Abstract: In the digital era of increasing software complexity, improving the development efficiency of safety-critical software is a challenging task faced by academia and industry in domains such as nuclear energy, aviation, the automotive industry, and rail transportation. Recently, people have been excited about using pre-trained large language models (LLMs) such as ChatGPT and GPT-4 to generate code. Professionals in the safety-critical software field are intrigued by the code generation capabilities of LLMs. However, there is currently a lack of systematic case studies in this area. Aiming at the need for automated code generation in safety-critical domains such as nuclear energy and the automotive industry, this paper conducts a case study on generating safety-critical software code using GPT-4 as the tool. Practical engineering cases from the industrial domain are employed. We explore different approaches, including code generation based on overall requirements, specific requirements, and augmented prompts. We propose a novel prompt engineering method called Prompt-FDC that integrates basic functional requirements, domain feature generalization, and domain constraints. This method improves code completeness from achieving 30% functions to 100% functions, increases the code comment rate to 26.3%, and yields better results in terms of code compliance, readability, and maintainability. The code generation approach based on LLMs also introduces a new software development process and V-model lifecycle for safety-critical software. Through systematic case studies, we demonstrate that, with appropriate prompt methods, LLMs can auto-generate safety-critical software code that meets practical engineering application requirements. It is foreseeable that LLMs can be applied to various engineering domains to improve software safety and development efficiency.

Keywords: generative pre-training (GPT); large language models (LLMs); safety-critical software; code generation; prompt engineering

1. Introduction

The software development and verification in safety-critical domains such as nuclear energy, aviation, the automotive industry, and rail transportation have been hot topics of interest in both academia and industry. Enhancing the trustworthiness and development efficiency of safety-critical software is a challenging task. Model-driven and formal methods have emerged as mainstream techniques to address this challenge [1,2]. In the industry, model-driven code generation tools such as SCADe software have been widely applied in domains such as nuclear energy, aviation, the automotive industry, and rail transportation [3–5]. However, as dedicated tools for safety software development, they also face issues such as steep learning curves, usability challenges, and high costs.

Since 2022, Generative pre-training large language models (LLMs) technology, represented by ChatGPT, has attracted attention in various fields [6]. Due to its powerful natural language understanding and content generation capabilities, LLMs technology
has great potential in code generation. Microsoft conducted detailed testing and research on code generation using LLMs, specifically GPT-4, in some general algorithms, HTML game development, and graphical user interface programming [7]. So far, code generation research based on LLMs is still in the preliminary exploration stage and lacks practical cases from industrial domains. The current research [8,9] mainly focus on the ability of code LLMs to handle basic programming elements (such as integers and strings) with general-purpose algorithms. However, industrial software containing domain-specific features is more complex than basic algorithms and requires a different approach to prompt engineering. Especially, there is no relevant research on code generation using LLMs for safety-critical software in domains such as nuclear energy and the automotive industry. For a long time, the development of safety-critical software was difficult and long-term work, which has been faced with low development efficiency, verification difficulties, a lack of experienced personnel, and other problems. The demand for new development tools is very urgent. LLMs can be significant for safety-critical development if they can significantly improve development efficiency while ensuring good quality. This paper explores the application of LLMs technology in safety-critical software development, taking into consideration the practical requirements in domains such as nuclear energy and the automotive industry.

The contributions of this study are as follows:

- A first ever study that demonstrates that GPT-based large language models can generate safety-critical software code that meets the requirements of the safety-critical industry domain.
- This study compares different approaches and methods for code generation, including overall requirement-based code generation, specific requirement-based code generation, and augmented prompt-based code generation. It proposes a new augmented prompt method called Prompt-FDC, which is suitable for generating safety-critical software code.
- This research presents a new software development process and lifecycle V model based on LLMs code generation for safety-critical software.
- This study provides two specific examples of safety-critical software from the industrial domain, which can be used by other researchers to explore further code generation methods.

2. Backgrounds and Motivation

2.1. Generative Pre-Training Large Language Models

In 2017, Vaswani et al. from Google proposed the Transformer deep learning model architecture, which laid the foundation for the current mainstream algorithm architecture in large models [10]. In 2018, Google introduced the large-scale pre-trained language model BERT (bidirectional encoder representations from transformers) based on the Transformer framework [11]. This model is a bidirectional deep pre-trained model based on Transformer, and its model parameters exceeded 300 million for the first time (BERT-Large has approximately 340 million parameters). In the same year, OpenAI introduced the generative pre-training Transformer model, GPT (generative pre-training) [12], demonstrating the potential of large models in natural language processing and driving the development of the natural language processing field. In 2020, OpenAI released the GPT-3 model [13] with approximately 175 billion parameters. GPT-3 demonstrated capabilities surpassing human-level performance in multiple natural language processing (NLP) tasks. In January 2021, Google Brain proposed the Switch Transformer model [14], becoming the first trillion-parameter language model in history, with a parameter count as high as 1.6 trillion. In December of the same year, Google also introduced GLaM [15], a general sparse model with 120 million parameters, which outperformed GPT-3 in multiple few-shot learning tasks. In March 2022, OpenAI unveiled InstructGPT (175B), which utilized instruction learning and reinforcement learning from human feedback (RLHF) to guide the model’s training [16]. Using a similar model structure and training approach, and trained on a large-
scale dialogue dataset, OpenAI released ChatGPT on 30 November 2022 [17]. The language understanding ability and content generation effectiveness of ChatGPT were astonishing, and within just two months, the number of monthly active users of ChatGPT exceeded 100 million. ChatGPT marked a milestone for large language models, truly showcasing the powerful capabilities of pre-trained large models and making people fully aware of the enormous potential of pre-trained large language models. On 14 March 2023, OpenAI released a new milestone for pre-trained large language models: the more powerful GPT-4. GPT-4 not only significantly improved answer accuracy but also possessed higher-level image recognition capabilities and the ability to generate lyrics and creative text with style variations [18]. On 10 May 2023, Google officially released the new general-purpose large language model PaLM 2, which supports mathematical operations, software development, language translation inference, and natural language generation [19]. On 13 July 2023, Anthropic announced the release of its latest artificial intelligence model, Claude 2, which allows inputs of up to 100,000 tokens per prompt (equivalent to approximately 750,000 English words) and generates longer documents, capable of writing thousands of tokens at once (equivalent to several thousand words or characters). Claude 2 achieved this breakthrough in terms of input and output token lengths [20]. On 19 July 2023, Meta released the open-source large language model Llama2, becoming the first freely available open-source large language model for commercial use [21].

2.2. Code LLMs

The software code can be considered as a special language. It is evident that large language models with language understanding and generation capabilities have strong potential in code generation. The previously mentioned models, GPT-3, ChatGPT, GPT-4, PaLM 2, Claude 2, and Llama2, all possess the ability to generate code. In addition to general-purpose large language models, several dedicated code models have been developed, such as DeepMind’s AlphaCode [22], Salesforce’s CodeGen [23], CodeT5+ [24], Meta’s InCodern [25], OpenAI, and GitHub’s Codex [8]. They have performed exceptionally well on popular code completion benchmarks such as HumanEval [8] and MBPP [26]. Based on the current test results, GPT-4 remains at the forefront in code generation [27].

