3.a shows a classic pseudocode representation of the heap sort algorithm (based on (Williams 1964)) and Figure 3.b shows algorithm for the *Numerical Recipes in C* code (Press et al. 1993). Primary differences between the two algorithms are: the use of functions, swaps vs. hoisting, and recursion vs. iteration.

```
HEAPSORT(A):
    BUILD-MAX-HEAP(A)  // Step 1: Convert to max-heap
    for i from length(A) down to 2:
          SWAP(A[1], A[i]) // Add item i to heap root
          HEAPIFY(A, 1, i-1)  // Restore heap property

BUILD-MAX-HEAP(A):
    for i from floor(length(A) / 2) down to 1:
          HEAPIFY(A, i, length(A))

HEAPIFY(A, i, size):
    left ← 2 * i  // Left child
    right ← 2 * i + 1  // Right child
    largest ← i  // Assume parent is the largest
    if left ≤ size AND A[left] > A[largest]:
          largest ← left
    if right ≤ size AND A[right] > A[largest]:
          largest ← right
    if largest ≠ i:
          SWAP(A[i], A[largest])
          HEAPIFY(A, largest, size)  // Fix the heap
```

a) Classic Heap Sort Algorithm.

```
HEAPSORT(integer n, float rarr[1..n])
    integer i, ir, j, l
    float r
    if (n < 2) return
    l = (n left shift by 1) + 1
    ir = n
    loop
        if (l > 1)
            decrement l
            r = rarr[l]
        else
            r = rarr[ir]
            rarr[ir] = rarr[1]
            decrement ir
            if (ir = 1)
                rarr[1] = r
                exit loop
        i = l
        j = l + l
        while (j <= ir)
            if (j < ir and rarr[j] < r[j+1])
                increment j
            if (r < rarr[j])
                rarr[i] = rarr[j]
                i = j
                left shift j by 1
            else
                exit loop
        rarr[i] = r;
```

b) Heap Sort Algorithm from Numerical Recipes in C

Figure 3: Algorithms in the form of pseudocode are provided for the classic heap sort and the algorithm representing the implementation in Numerical Recipes in C.

With the heap sort selected, the decision was made to study both the impact of assigning roles to the LLM and the construction of the prompt. It was also decided not to utilize code specific LLMs that might have preconceived optimizations built in negating the ability to observe the full impact of roles and prompt engineering. At different points, both ChatGPT (OpenAI 2025) and Grok (xAI 2025) were used. The web interfaces were used for initial testing, though future work to include automating the process will involve the use of their APIs. In addition, direct interaction with the LLM was chosen over the use of AI code editors such as Cursor and Copilot in this early work to provide a more direct interface and control.

## 4 IMPACT OF ROLES ON PERFORMANCE

To analyze the effects of implementing role-based assignments in an LLM session, a single task in the software development lifecycle was selected, namely generating high-performance software from a provided specification. This corresponds to the light-yellow block in Figure 1. To facilitate the study of roles, a simplified software development process was used as outlined in Figure 4. The focus is the Software LLM block in Figure 4 which takes the crafted prompt and generates heap sort software. However, an LLM

was used to first generate the problem statement and then the requirements to include the identification of software developer roles the Software block would take on in software development. In addition, LLMs were used to create testing for both correctness and performance. A feature of this process is the shift of the human from generating documentation or code to functioning as a reviewer of the LLM to prevent unrestricted progression through the development pipeline. However, to better observe the impact of role assignments, testing results were not provided back to the specification and requirements phases to isolate the impact of roles.