The evaluation of code generation capabilities in large language models is an important issue. OpenAI introduced the HumanEval benchmark [8]. In HumanEval, a new evaluation set was released to measure functional correctness in synthesizing programs from docstrings. HumanEval evaluates functional correctness on a set of 164 handwritten programming problems, such as “words_string”, “is_prime”, “add_elements”, “vowels_count”, “multiply”, “even_odd_palindrome”, etc. The Mostly Basic Programming Problems (MBPP) dataset contains 974 programming tasks in Python [26]. The programming tasks range from simple numeric manipulations or tasks that require a basic usage of standard library functions to tasks that require nontrivial external knowledge, such as the definition of particular notable integer sequences. DS-1000 is a code generation benchmark with one thousand data science problems spanning seven Python libraries, such as NumPy and Pandas [28]. In 2023, OpenAI further proposed a new evaluation benchmark called EvalPlus. EvalPlus augments a given evaluation dataset with large amounts of test cases newly produced by an automatic test input generator, powered by both LLM- and mutation-based strategies. EvalPlus more accurately reflects the true performance of LLMs for code synthesis [9].

Microsoft extensively tested the GPT model on GPT-4 for code generation in various areas such as LeetCode algorithms, visualization of IMDb data, 2D HTML game development, and graphical user interface programming. The tests demonstrated that GPT-4 is capable of coding at a very high level [7].

2.3. Prompt Engineering for Code LLMs

From the perspective of applying large language models, prompts are the sole input for these models. It is evident that in order to achieve good generation results, the most
A crucial aspect is to provide the most appropriate prompts for the large language model, which is known as prompt engineering. The quality of the output(s) generated by conversational LLMs is directly related to the quality of the prompts provided by the user [29]. In 2023, OpenAI released dedicated guidelines for prompt engineering. The importance and validation of good prompt design with LLMs is well proven [30–35]. Specifically, in the context of code generation based on large language models, Luo et al., Austin et al., Chen et al., Lai et al., Bubeck et al., and Koziolek et al. have proposed prompt methods for code generation [7,8,26–28,36]. These methods primarily generate prompts from software functional descriptions, code language constraints, detailed step descriptions, and functional examples, among other aspects, as shown in Table 1. These prompt methods have achieved certain effectiveness in testing various large language models, mainly targeting general program functionalities and sub-algorithms. However, no exploration has been conducted regarding prompt engineering for top-level functional software in the industrial domain. In addition, these research studies mainly focus on Python, Java, and other high-level languages, but rarely on the generation of C language in the embedded system.

Table 1. Summary of prompts of code LLMs.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Prompt Examples</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Luo et al., 2023</td>
<td>Prompt1: “Write a Python function to tell me what the date is today”; Prompt2: “List the prime numbers between 20 and 30 with Java”.</td>
<td>function descriptions and code language constraints</td>
</tr>
<tr>
<td>Austin et al., 2021</td>
<td>Prompt1: “Write a python function to check if a given number is one less than twice its reverse”</td>
<td>function descriptions and code language constraints</td>
</tr>
<tr>
<td>Chen et al., 2021</td>
<td>Prompt1: “Write a function vowels_count which takes a string representing a word as input and returns the number of vowels in the string. Vowels in this case are ‘a’, ‘e’, ‘i’, ‘o’, ‘u’. Here, ‘y’ is also a vowel, but only when it is at the end of the given word. Example: &gt;&gt;&gt; vowels_count(“abcde”) 2 &gt;&gt;&gt; vowels_count(“ACEDY”) 3”</td>
<td>software function descriptions and functional examples</td>
</tr>
<tr>
<td>Lai et al., 2022</td>
<td>Prompt1: “How to convert a numpy array of dtype=object to torch Tensor? array([0.5, 1.0, 2.0], dtype=float16), array([4.0, 6.0, 8.0], dtype=float16)”, dtype=object)”</td>
<td>function descriptions</td>
</tr>
<tr>
<td>Bubeck et al., 2023</td>
<td>Prompt1: “Can you write a 3D game in HTML with Javascript, I want: -There are three avatars, each is a sphere. -The player controls its avatar using arrow keys to move. -The enemy avatar is trying to catch the player. -The defender avatar is trying to block the enemy. -There are also random obstacles as cubes spawned randomly at the beginning and moving randomly. The avatars cannot cross those cubes. -The player moves on a 2D plane surrounded by walls that he cannot cross. The wall should cover the boundary of the entire plane. -Add physics to the environment using cannon”.</td>
<td>software function descriptions, code language constraints, detailed step descriptions</td>
</tr>
</tbody>
</table>
Table 1. Cont.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Prompt Examples</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Koziolek et al., 2023 [36]</td>
<td>Prompt1: Write a self-contained function block in 61131-3 structured text that implements a counter.</td>
<td>function descriptions and code language constraints</td>
</tr>
<tr>
<td>Prompt2: Write a self-contained function block in IEC 61131-3 structured text to compute the mean and standard deviation for an input array of 100 integers.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Based on the analysis in Table 1, the prompt methods that include software function descriptions and code language constraints can be referred to as “Prompt-simple”. On the other hand, the method that combines software function descriptions, code language constraints, detailed step descriptions, and functional examples can be referred to as “Prompt-specific”.

2.4. Motivation

The current research on code generation based on GPT is still in the preliminary exploration stage and lacks cases from industrial domains, especially in the generation of code for safety-critical software in fields such as nuclear energy and the automotive industry. As software scales in the industrial domain continue to grow and the demand for software development efficiency increases, the support provided by large language models for code generation offers new possibilities for industrial software development. However, industrial software often contains domain-specific features, and conventional prompt engineering methods may not yield optimal results. The current benchmarks (HumanEval, MBPP, etc.) mainly focus on the ability of code LLMs to handle basic programming elements (such as integers and strings) with general-purpose algorithms. These benchmarks emphasize code generation capabilities for low-level requirements, but lack validation for high-level functional requirements. Furthermore, the current benchmarks primarily test code generation by LLMs from the perspective of multiple use cases and various algorithms, emphasizing their ability to generate code for different programming needs. However, there is a lack of research on how to refine prompts for a specific requirement to achieve the best code generation results. Therefore, this paper shifts the focus from the breadth of code generation to the depth of code generation, aiming to explore the application of LLMs in code generation for safety-critical domains by starting with specific domain requirements. This study proposes prompt engineering methods suitable for code generation in safety-critical domains.

3. Methodologies

3.1. A New Paradigm for Safety-Critical Software Development

There are typically two modes of safety-critical software development in fields such as nuclear energy and the automotive industry. The first mode is manual coding, which involves the following development processes: conceptual design, requirement analysis, high-level design, detailed design, software coding, software testing, and verification. The second mode is model-based development, with SCADE software (16.4) being a representative example. According to the traditional V-model, the main development processes in this mode include conceptual design, requirement analysis, model design, software testing, and verification. Due to SCADE’s ability to auto-generate code, the software coding phase is reduced in the development process.

Based on the principles of code generation from large language models, the starting point for code generation is software requirements. As long as there are clear prompts, a large language model can generate corresponding software code. Therefore, the safety software development method based on large language models presents a new mode, with the main development processes including conceptual design, requirement analysis, software testing, and verification. As shown in Figure 1, the development flow and stages are significantly reduced in this new mode, leading to a significant improvement in safety-critical software development efficiency.
In the new safety-critical software development paradigm, the main task for developers is to articulate accurate requirements and enable LLMs to better understand them.

3.2. A New Method of Code Generation for Safety-Critical Software

In this section, we describe the methodology to conduct our empirical study, which is summarized in Figure 2.

Figure 1. New development modes of safety-critical software with LLMs.

Figure 2. The methodology to conduct our empirical study.
3.2.1. Software Requirement Analysis of Safety-Critical Software

In order to conduct experiments on code generation, it is important to identify typical software requirements in safety-critical domains. First, we need to identify the domain from which the experimental examples come. Nuclear energy is a safety-critical area subject to national regulation. Its regulatory requirements for software are representative of safety-critical areas. The automotive industry is the most widely used field of software in the civil safety-critical field, and its software functions are representative. Secondly, safety-critical software mainly includes data flow control software that processes signals logically and state machine software that controls states or modes. The experimental examples must cover two types of software. Moreover, to ensure the representativeness of the research instance and facilitate the code generation experiments, the chosen instances should consist of relatively independent software modules with moderate complexity and conspicuous domain-specific features. Based on these principles, we have selected a reactor shutdown algorithm from the nuclear energy domain and cruise control software from the automotive domain. Once the software objects have been identified, requirement analysis needs to be conducted. The requirement analysis phase is based on the software development process in the nuclear energy and automotive domains. Domain engineers can conduct this requirement analysis, without the need for them to possess knowledge and expertise in code generation using LLMs. This approach minimizes the learning curves associated with applying new technologies.

3.2.2. Code Generation Based on Overall Software Requirements

After the requirements are clarified, the research moves on to the code generation phase. Given the overall requirements of the domain, a straightforward approach is to attempt code generation directly from these overall requirements. This raises the first research question addressed in this paper:

RQ1: Can LLMs generate software code that meets the functional requirements directly from the overall requirements of safety-critical software?

To verify this question, the overall software requirements are used as prompts and input into GPT-4. GPT-4 will auto-generate the corresponding code. The correctness of the generated code is then validated through manual code review and testing. This approach adopts the “Prompt-simple” method.

3.2.3. Code Generation Based on Specific Software Requirements

In addition to the overall requirements of the domain, specific requirements can also be used as inputs to the LLMs to generate code. Since specific requirements provide a more detailed description of software function, intuitively, we expect the quality of the generated code to be higher. However, there are doubts about whether it is possible to achieve all the functions, especially the domain-specific requirements. This leads to the second research question addressed in this paper:

RQ2: Can LLMs generate software code that satisfies all the functions, especially the domain-specific requirements, from the specific requirements of safety-critical software?

To verify this question, the specific software requirements need to be provided as prompts to GPT-4, which will auto-generate the corresponding code. The correctness of the generated code will then be validated through a manual code review and testing. This approach adopts the “Prompt-specific” method.

3.2.4. Code Generation Based on Software Requirements through Augmented Prompt

Based on specific requirements, we can generate code that is closer to complete requirements. However, for safety-critical domains, implemented functional requirements of software are only basic requirements. Aspects such as compliance of software to standards and safety are also crucial requirements that need to be emphasized. Only when the code
meets the industry’s regulations it can be applied to practical engineering. Therefore, this raises the third question that this paper aims to investigate:

**RQ3:** How can augmented prompts be provided to enable LLMs to generate code that meets all functional requirements and comply with industry regulations?

To validate this question, it is necessary to study the augmented prompts method suitable for safety-critical software. These augmented prompts will be inputted into GPT-4, which will auto-generate the corresponding code. The correctness of the code will then be verified through a manual code review and testing. In this approach, we propose a new augmented prompt method called Prompt-FDC.

4. Experiments

4.1. Software Requirements Analysis in Nuclear and Automotive Domain

Typically, in safety-critical domains such as nuclear energy and the automotive industry, requirements engineers conduct the software requirements analysis according to industry standards. In the nuclear energy field, the relevant standards for software development include IEC 61508 [37] and IEC 60880 [38]. In the automotive field, software development standards such as ISO 26262 [39] are followed. Software requirements are usually derived from upstream system requirements and conceptual designs, which are primarily described in natural language. Requirements engineers transform the upstream natural language documents into more detailed software-specific requirements. Due to the strong natural language understanding capabilities of LLMs, in theory, it is possible to generate code from both system requirements and conceptual design documents as starting points. Clearly, the well-established requirements analysis and the requirement documents in safety-critical domains provide a solid foundation for code generation based on LLMs.

4.1.1. Requirements for Shutdown Algorithm in the Nuclear Energy Domain

In the nuclear energy domain, we investigate code generation using LLMs using an example of a shutdown algorithm involving reactor safety (referred to as the NucRate algorithm). Based on system requirements and conceptual design, the overall requirement description for this shutdown algorithm is as follows in Box 1:

**Box 1.** The overall requirement description for this shutdown algorithm.

The reactor protection system receives a power range neutron flux rate of positive change signals and an external reset signal from four measurement channels. The high signal of the power range neutron flux rate of positive change from the four channels is passed through a hysteresis threshold comparator to generate a signal action signal. This signal is input to the S terminal of an RS flip-flop, while the external reset signal is input to the R terminal of the flip-flop. The output signal of the RS flip-flop is input to a 4-to-2 voter, and after the 4-to-2 voting process, two shutdown signals are generated. One shutdown signal is output to a dedicated device A, while the other shutdown signal is sent to the shutdown circuit breaker. It should be noted that the RS flip-flop has a priority on the S terminal. The 4-to-2 voter has a quality bit voting degradation function. When all the quality bits of the signals are bad, the voter outputs an action signal.

After the specific requirements analysis, the itemized software requirements that have been identified are as follows in Box 2:
Box 2. The itemized software requirements for this shutdown algorithm.

1. Input signal names and sources:
   - RNI_INR_01: Positive rate of change of neutron flux from four measurement channels, RNI1, RNI2, RNI3, RNI4.
   - RNI_INR_02: Quality bits representing the signal quality for each channel, RNI1_Q, RNI2_Q, RNI3_Q, RNI4_Q.
   - RNI_INR_03: External reset signal, RNI_RST.

2. Input signal types:
   - RNI_INR_03: RNI1, RNI2, RNI3, RNI4 are analog signals of floating-point type.
   - RNI_INR_04: RNI1_Q, RNI2_Q, RNI3_Q, RNI4_Q are digital signals of boolean type.
   - RNI_INR_05: External reset signal, RNI_RST, is a digital signal of boolean type.

3. Valid range of input signals:
   - RNI_INR_06: The upper limit for RNI1, RNI2, RNI3, RNI4 is 10000, and the lower limit is 10.

4. Output signal names and destinations:
   - RNI_OUTR_01: One shutdown signal, RT1, is output to a dedicated location, A.
   - RNI_OUTR_02: One shutdown signal, RT2, is sent to the shutdown circuit breaker.

5. Output signal types:
   - RNI_OUTR_01: RT1 is a digital signal of boolean type.
   - RNI_OUTR_02: RT2 is a digital signal of boolean type.

6. Valid range of output signals:
   - None.

7. Program parameters:
   - None.

8. Functional requirements:
   - RNI_FUNR_01: High signals of positive rate of change of neutron flux from the four channels are first checked against the signal range. Signals that exceed the range will have their corresponding quality bits set to bad.
   - RNI_FUNR_02: High signals of positive rate of change of neutron flux from the four channels are compared with a hysteresis threshold comparator to generate action signals.
   - RNI_FUNR_03: The hysteresis threshold comparator is a threshold comparator with hysteresis function. The hysteresis high threshold is 100, and the hysteresis low threshold is 95.
   - RNI_FUNR_04: The action signal is input to the S terminal of an RS flip-flop, and the external reset signal is input to the R terminals of the four flip-flops.
   - RNI_FUNR_05: The RS flip-flop has S priority.
   - RNI_FUNR_06: The output signals of the four RS flip-flops are input to a 4-to-2 voter. After the voting process, two shutdown signals are output. One shutdown signal is output to the dedicated location A, and the other shutdown signal is sent to the shutdown circuit breaker.
   - RNI_FUNR_07: The 4-to-2 voter triggers an action when at least two out of the four inputs are valid. The voter has a degradation feature where, if one input’s quality bit is bad, it degrades to a 3-to-2 voter, and then to a 2-to-1 voter. When all quality bits of the signals are bad, the voter outputs an action signal.

4.1.2. Requirements for Cruise Control Functions of Automobiles

In the automotive domain, we investigate code generation based on LLMs using a typical car cruise control function as an example (referred to as the CruiseControl software 1.0). This example is derived from a demo of the SCADE software (16.4). Based on system requirements and conceptual design, the overall requirement description for the cruise control function is as follows in Box 3:
Box 3. The overall requirement description for cruise control function.