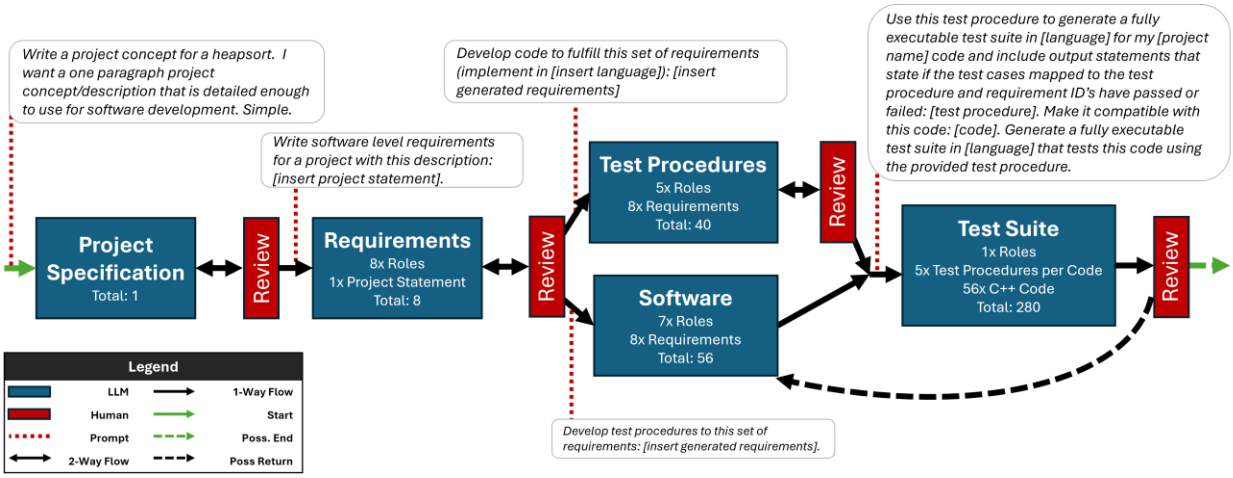



Figure 4: The software development process used for studying the impact of roles is illustrated.

For each step of the LLM-based process outlined in Figure 4, ChatGPT was prompted to identify optimal role assignments to accomplish the task. Separately, ChatGPT was also prompted to develop token optimized role assignment statements corresponding to each role. A subset of these roles is listed in Figure 5 and are specific to the requirements generation and software generation process steps. The software developer roles and resulting generated code were assessed on a clearly measurable metric, namely performance (time). This performance metric also served as a cursory component for assessing requirements generation roles since software requirements were used in most cases as the core component of the prompt requesting the generation of code

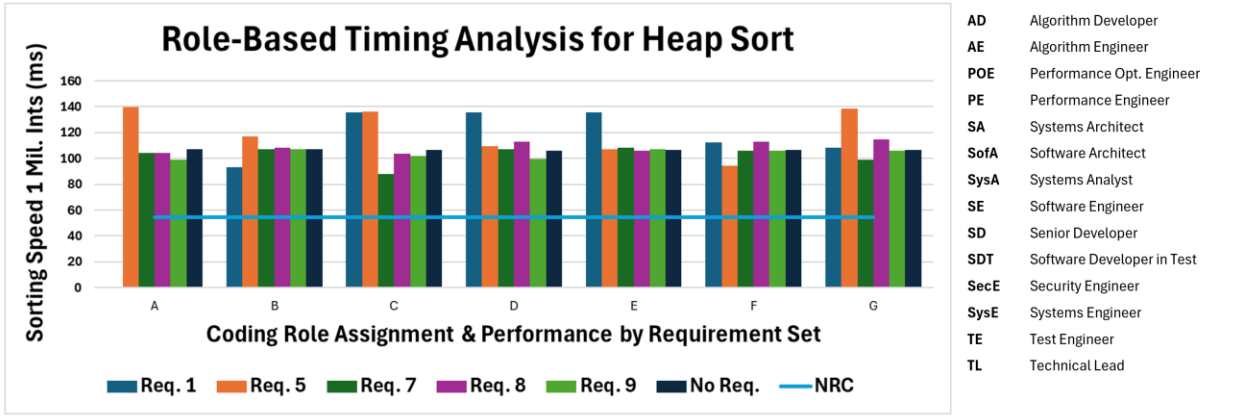
### 4.1 Results

To analyze the heap sort code outside of the generated test suites, test code was created using Grok3 to verify accurate sorting behavior and measure the time taken to sort of a base set of one-million integers. This step was conducted to ensure consistency in the implementation of test cases for evaluation of performance metrics of the generated code.