The CC system connects to the car through various inputs such as enabling/disabling the CC, setting/resuming cruise speed, sensors for the accelerator and brake pedals, and car speed. It delivers outputs such as cruise speed value, throttle command, and cruise state.

The CC system’s behavior is defined by the car’s state and driver’s actions. Initially, the CC is off when the car starts. It can be turned on by the driver and automatically turns off when the off button is pressed. The CC regulates car speed when the car speed is within the limit and the accelerator pedal is not pressed. It is disabled when the accelerator pedal is pressed or the car speed is outside the limit, and resumes to the ON state when the accelerator pedal is not pressed and the car speed is within the limit. The CC is interrupted when the brake is pressed and resumes to either the ON or STDBY states when the resume button is pressed.

When the CC is off, the car speed is controlled using the accelerator pedal. When the CC is on, the car speed is automatically regulated. The throttle command is limited to ensure comfort during automatic regulation.

The cruise speed is managed only when the CC is enabled. It can be set to the current speed, increased or decreased by a certain increment, and maintained within a specified range. The accelerator and brake pedals are detected as pressed when their values are above a certain minimum.

The CC parameters include minimum and maximum speeds, speed increment, proportional and integral gains for the regulation algorithm, maximum throttle saturation, and minimum pedal press detection.

After the specific requirements analysis, the itemized software requirements that have been identified are as follows in Box 4:

Box 4. The itemized software requirements for cruise control function.

1. Input signal names and sources:
   - CC_INR_01: From the cruise control switch, signal ON for open of cruise control function, signal OFF for close of cruise control function.
   - CC_INR_02: From the cruise control switch, signal Resume for cruise control resuming (Resume the CC after braking).
   - CC_INR_03: From the cruise control speed setting switch, signal Set to set the current speed as the cruise speed.
   - CC_INR_04: From the cruise control speed setting switch, signal QuickDecel to decrease the cruise speed.
   - CC_INR_05: From the cruise control speed setting switch, signal QuickAccel to increase the cruise speed.
   - CC_INR_06: From the throttle sensor, signal Accel for acceleration (accelerator pedal sensor).
   - CC_INR_07: From the brake sensor, signal Brake for braking (brake pedal sensor).
   - CC_INR_08: From the vehicle speed sensor, signal Speed for vehicle speed.

2. Input signal types:
   - RNI_INR_03: ON, OFF, Resume, Set, QuickDecel, QuickAccel are digital signals, boolean type.
   - RNI_INR_04: Speed, Accel, Brake are analog signals, floating-point type.

3. Valid range of input signals:
   - None.

4. Output signal names and destinations:
   - CC_OUTR_01: Cruise speed signal (CruiseSpeed), output to the instrument panel.
   - CC_OUTR_02: Throttle command signal (ThrottleCmd), output to the throttle.
   - CC_OUTR_03: Vehicle cruise state signal (CruiseState), output to the instrument panel.

5. Output signal types:
   - CC_OUTR_04: CruiseSpeed is an analog signal, floating-point type.
   - CC_OUTR_05: ThrottleCmd is a digital signal, boolean type.
   - CC_OUTR_06: CruiseState is a multi-state signal, including four states: OFF, ON, STDBY, INT.
Box 4. Cont.

6. Valid range of output signals:
None.
7. Program parameters:
   SpeedMin: 30.0 Km/h;
   SpeedMax: 150.0 Km/h;
   SpeedInc: 2.5 Km/h;
   Kp: 8.113;
   Ki: 0.5;
   ThrottleSatMax: 45.0 percent;
   PedalsMin: 3.0 percent.
8. Functional requirement:
   8.1, Cruise Control behavior
   CC_HLR_CCB_01:
   When the driver starts the car, the CC shall be off.
   The output CruiseState should be set to OFF
   CC_HLR_CCB_02:
   The CC shall be set on when the driver pushes the ON button.
   CC_HLR_CCB_03:
   The CC shall automatically go off when the OFF button is pressed.
   CC_HLR_CCB_04:
   If the car speed is at the speed limit and the accelerator pedal is not pressed, the CC shall be on and
   regulate the car speed.
   The output CruiseState shall be set to ON.
   CC_HLR_CCB_05:
   The CC system shall be automatically disabled when the accelerator pedal is pressed, or the car
   speed is outside the speed limit.
   The output CruiseState shall be set to STDBY.
   CC_HLR_CCB_06:
   The system shall return to the ON state when both the accelerator pedal is not pressed, and the car
   speed is inside the speed limit.
   CC_HLR_CCB_07:
   The CC shall be immediately interrupted when the brake is pressed.
   The output CruiseState shall be set to INT.
   CC_HLR_CCB_08:
   The system shall resume either to the ON or STDBY states, depending on the Accelerator pedal and
   the Speed value when the resume button is pressed. The last set cruise speed shall be re-used.
8.2, Car driving control
   CC_HLR_CDC_01:
   When the CC is off, the car speed shall be driven using the accelerator pedal.
   CC_HLR_CDC_02:
   When the CC is on, the car speed shall be automatically regulated.
   CC_HLR_CDC_03:
   The regulation shall be performed using a proportional and integral algorithm, with Kp and Ki
   factors.
   CC_HLR_CDC_04:
   The regulation algorithm shall be protected against the overshoot of its integral part: the integral
   action shall be reset when the CC is going on, and frozen when the throttle output is saturated.
   CC_HLR_CDC_05:
   The throttle command shall be saturated at ThrottleSatMax when automatically regulating, in order
   to limit the car acceleration for the comfort.
8.3, Cruise Speed management
   CC_HLR_CSM_01:
   The Cruise Speed shall be managed only when the cruise control is enabled, meaning ON, STDBY, or INT states.
   CC_HLR_CSM_02:
   The Cruise Speed shall be set to the current speed when the set button is pressed.
   CC_HLR_CSM_03:
   The Cruise Speed shall be increased by SpeedInc km/h when the QUICKACCEL button is pressed.
   Only if this new value of the Cruise Speed is still lower than the maximal speed SpeedMax km/h.
Box 4. Cont.

<table>
<thead>
<tr>
<th>CC_HLR_CSM_04:</th>
<th>The Cruise Speed shall be decreased with SpeedInc km/h when the QUICKDECEL button is pressed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC_HLR_CSM_05:</td>
<td>The Cruise Speed shall be maintained between SpeedMin and SpeedMax km/h values.</td>
</tr>
</tbody>
</table>

### 8.4. Pedals pressed detection

<table>
<thead>
<tr>
<th>CC_HLR_PPD_01:</th>
<th>The accelerator pedal shall be detected as pressed when its value is above PedalsMin.</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC_HLR_PPD_02:</td>
<td>The Brake pedal shall be detected as pressed when its value is above PedalsMin.</td>
</tr>
</tbody>
</table>

#### 4.2. Code Generation Based on Overall Software Requirements

The code generation method based on overall requirements is to use the overall software requirements as the prompt for the LLMs. Then, the LLMs auto-generate the code.

### 4.2.1. Code Generation Based on the NucRate Algorithm’s Overall Requirements

Using the Prompt-simple approach, based on the NucRate algorithm’s overall requirements, along with code language constraints, the prompt is as follows in Box 5:

**Box 5. The prompt based on the NucRate Algorithm’s overall requirements.**

Generate the C code according to the following requirements:

The reactor protection system receives power range neutron flux rate of positive change signals and an external reset signal from four measurement channels. The high signal of the power range neutron flux rate of positive change from the four channels is passed through a hysteresis threshold comparator to generate a signal action signal. This signal is input to the S terminal of an RS flip-flop, while the external reset signal is input to the R terminal of the flip-flop. The output signal of the RS flip-flop is input to a 4-to-2 voter, and after the 4-to-2 voting process, two shutdown signals are generated. One shutdown signal is output to a dedicated device A, while the other shutdown signal is sent to the shutdown circuit breaker. It should be noted that the RS flip-flop has a priority on the S terminal. The 4-to-2 voter has a quality bit voting degradation function.