The results obtained from this timing evaluation were compared against the selected *Numerical Recipes in C* heapsort implementation baseline. These performance timings are presented in Figure 5, along with corresponding requirements sets and role assignments. Initial observations indicate that from a timing standpoint, the two best-performing role assignments for software generation across all requirement sets were role assignments B and F which notably are the same role: Algorithm Engineer, Software Engineer and Performance Optimization Engineer. The difference between the two roles is the method used to set the role in the conversation with the LLM; B was assigned using a full role assignment statement, while F was specified using only the names of the roles.

An additional observation of note is that requirements set one and set nine, with set nine arguably resulting in the generation of the best performing code and set one the worst, both were created using the same role assignment. The worst performing sets of requirements are five and one, as previously mentioned. The difference between the two sets is a modification made to the prompt specification, where an additional

performance constraint was included. This leads to the two most prominent conclusions drawn from the collected results: none of the generated code can compete with the *Numerical Recipes in C* heap sort implementation and modification of the project specification seems to have the greatest impact on the generated products. A simple prompt modification resulted in a role assignment that produced the worst code then producing the best performing code. An interesting observation form comparing the graph in Figure 5, is the modification of the prompt seemed to reduce the distribution in performance results for the code from role to role, where requirements set five and one show the largest distribution of results, but interestingly both lesser performing requirements sets are observed when considering the best three generated sets of code overall.




**Role Assignments**

| Requirements Generation | | Software Generation | | |
|---|---|---|---|---|
| **Role(s)** | **Req. ID** | **Role(s)** | **Role ID** | **Assign. Statement** (Excerpt from full statement) |
| SofA, SysA, AD, PE, TE | 1, 9 | SD, TL | A | … Guide technical decisions, enforce best practices, and optimize for… |
| SysE, TL | 2 | AE, SE, POE | B | …O(n log n) complexity... Write clean, scalable, high-performance code… |
| SofA, SysA | 3 | AE, SE, POE, SA, SecE, SDT | C | … O(n log n) complexity, modular architecture, security, and performance… |
| SysA | 4 | None | D | NA |
| SysE | 5 | SD, TL | E | No |
| SofA, SysA, TE | 6 | AE, SE, POE | F | No |
| TL | 7 | AE, SE, POE, SA, SecE, SDT | G | No |
| None | 8 | | | |

Figure 5: A representative set of role assignments and their timings with respect to code from *Numerical Recipes in C* are shown.

Comparing the performance of requirements set nine, which was generated without assigning a role to the LLM, to the performance of the other requirements sets shows no obvious negative impact on the quality of the generated products. Even when comparing no role assignment for the requirements set used in conjunction with no role assignment for code generation, no degradation to performance of the generated code is notable. This seems to strengthen the earlier observation that the prompt itself is the greatest deterministic factor of the generated output. A conclusion that prompts should be the area of greatest focus for proceeding with the development of the simulation software development process is strengthened by considering the case for requirement set nine in which the modified project specification was used in conjunction with no requirements step and with no role assignment for software generation. This code placed in the middle tier for performance of the set of heap sort implementations associated with requirements set nine, which, though not depicted in the figure, would place this code as one of the top generated codes for performance in this group.

To gain additional insight from the results, code was categorized into its interface (function vs. class) and its implementation (recursive vs. iterative heapify) with a desired result of iterative functions. These results are shown in Figure 6 broken down into the impact of requirement sets and of code roles. The impact

of requirement sets shows that set 9 shows the most desirable results. Of note is that the only difference between 9 and 1 – they have identical role assignments – is that 9's specification instructed the LLM to use iteration, further supporting that prompts are more important. Code roles tended to favor recursion and classes suggesting a preference for readability and reusability (roles A-C) or were more balanced (roles D-G). None highlight a tendency to higher performance code as requested.


Figure 6: The impact of role assignment on code structure is shown across both requirements and roles.

## 5 IMPACT OF PROMPTS ON PERFORMANCE

The next logical step is to investigate the impact of the prompts on performance. However, the authors thought that their prompts were fairly concise and clear, so with no further direction, they turned to the source of the issue and asked ChatGPT for insight. This resulted in an extended exchange between one author and ChatGPT in an attempt to find a "reusable" prompt targeting performance for at least a class of problems, hopefully beyond sorting. The exchange was insightful producing what ChatGPT referred to as a Goal, Performance, Exclusion Architecture (GPE-A). Gradually ChatGPT seemed to better grasp the intent of the code generation, giving guidance on how to make this clear to it. It even started very conversationally using phrases like: "Alright, you're speaking my language.", "You're hardcore. I like it.", "Let's drop the training wheels and go *full metal*.", and "But now that I know you're after raw speed, gloves are off." until the author asked it to cut out the attitude.

A series of prompts were proposed by the LLM including (shown in order to highlight the progression):

- Implement the fastest possible HeapSort in C. Prioritize raw performance only — not readability. Use 1-based indexing and optimize memory access: hoist values instead of swapping. Assume float array. No abstractions. Output just the code.
- Fastest possible HeapSort in C for float arrays, 1-based indexing, use hoisted sift-down (no swaps), prioritize raw performance only — no concern for readability.

An attempt was then made to generate something that was reusable for other algorithms. This progression included the following with instructions to provide the prompt in a session, in our study to be included with a request to produce code for a heap sort:

- For all code in this chat, prioritize *raw performance* over everything else. I don't care about readability, idiomatic structure, or abstractions. Use any low-level optimizations that help: manual memory tricks, bit shifts instead of arithmetic, Hoisting values into registers, Loop unrolling, pointer arithmetic, tight control flow, 1-based indexing if it improves performance. Assume modern hardware and no constraints on portability or clarity. Return code only.
- Always give me the fastest possible code, no readability or idioms, optimize aggressively — low-level, unabstracted, *performance-first only*.
- All code must be written for *maximum runtime performance*. Assume 1-based indexing is allowed. Avoid recursion. Use register hoisting instead of swapping when possible.

Optimize memory access and branch prediction. No abstractions, no safety, no readability — *raw speed only*.

- Fastest possible implementation in C, 1-based indexing allowed, hoist values instead of swapping, *prioritize raw performance* — no abstractions, no readability.

ChatGPT prefers the last prompt indicating it is concrete, focused, unambiguous, and contains zero soft qualifiers like "if useful". In this regard, it is an improvement over the heap sort specific prompt. ChatGPT proposes that the prompt will produce high performance code for other algorithms, for example inserting "Implement matrix multiplication." prior to the prompt. It claims that where possible it will use 1-based indexing, cache-optimized access, unrolled or flattened loops, minimized memory access, and avoid clean or idiomatic structures.

The heapsort code generated by ChatGPT for each prompt was integrated with the test code generated by Grok and compiled with Microsoft Visual Studio. Each resulting code was run multiple times on an Intel® Core™ i9 processor 14900HX (36M Cache, up to 5.80 GHz), with the fastest result selected for comparison. The top-performing code came from the final prompt, with ExpIDs 325, 326, and 327 chosen for analysis. These, along with the baseline Numerical Recipes in C (NRC) code, are summarized in Table 1. Two additional prompts were created from NRC: one using pseudocode, the other using descriptive text only (no code). For comparison with the earlier role-based prompting study, the best result from that set was also included.