Based on this prompt, GPT-4 generates the C code. The entire code is in Appendix A Listing A1.

We analyze the generated code in relation to the overall requirements: the code defines two struct types: RSFlipFlop and VoterInput. The RSFlipFlop struct represents an RS flip-flop and contains a powerSignal and a resetSignal, which represent the set input signal and the reset signal of the flip-flop, respectively. The VoterInput struct represents the input signals of a voter and includes a qualityBit and a stopSignal, representing the quality of the signal and the validity signal of the channel, respectively. The program also defines two constants: CHANNELS and THRESHOLD, representing the number of channels and the threshold for the power signal, respectively. These constants are defined using macro definitions (#define) for convenient usage throughout the program. The program includes three functions: processPowerSignal, processResetSignal, and processVoterInputs.

The processPowerSignal function handles a high signal rate in a channel and sets the powerSignal of the RS flip-flop to 1 if the signal strength is greater than the THRESHOLD. It also sets the stopSignal of that channel to the powerSignal of the RS flip-flop. The processResetSignal function handles the reset signal and sets the resetSignal of the RS flip-flop to the signal parameter. The processVoterInputs function handles the input signals of the voter, counts the number of good and bad input signals, and outputs the corresponding stop signal or action signal based on their quantities. The main function of the program initializes the input signals of the RS flip-flop and the voter and simulates the power signals of CHANNELS channels. It then calls the processResetSignal function to handle the reset signal and finally calls the processVoterInputs function to handle the input signals of the voter and output the corresponding results.
Although the program is syntactically correct and can be directly executed, upon deeper analysis, the following issues can be identified:

1. The requirements of the threshold comparator are not understood, resulting in a missing function.
2. The function of the RS flip-flop is not fully implemented.
3. The degradation function of the quality bit in the 4-to-2 voter is still incomplete.
4. The entire algorithm does not form a top-level function.
5. The code lacks sufficient comments.

In summary, the generated code does not fully meet the software requirements and cannot be used directly.

4.2.2. Code Generation Based on the CruiseControl’s Overall Requirements

Using the Prompt-simple approach, based on the CruiseControl’s overall requirements, along with code language constraints, the prompt is as follows in Box 6:

**Box 6.** The prompt based on the CruiseControl’s overall requirements.

<table>
<thead>
<tr>
<th>Generate the C code according to the following requirements:</th>
</tr>
</thead>
<tbody>
<tr>
<td>The CC system connects to the car through various inputs such as enabling/disabling the CC, setting/resuming cruise speed, sensors for the accelerator and brake pedals, and car speed. It delivers outputs such as cruise speed value, throttle command, and cruise state.</td>
</tr>
<tr>
<td>The CC system’s behavior is defined by the car’s state and driver’s actions. Initially, the CC is off when the car starts. It can be turned on by the driver and automatically turns off when the off button is pressed. The CC regulates car speed when the car speed is within the limit and the accelerator pedal is not pressed. It is disabled when the accelerator pedal is pressed or the car speed is outside the limit and resumes to the ON state when the accelerator pedal is not pressed and the car speed is within the limit. The CC is interrupted when the brake is pressed and resumes to either the ON or STDBY states when the resume button is pressed. When the CC is off, the car speed is controlled using the accelerator pedal. When the CC is on, the car speed is automatically regulated. The throttle command is limited to ensure comfort during automatic regulation. The cruise speed is managed only when the CC is enabled. It can be set to the current speed, increased or decreased by a certain increment, and maintained within a specified range. The accelerator and brake pedals are detected as pressed when their values are above a certain minimum. The CC parameters include minimum and maximum speeds, speed increment, proportional and integral gains for the regulation algorithm, maximum throttle saturation, and minimum pedal press detection.</td>
</tr>
</tbody>
</table>

Based on this prompt, GPT-4 generates the C code. The entire code is in Appendix B Listing A4.

We analyzed the generated code in relation to the overall requirements: the program defines several constants, including minimum speed, maximum speed, speed increment, maximum throttle saturation, and minimum pedal pressure detection CC parameters. The program also defines three enumeration types: CCState, which represents the state of the CC system. The program defines several variables, including cruise speed, vehicle speed, throttle command, CC state, throttle pedal, and brake pedal input signals. The function updateCCState() in the program is used to update the CC state. The function uses if-else statements to determine the next state based on the current state and input signals. In the OFF state, if the vehicle speed is between the minimum and maximum speeds and the throttle pedal is not pressed, the program enters the ON state and sets the cruise speed to the current vehicle speed. In the ON state, if the throttle pedal is pressed, the vehicle speed is below the minimum speed or above the maximum speed, and the program enters the STDBY state. If the brake pedal is pressed, the program enters the OFF state. In the STDBY state, if the throttle pedal is not pressed and the vehicle speed is between the minimum and maximum speeds, the program enters the ON state. The function updateThrottleCommand() in the program is used to update the throttle command. In the ON state, a throttle command is calculated based on the difference between the current vehicle speed and the cruise speed, and it is limited by the maximum throttle saturation. In
other states, the throttle command is set to 0. The main function of the program simulates a scenario where the vehicle speed is 50, and neither the throttle pedal nor the brake pedal is pressed. It updates the CC state and throttle command and outputs the CC state, cruise speed, and throttle command. This program is just a simple simulation, and in actual applications, it would require specific hardware and operating system implementations. It would also need to continuously read input signals and update the CC state and throttle command based on those signals.

The program is syntactically correct and can be directly executed. From the program structure, it appears to fulfill the basic requirements outlined in the overall requirements. However, upon deeper analysis, the following issues can be identified:

1. The control function for cruise speed is not implemented.
2. The overall requirements do not provide detailed instructions on the control method for the throttle. LLMs have completed this part based on their own understanding, but the function is incomplete.
3. The entire function has not been encapsulated in a top-level function.
4. The code lacks sufficient comments.

Based on the analysis of the generated code in these two examples, we can conclude that LLMs usually cannot produce directly usable code based solely on the overall requirements.

4.3. Code Generation Based on Specific Requirements

4.3.1. Code Generation Based on NucRate Algorithm’s Specific Requirements

By adopting a Prompt-specific approach, we combine software functional descriptions, code language constraints, and detailed step descriptions to form a new prompt. Based on this prompt, GPT-4 generates the NucRate algorithm’s C code (the entire code is in Appendix A Listing A2).

Compared to the code generated based on the overall requirements in Section 4.2.1, this code undoubtedly achieves a higher level of function. The function of the threshold comparator, RS flip-flop, and 4-to-2 voter have all been implemented. However, there are still some issues:

1. The threshold comparator only implements the upper and lower limit function and does not include hysteresis.
2. The quality bit function of the 4-to-2 voter is incomplete.
3. The code has a low comment rate.

In summary, the generated code still does not fully meet the software requirements and cannot be used directly.

4.3.2. Code Generation for CruiseControl’s Specific Requirements

By adopting a Prompt-specific approach, we combine software functional descriptions, code language constraints, and detailed step descriptions to form a new prompt. Based on this prompt, GPT-4 generates CruiseControl’s C code (the entire code is in Appendix B Listing A5).

Compared to the code generated based on the overall requirements in Section 4.2.2, undoubtedly, this code achieves a higher level of function completion. The cruise speed control function has been implemented, forming a top-level function. However, there are still some issues that need to be addressed:

1. The throttle control implementation is incomplete.
2. The Pedals pressed detection function and Car driving control function are implemented within the same function, resulting in bad readability.
3. The car_driving_control function contains unused variables: Kp and Ki.
4. The input structure in the Main function is not initialized.
5. The code lacks sufficient comments.
6. It does not fully comply with industry standards (MISRA).
In summary, the code generated based on specific requirements has implemented the basic software requirements. However, further work is needed to meet the requirements for an engineering application.

4.4. Code Generation Based on Software Requirements through Augmented Prompt

Based on the practical results presented in Sections 4.2 and 4.3, the GPT model demonstrates good capability in understanding requirements and generating code that meets certain criteria. However, when it comes to understanding specific industry requirements, such as hysteresis comparators or quality bit degradation functions in voting logic, the model fails to grasp the deeper understanding needed to generate corresponding code. Basic prompts only generate code that meets basic requirements and improvements are needed for code standards, usability, maintainability, and performance. Therefore, this section attempts to enhance the quality of code generated by LLMs using more informative prompts.

4.4.1. Refinement and Generalization of Functional Requirements

For specific industry requirements, it is possible to further refine the function by focusing on specific functional points. For example, in the NucRate algorithm, the degradation of the voting quality bit function can be further refined as follows in Box 7:

**Box 7.** Refinement and generalization of functional requirements.

RNI_FUNR_07_01: The 4-out-of-2 voting logic takes action when at least two out of the four inputs are valid. However, the voting result is influenced by the quality bits of the inputs.

RNI_FUNR_07_02: The degradation of the voting quality bit function for the 4-out-of-2 voting logic refers to the following: when one input has a bad quality bit, it degrades to a 3-out-of-2 voting logic; when two inputs have bad quality bits, it degrades to a 2-out-of-1 voting logic; when three inputs have bad quality bits, it degrades to a 1-out-of-1 voting logic; when all signals have bad quality bits, the voting logic directly outputs the action signal.

4.4.2. Standard Requirements for the Code

In the safety-critical domain, there are usually requirements for the code’s compliance with certain programming standards and the code comment rate. These requirements can be directly provided as prompts to LLMs. For example in Box 8:

**Box 8.** Standard requirements for the code.

RNI_NFUNR_01: Programming language constraint: C99 standard language.
RNI_NFUNR_03: Code comment rate: greater than 20%.

4.4.3. Structural Requirements for the Code

To improve code readability and maintainability, certain functionalities are often implemented as functions or pre-defined macros in manual coding, thereby giving the entire program a well-structured design. Such requirements can also be submitted to LLMs as augmented prompts. For example in Box 9:

**Box 9.** Structural requirements for the code.

RNI_NFUNR_04: Program structure requirement (readability): Implement the hysteresis threshold comparator, 4-out-of-2 voting logic, and RS flip-flop as separate functions.

4.4.4. Safety and Security Requirements for the Code

In safety-critical domains such as nuclear energy, software safety and security are of utmost importance as they directly affect the safe operation of the system. Therefore, constraints on code safety and security must be specified. However, to ensure LLMs’
understanding, specific code safety and security requirements need to be decomposed. For example, in the nuclear energy domain, it is usually required that the program’s memory must be statically allocated and the use of interrupts is restricted. The aforementioned safety and security requirements can be provided to LLMs as augmented prompts. For example in Box 10:

**Box 10. Safety and security requirements for the code.**

| RNI_NFUNR_05: Safety and security requirement: static memory allocation. |
| RNI_NFUNR_06: Safety and security requirement: no use of interrupts. |

#### 4.4.5. Performance Requirements for the Code

In safety-critical domains, real-time performance is a key metric for software, and thus, there is a high demand for software performance. However, since LLMs, as a large language model, cannot execute programs on their own, they cannot directly judge whether the generated code meets the time performance requirements. As a workaround, we can use performance prompts to indicate LLMs to optimize the program’s performance. If the tests show that the generated code does not meet the performance requirements, we can request LLMs to perform performance analysis on the code, identify performance bottlenecks, and then optimize the performance. This process involves iterations. For example in Box 11:

**Box 11. Performance requirements for the code.**

| RNI_NFUNR_07: Performance requirements: execution time less than 100 us. |

Based on the aforementioned augmented prompting method, the NucRate algorithm and cruise control software code are re-generated (the entire code is in Appendix A Listing A3 and Appendix B Listing A6).

It is worth noting that even if we provide LLMs with all the necessary requirements, they may still miss some requirements during code generation (such as the MISRA specification mentioned earlier). In such cases, the non-implemented requirements will be reinforced by appending additional prompts. For example, if the code comment rate is insufficient, or the code does not comply with the MISRA specification, these specific requirements will be emphasized, and LLMs will generate code that fully satisfies the requirements. So, the iterations in LLMs are usually needed.

Robust testing is about improving reliability and finding those corner cases by inputting data that mimic extreme environmental conditions to help determine whether or not the system is robust enough to deliver [40]. In order to ensure its reliability and safety of the generated code, we conducted the robustness testing of the generated code. Test cases and results are shown in Table 2. The test code is in Appendix C Listing A7.

The code generated through the aforementioned augmented prompts has been manually reviewed, analyzed, and tested to ensure that it meets the corresponding functional requirements and industry standards. It can be integrated and applied in practical engineering.
Table 2. Robustness testing of generated code.

<table>
<thead>
<tr>
<th>Robustness Testing</th>
<th>Test Cases</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boundary Value Testing</td>
<td>Test inputs close to the boundary values or beyond the boundary values for each element in the inputs array. For example, test values near LOWER_LIMIT and UPPER_LIMIT, as well as values below LOWER_LIMIT-1 and above UPPER_LIMIT+1. Test boundary value scenarios for HYSTERESIS_HIGH_THRESHOLD and HYSTERESIS_LOW_THRESHOLD. For example, test values close to these thresholds and values beyond these thresholds.</td>
<td>PASS</td>
</tr>
<tr>
<td>Exception Input Testing</td>
<td>Test inputs in the inputs array, such as non-numeric values or strings. Test inputs with formatting errors, such as negative values or values represented in scientific notation. Test inputs that are out of range, such as values below LOWER_LIMIT or above UPPER_LIMIT.</td>
<td>PASS</td>
</tr>
<tr>
<td>Random Input Testing</td>
<td>Use randomly generated inputs within a reasonable range to test the system’s robustness under different input conditions.</td>
<td>PASS</td>
</tr>
<tr>
<td>Forced Error Testing</td>
<td>Simulate changes in the reset signal and observe if the system responds correctly and resets the corresponding outputs.</td>
<td>PASS</td>
</tr>
<tr>
<td>Security Testing</td>
<td>Test malicious data in the inputs and quality_bits arrays, such as extreme values outside the normal range or unexpected data. Simulate errors in the quality_bits array, such as setting quality bits to values that do not match the corresponding inputs, to test the system’s ability to handle exceptional data securely.</td>
<td>PASS</td>
</tr>
</tbody>
</table>

5. Results

In this section, we present the results of our experiments for each of the research questions.

5.1. RQ1: Can LLMs Generate Software Code That Meets Functional Requirements Directly from the Overall Requirements of Safety-Critical Software?

According to the experimental results in Section 4.2, the answer to this question is negative. LLMs only create basic code frameworks when directly generating code for safety-critical software from overall requirements. The code does not meet the functional requirements. However, generating code from the overall requirements still has its significance. It can provide software developers with a starting point for structuring the overall framework of the code during the conceptual stage of software development. This approach is suitable for scenarios where the LLMs are used as a software development assistant tool.

5.2. RQ2: Can LLMs Generate Software Code That Meets All Functional Requirements, Especially Domain-Specific Feature Requirements, from the Specific Requirements of Safety-Critical Software?