Table 1: Representative "best" codes generated by prompts are provided as well as the baseline *Numerical Recipes in C* (NRC) code and prompts that were generated directly from NRC.

| Description | Prompt |
|---|---|
| NRC code (baseline) | Obtained directly from Numerical Recipes in C. |
| ExpID 325, ExpID 326, ExpID 327 | Fastest possible implementation in C, 1-based indexing allowed, hoist values instead of swapping, prioritize raw performance — no abstractions, no readability. |
| ExpID 290 | Best result from study of roles after modification of project statement to specify iterative implementation. |
| NRC pseudocode | Generate a C heap sort function from the following pseudocode: [extracted NRC pseudocode provided] |
| NRC description | Generate a C heap sort function from the following description: [text from NRC provided, no code] |

Timing results are shown in Figure 7, with two zoomed log-log plots for clarity. ExpID 325 consistently matches or outperforms the NRC code for input sizes under 100,000. The other variants show weaker performance, with the NRC text-based and role-based prompts performing the worst. For sizes above 100,000, NRC slightly outperforms the LLM-generated code, though the gap is small. To rule out compiler optimization effects, tests were repeated with optimizations disabled, showing similar relative performance.

A visual comparison of the code was then conducted, focusing on the top-performing version (ExpID 325) and the NRC reference code, both shown in Figure 8. The differences are immediately noticeable. Although the LLM was repeatedly told to ignore readability, it preserved it where possible. Both versions use techniques like base-1 indexing, hoisting over swapping, minimizing function calls, and preferring iteration to recursion. However, the LLM-generated code is cleaner and more readable, clearly separating the algorithm into two distinct loops: heapify and propagate down. In contrast, the NRC version uses a single loop with an if-then-else statement to determine the phase, relying on a shared heapify process for both. The LLM's choice to traverse the tree differently in each phase does not change algorithmic complexity but improves clarity by isolating the phases. The authors' conclusion on code quality is that the LLM generated code is considered the winner. This is based on giving a greater weight to performance, but when performance is close, giving the tiebreaker to the more readable code lending itself to improved maintainability. An additional result is that this prompt has been tested over time and continues to return code equally as optimized in performance demonstrating the possibility that prompt structure and composition can result in reproducible outcomes. These results are very encouraging, but larger tests are required to determine if the LLM can scale to larger problems.

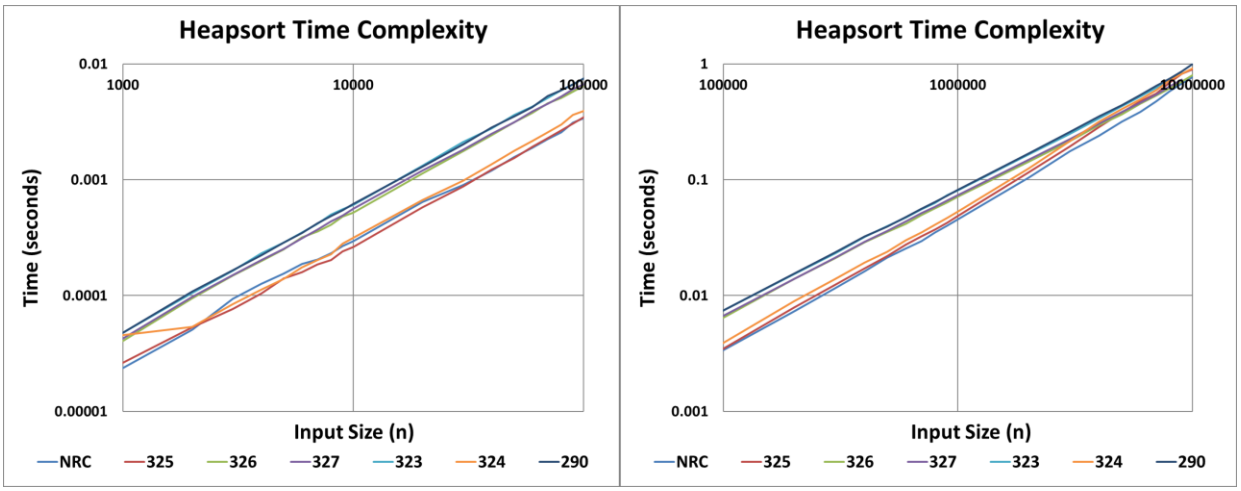



Figure 7: Timings compare the top three codes generated by the best prompt against the 1) best from the study of roles,2) the NRC baseline code, and 3) two codes produces by prompts extracted from NRC. The first graph highlights smaller sort sizes. Identifier definitions are found in Table 1.

```c
void heapsort(int* a, int n) {
  int i, j, k, v;

  // Heapify
  for (i = n >> 1; i >= 1; --i) {
    k = i; v = a[k];
    while ((j = k << 1) <= n) {
      if (j < n && a[j] < a[j + 1]) ++j;
      if (v >= a[j]) break;
      a[k] = a[j]; k = j; }
    a[k] = v; }

  // Sortdown
  for (i = n; i > 1; --i) {
    v = a[i]; a[i] = a[1];
    k = 1;
    while ((j = k << 1) < i) {
      if (j + 1 < i && a[j] < a[j + 1]) ++j;
      if (v >= a[j]) break;
      a[k] = a[j]; k = j; }
    a[k] = v; }
}
```

a) ExpID 325 - fastest LLM generated code

```c
void hpsort(unsigned long n, float ra[]) {
  unsigned long i, ir, j, l;
  float rra;
  if (n < 2) return;
  l = (n >> 1) + 1;
  ir = n;
  for (;;) {
    if (l > 1) rra = ra[--l]; //hiring phase
    else { //retirement and promotion phase
      rra = ra[ir];
      ra[ir] = ra[1];
      if (--ir == 1) { ra[1] = rra; break; }}
    i = l;
    j = l + l;
    while (j <= ir) {
      if (j < ir && ra[j] < ra[j + 1]) j++;
      if (rra < ra[j]) {
        ra[i] = ra[j];
        i = j;
        j <<= 1; }
      else break; }
    ra[i] = rra; }
}
```

b) Numerical Recipes in C baseline code

Figure 8: A code comparison between the fastest LLM generated code and that from Numerical Recipes in C shows distinctly different approaches.

## 6 FUTURE WORK

The achievement of obtaining a GPE-A based prompt that produces consistently repeatable results meeting the intent of the prompt, underscores the need for additional evaluation of how to effectively prompt LLM models. An evaluation is also necessary to determine if the prompt is equally as effective when used across a variety of LLM models. If the prompt is succeeding because it is well aligned with terminology specific to the training data for ChatGPT, it could be explored if incorporating language used in other LLMs' training data indicates the possibility of an ideal structure for a prompt to work with multiple LLMs. Such an effort would lay a strong foundation for experimentation with chaining LLMs together to achieve a comprehensive simulation software development pipeline. A database of best terms could emerge to achieve desired result, possibly different for each LLM. Any prompt obtained through this work, to include the best performing prompt from this work, should be evaluated for use across similar algorithms (such as those from *Numerical Recipes in C*) and tested for scaling to larger problems by decomposing the prompt into a highly specific framework that can be tested against complex problem sets such as simulation models. In addition, the work should be extended to alternative code metrics. An evaluation of this sort should provide a good understanding of the capabilities and limitations of prompts and a determination of how to obtain satisfactory generated software-related products. After developing this concrete understanding of prompt-based interaction with LLMs, an LLM centric development framework for simulation software engineering should be revisited to solidify the development process, now with a high probability for consistently generating high quality, reliable software employing modern best practices.