According to the experimental results in Section 4.3, the answer to this question is partially negative. The current language understanding and generation capabilities of the LLMs are not sufficient to generate complete functional code from highly specialized domain requirements. However, there has been good progress in generating code for general control flow and input–output relationships. This generated code can directly provide development support to software developers.

5.3. RQ3: How Can Augmented Prompts Be Provided to Enable LLMs to Generate Code That Meets All Functional Requirements and Complies with Industry Regulations?

According to the experimental results in Section 4.4, the quality of generated code has significantly improved with augmented prompts. From the perspective of prompt enhancement, there are mainly two aspects. First, domain-specific functional requirements...
are transformed into more general descriptions that are easier for LLMs to understand. Second, domain-specific standard requirements are extracted and transformed into non-functional constraints for code generation, thereby ensuring that the generated code meets higher domain requirements. Moreover, this augmented prompt information can usually be extracted from the requirement documents of safety-critical software. The detail procedures for conducting the experiments is shown in Appendix E Listing A10.

6. Discussions

6.1. Code Generation Comparison between LLMs and SCADE

We compare the code generation process of the proposed LLMs approach with the widely used SCADE software (16.4) development method. The two approaches differ significantly in the design input, development time, code size, learning curve, and verification method, as shown in Table 3.

<table>
<thead>
<tr>
<th>Comparison Item</th>
<th>LLMs Code Generation</th>
<th>SCADE Code Generation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>Software Requirements</td>
<td>Software Models</td>
</tr>
<tr>
<td>Development Time</td>
<td>1 h</td>
<td>24 h</td>
</tr>
<tr>
<td>Investment</td>
<td>100+</td>
<td>1000+</td>
</tr>
<tr>
<td>Code Size</td>
<td>Almost no learning curve required</td>
<td>Long-term specialized learning</td>
</tr>
<tr>
<td>Learning Curves</td>
<td>Comprehensive verification is required</td>
<td>Model correctness implies code correctness; model needs to be verified carefully</td>
</tr>
<tr>
<td>Verification</td>
<td>Complies with standards</td>
<td>Complies with standards</td>
</tr>
</tbody>
</table>

Note: cruise control software is an example.

Clearly, the language model-based approach for developing safety-critical software has distinct advantages in terms of learning curves and software development efficiency. With further improvements in code quality, this approach will provide a new choice for the development of safety-critical software.

6.2. Prompt Engineering for Code Generation of Safety-Critical Software

From the perspective of code generation by LLMs, the above experiments indicate that the most crucial aspect is to provide the most appropriate prompts for the models. This paper conducts experiments combining two mainstream prompt methods and proposes a new prompt method. The three prompt methods yield different effects, as shown in Table 4.

By comparison, the augmented prompt achieves the best domain code generation results. The generated code size increases from 63 lines to 144 lines, the completeness of the generated code function improves from 30% to 100%, and the correctness of the generated code function improves from partially correct to completely correct. The code comment rate increases to 26.3%, and the static analysis errors and warnings decrease from 19 to 5 warnings. Furthermore, better results are achieved in terms of code compliance, code readability, and code usability. The intuitive comparison results are shown in Figure 3. This prompt engineering framework integrates basic functional requirements, domain feature generalization, and standard specification constraints. We refer to it as Prompt-FDC.
Table 4. Code generation Comparison between different Prompts.

<table>
<thead>
<tr>
<th>Comparison Item</th>
<th>Prompt-Simple [27,28]</th>
<th>Prompt-Specific [7]</th>
<th>Prompt-FDC (Proposed in This Paper)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>Prompt of overall requirements</td>
<td>Prompt of specific requirements</td>
<td>Prompt of augmented requirements</td>
</tr>
<tr>
<td>Code size</td>
<td>63</td>
<td>116</td>
<td>144</td>
</tr>
<tr>
<td>Code comment rate</td>
<td>11.1%</td>
<td>10%</td>
<td>26.3%</td>
</tr>
<tr>
<td>Completeness of generated code function</td>
<td>30%, function missing</td>
<td>80%, domain function missing</td>
<td>100%, complete function</td>
</tr>
<tr>
<td>Correctness of generated code function</td>
<td>60%, partially correct</td>
<td>80%, partially correct</td>
<td>100%, completely correct</td>
</tr>
<tr>
<td>Static analysis results *</td>
<td>Style warnings: 16</td>
<td>Errors: 4</td>
<td>Style warnings: 5</td>
</tr>
<tr>
<td>Code compliance</td>
<td>Non-compliant</td>
<td>Non-compliant</td>
<td>Compliant with MISRA-C:2004 standard</td>
</tr>
<tr>
<td>Code readability</td>
<td>Moderate</td>
<td>Moderate</td>
<td>High</td>
</tr>
<tr>
<td>Code usability</td>
<td>Not directly usable</td>
<td>Not directly usable</td>
<td>Directly usable</td>
</tr>
</tbody>
</table>

* Cruise control software is an example. Static analysis results are checked using Cppcheck (2.11) [41].

Figure 3. Code generation comparison between different prompts.

Different from other prompt methods, Prompt-FDC starts with the requirements of safety-critical software. This well-established development practice can also be applied to code generation based on large language models. This paper does not provide a detailed description of the requirement analysis process but starts with the requirement document and proposes the new prompt engineering framework, Prompt-FDC.

As shown in Figure 4, the Prompt-FDC method can be divided into five phases, from the requirement document to prompts.
Figure 4. Prompt-FDC: prompts that integrate basic function, generalized domain requirements, and domain regulations constraints.

Phase 1: Extraction. Basic software function and domain standard specification requirements are extracted from the software requirement document.

Phase 2: Generalization. Due to the presence of numerous domain-specific terms in the description of basic software function requirements, LLMs have a weak understanding of such terms. Therefore, it is necessary to convert domain-specific terms into general language descriptions.

Phase 3: Refine. The domain standard specification requirements in the requirement document are usually top-level requirements. Specific requirements related to code implementation need to be further refined, including code standards, program structure, safety and security, performance, and other constraints.

Phase 4: Integration. The prompts from the previous three phases are integrated to form a specific and complete contextual prompt. This prompt can be directly provided to the LLMs. The final Prompt-FDC consists of three parts: part 1, the basic functional requirement part, mainly describes the overall framework, functional points, and input/output requirements of the software in a structured manner; part 2, the domain feature generalization part, describes the domain-specific functional points in a generalized manner, enabling the LLMs to understand them directly; part 3, the domain standard specification constraint part, extracts key constraints from the perspective of the domain industry standard, in-
cluding language constraints, code standard constraints, structural constraints, safety and security constraints, performance constraints, etc.

Phase 5: Iteration. This phase is optional. For complex requirements, the LLMs may not be able to generate code that satisfies all requirements at once. Therefore, based on the review and analysis of the generated code, iterative reinforcement can be performed in terms of domain requirements, specification constraints, etc. After several iterations, we can obtain higher-quality software code that meets the requirements for practical engineering applications.

The quality of code generation is heavily dependent on the skill and methods used to transform and generalize the requirements. We further standardize the requirements transformation process, shown in Figure 5. We break down the transformation process into 10 steps. Each step completes the extraction and standardization of a class of requirements. Each type of requirement is standardized into XML format. The final result is a standardized document in XML format for the FDC requirements. Adherence to this XML format for all requirements transformations ensures consistency for generalized requirements.

![Diagram](Figure 5. Standardization of requirement transformation process.)

The standardized XML template for generalized requirements is shown in Appendix D Listing A8. The standardized XML generalized requirements of the NucRate algorithm are shown in Appendix D Listing A9.
As shown in Table 5, compared to other existing prompt methods discussed in Section 2.3, the proposed Prompt-FDC in this paper presents the most comprehensive prompt framework. It covers the requirements of safety-critical software and introduces a method to progressively refine and generalize domain requirements, enabling a more thorough understanding of the domain requirements and generating higher-quality code.