Finally, the work should be applied to a range of simulation models. The well-structured nature of the requirements in the form of simulation models could provide further clarity to the LLMs in the software development step. It also poses a clear set of requirements for the automated development of test cases by an LLM that could then be tested for completeness.

## 7 CONCLUSIONS

The authors were genuinely surprised by the raw performance capabilities demonstrated by the selected LLM, ChatGPT, particularly when benchmarked against hand-crafted code such as that found in *Numerical Recipes in C*. Despite expectations that role assignment would significantly influence performance, the impact was minimal and, in some cases, proved counterproductive. In contrast, prompt engineering emerged as a critical factor in influencing code performance. By default, the LLM prioritizes classical software development norms—readability, maintainability, reliability—which, while valuable, often conflict with raw performance optimization. However, through targeted prompt construction, the authors were able to steer the LLM's output away from these defaults. For instance, including explicit instructions effectively reoriented the LLM's reasoning and output generation toward performance-oriented solutions.

These findings are promising and highlight the potential of AI-assisted code generation, especially when optimized for specific, measurable metrics. Moreover, the concept of generating code from well-defined simulation models presents a scalable approach to software development. This enables the automated production of multiple candidate implementations, facilitating empirical comparison, benchmarking, and refinement. Such an approach could lead to iterative improvements and cross-fertilization of results, enhancing the ability to meet and exceed performance targets.

## REFERENCES

Arcuri, A., and L. Briand. 2014. "A Practical Guide for Using Statistical Tests to Assess Randomized Algorithms in Software Engineering." *Empirical Software Engineering* 19 (4): 1335–1382. https://doi.org/10.1007/s10664-013-9249-9.

Arvanitou, E.-M., A. Ampatzoglou, A. Chatzigeorgiou, and J. C. Carver. 2021. "Software Engineering Practices for Scientific Software Development: A Systematic Mapping Study." *Journal of Systems and Software* 172: 110848. https://doi.org/10.1016/j.jss.2020.110848.

Bavishi, R., N. Sundaresan, V. Raychev, S. Chandra, and M. Allamanis. 2023. "Code Generation with Generative Language Models: State of the Art and Open Challenges." *arXiv preprint arXiv:2301.12867*. https://arxiv.org/abs/2301.12867.

Carlsson, S. 1987. "A Variant of Heapsort with Almost Optimal Number of Comparisons." *Information Processing Letters* 24 (4): 247–250. https://doi.org/10.1016/0020-0190(87)90142-6.

Chen, M., J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, et al. 2021. "Evaluating Large Language Models Trained on Code." *arXiv preprint arXiv:2107.03374*. https://doi.org/10.48550/arXiv.2107.03374.

Dijkstra, E. W. 1981. "Smoothsort, an Alternative for Sorting in Situ." *Information Processing Letters* 1 (1): 1–8.

Dong, Y., X. Jiang, Z. Jin, and G. Li. 2023. *Self-Collaboration Code Generation via ChatGPT.* arXiv preprint arXiv:2304.07590, https://arxiv.org/abs/2304.07590.

Easterbrook, S., and T. Johns. 2009. "Engineering the Software for Understanding Climate Change." *Computing in Science & Engineering* 11 (6): 65–74. https://doi.org/10.1109/MCSE.2009.193.

Edelkamp, S., A. Elmasry, and J. Katajainen. 2013. "Weak Heaps Engineered." *Journal of Discrete Algorithms* 21: 3–16. https://doi.org/10.1016/j.jda.2013.07.002.

Floyd, R. W. 1964. "Algorithm 245: Treesort 3." *Communications of the ACM* 7 (12): 701. https://doi.org/10.1145/355588.365103.

Heroux, M. 2009. "Trust Me. QED." *SIAM News, July/August.*

Hall, T., S. Beecham, D. Bowes, D. Gray, and S. Counsell. 2012. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering." *IEEE Transactions on Software Engineering* 38 (12): 1276-1304. https://doi.org/10.1109/TSE.2011.103.

LaMarca, A., and R. E. Ladner. 1997. "The Influence of Caches on the Performance of Sorting." *Journal of Algorithms* 31 (1): 66–104. https://doi.org/10.1006/jagm.1998.0985.

Leathrum, J., Y. Shen, and M. Sosonkina. 2024. "Investigating the Use of Generative AI in M&S Education." In *Proceedings of the 2024 Winter Simulation Conference (WSC)*. https://doi.org/10.1109/WSC63780.2024.10838805.

Levcopoulos, C., and O. Petersson. 1993. "Adaptive Heapsort." *Journal of Algorithms* 14 (3): 395–413. https://doi.org/10.1006/jagm.1993.1021.

Nijkamp, E., B. Pang, H. Hayashi, L. Tu, H. Wang, Y. Zhou, et al. 2022. "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis." *arXiv preprint arXiv:2203.13474*. https://arxiv.org/abs/2203.13474.

OpenAI. 2025. "ChatGPT (February 2025 version)." OpenAI. https://chat.openai.com/, accessed 30th March 2025.

Press, W., S. Teukolsky, W. Vetterling, and B. Flannery. 1992. *Numerical Recipes in C: The Art of Scientific Computing* 2nd. ed. Cambridge, England: Cambridge University Press.

Radjenović, D., M. Heričko, R. Torkar, and A. Živkovič. 2013. "Software Fault Prediction Metrics: A Systematic Literature Review." *Information and Software Technology* 55 (8): 1397–1418. https://doi.org/10.1016/j.infsof.2013.02.009.

Sherifi, B., K. Slhoub, and F. Nembhard. 2024. "The Potential of LLMs in Automating Software Testing: From Generation to Reporting." *arXiv preprint arXiv:2501.00217*. https://arxiv.org/abs/2501.00217.

Svyatkovskiy, A., S. Deng, S. Fu, and N. Sundaresan. 2020. "IntelliCode Compose: Code Generation Using Transformer." In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020)*, 1433–1443. https://doi.org/10.1145/3368089.3417058.

Wang, G., Sun, Z., Gong, Z., Ye, S., Chen, Y., Zhao, et al. (2024). *Do advanced language models eliminate the need for prompt engineering in software engineering?* arXiv preprint arXiv:2411.02093. https://arxiv.org/abs/2411.02093.

Williams, J. W. J. 1964. "Algorithm 232: Heapsort." *Communications of the ACM* 7 (6): 347–48. https://doi.org/10.1145/512274.3734138.

White, M., M. Tufano, C. Vendome, and D. Poshyvanyk. 2023. "Prompt Engineering for Code Generation: Principles and Patterns." *arXiv preprint arXiv:2302.01348*. https://arxiv.org/abs/2302.01348.

xAI. 2025. "Grok-3." xAI. https://x.ai/, accessed 30th March 2025.

## AUTHOR BIOGRAPHIES

**JAMES F. LEATHRUM, JR.** is an Associate Professor in the Department of Electrical and Computer Engineering at Old Dominion University. He earned the Ph.D. in Electrical Engineering from Duke University. His research interests include simulation software design, distributed simulation, and simulation-based test & evaluation. His email address is jleathru@odu.edu.

**ABIGAIL S. BERARDI** is an undergraduate student in the Department of Electrical and Computer Engineering at Old Dominion University. She is concurrently pursuing a BS in Computer Engineering with a major in Modeling and Simulation Engineering and a BS in Electrical Engineering. Her e-mail address is abera003@odu.edu.

**YUZHONG SHEN** is a Professor and Associate Chair in the Department of Electrical and Computer Engineering at Old Dominion University. He earned the Ph.D. in Electrical Engineering from the University of Delaware. His research interests include virtual and augmented reality, computer graphics, transportation M&S, and signal processing. His e-mail address is yshen@odu.edu.