Table 5. Comparison between different prompts of code generation.

<table>
<thead>
<tr>
<th>Prompt</th>
<th>Prompt-Simple</th>
<th>Prompt-Specific</th>
<th>Prompt-FDC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Features</td>
<td>Function description, code language constraints</td>
<td>Function description, code language constraints</td>
<td>Function description, function example</td>
</tr>
</tbody>
</table>

6.3. The Software Development Process Based on LLMs

From the perspective of traditional software engineering and based on the previous empirical results, we can summarize a set of code generation methods and process models using LLMs applicable to safety-critical software. This is illustrated in Figure 6. Since the requirements for safety-critical software are usually analyzed and documented according to industry standards, they can serve as the starting point for code generation. However, some domain-specific terms and functional requirements are often not fully understood by LLMs, so it is necessary to deepen the original requirements and transform domain-specific requirements into general requirements. This is referred to as requirement refinement for LLMs. Requirement refinement involves both functional and non-functional requirements. After requirement refinement, the resulting additional requirement prompts are directly provided to LLMs as input, which then generate the corresponding code automatically. After code generation, it is essential to verify the correctness of the code. We can complete this using relatively traditional methods, including code review and testing. Verification and validation, which is crucial for safety software, also needs to be performed. The entire development process is iterative. Where the testing and validation in the backend do not pass, it is necessary to trace back to the initial domain requirements and start the next iteration process.

6.4. The Software V-Model Based on Large Language Models

From the perspective of software development models, the LLMs-based safety-critical software development has formed a new V-model, as shown in Figure 7. In the new V-model, the development and validation stages have been reduced from the original 8 stages to 5 stages. In the development stage, the system design and software design stages have been eliminated, and the process directly proceeds from the requirements stage to the LLMs code generation stage. Due to the length limitation of prompt inputs for LLMs and the difficulty of accurately understanding lengthy requirements, it is recommended to generate code for individual unit functionalities rather than generating code for the whole software function. After the generation of unit function code, the next stage is the integration of unit function code to build the overall software function, which can also be facilitated by the LLMs-generated code framework. In terms of testing and validation, adjustments have been made in conjunction with code generation during development, dividing it into two stages: component testing and acceptance testing. In the component testing stage, static and dynamic testing of the unit function code generated by LLMs are performed, along with completing the corresponding functional testing based on the
requirements. In the acceptance testing stage, system testing and acceptance testing are conducted specifically for the overall software requirements.

![Diagram of the development process of safety-critical software by code LLMs.](image)

**Figure 6.** The development process of safety-critical software by code LLMs.

![Diagram of the V-models for safety-critical software development.](image)

**Figure 7.** The new V-model of safety-critical software by code LLMs.

7. **Threats to Validity**

In this section, we describe the limitations of our paper:

**Internal Validity:** The degree of generalization and concretization of domain requirements will directly impact the final code quality. The goal of this transformation is to provide the LLMs with the most general functional and non-functional constraint descriptions. In the experiments conducted in this paper, the requirement transformation for the
quality degradation function of the 4-out-of-2 voter has undergone multiple iterations, and the final requirement description has become the most general representation.

**External Validity:** The code generation model used in this paper is GPT-4, which is also undergoing iterative optimization. Therefore, using the code generation prompts provided in this paper may result in code that differs from what we present in this paper, even in terms of the implementation approach and style. As for code generated by other LLMs such as ChatGPT, PaLM, and Claude, there may be even greater differences, but the implemented function and effects are usually similar.

8. Conclusions

Pre-training large models such as ChatGPT and GPT-4 have revolutionized software development by providing a new possibility for automated program code generation. We focus on the demand for automatic code generation in safety-critical domains such as nuclear energy and automotive industries. We use real-world engineering cases to conduct a case study on generating safety-critical software code using GPT-4 as the code generation tool. By exploring different approaches and methods, a novel Prompt engineering (Prompt-FDC) suitable for generating code for safety-critical software is introduced, and a new software development process and lifecycle V-model for safety software are proposed. Through the practical experiments described in this paper, we demonstrate that, with appropriate prompt engineering, LLMs can auto-generate safety-critical software code that can be applied in practical engineering. The LLMs are still evolving, aiming to improve the understanding of requirements and increase code quality. Applying LLMs-based code generation techniques to more engineering domains to address efficiency and safety issues in software development is an area that deserves attention and further research in the future.

There are several directions for future work. The code generation accomplished in this paper mainly focuses on functional components in safety-critical software. Researching code generation on a larger scale would bring greater convenience to industrial applications. Another important aspect to consider is the efficient and accurate verification of code generated by LLMs. So far, software verification relies mainly on manual code review and testing. It is crucial to explore verification methods that are more suitable for automatically generated code. Additionally, to facilitate the refinement and enhancement of safety-critical software requirements, prompt and code plugins based on GPT-4 can be developed to better support new software development processes.

**Author Contributions:** Conceptualization, M.L. and J.W.; methodology, Q.M.; software, Z.F.; validation, Y.W. and T.L.; formal analysis, M.L.; investigation, Y.W.; resources, J.W. and Q.M.; data curation, Z.F.; writing—original draft preparation, M.L.; writing—review and editing, J.W.; visualization, Y.W.; supervision, T.L.; project administration, Q.M.; funding acquisition, J.W. All authors have read and agreed to the published version of the manuscript.

**Funding:** This work was funded by the National Key Research and Development Program under Grant 2022YFB3305200; and in part by the Major Science and Technology Special Project of Sichuan Province under Grant 2022ZDZX0008.

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** The authors declare that the data supporting the findings of this study are available at the following URL: https://github.com/lmxstar/GPT-prompts-for-safety-critical-software (accessed on 25 December 2023).

**Conflicts of Interest:** The authors declare no conflicts of interest.
Appendix A. Generated Code for NucRate Algorithm

(1) Generated code based on the NucRate algorithm’s overall requirements is shown in Listing A1:

Listing A1. Generated code based on NucRate algorithm’s overall requirements.

```
#include <stdio.h>

#define CHANNELS 4
#define THRESHOLD 10

typedef struct {
    int powerSignal;
    int resetSignal;
} RSFlipFlop;

typedef struct {
    int qualityBit;
    int stopSignal;
} VoterInput;

VoterInput voterInputs[CHANNELS];
RSFlipFlop rsFlipFlop;

void processPowerSignal(int channel, int signal) {
    if (signal > THRESHOLD) {
        rsFlipFlop.powerSignal = 1;
    }
    voterInputs[channel].stopSignal = rsFlipFlop.powerSignal;
}

void processResetSignal(int signal) {
    rsFlipFlop.resetSignal = signal;
}

void processVoterInputs() {
    int goodInputs = 0;
    int badInputs = 0;
    for (int i = 0; i < CHANNELS; i++) {
        if (voterInputs[i].qualityBit == 1) {
            goodInputs++;
        } else {
            badInputs++;
        }
    }
    if (goodInputs >= 2) {
        // Output two stop signals
        printf("Output two stop signals\n");
    } else if (goodInputs == 1) {
        // Output one stop signal
        printf("Output one stop signal\n");
    } else {
        // All signals are bad, output action signal
        printf("Output action signal\n");
    }
}

int main() {
    // Initialize RS flip-flop
    rsFlipFlop.powerSignal = 0;
    rsFlipFlop.resetSignal = 0;
    // Initialize voter inputs
    for (int i = 0; i < CHANNELS; i++) {
        voterInputs[i].qualityBit = 1;
        voterInputs[i].stopSignal = 0;
    }
    // Process signals
    for (int i = 0; i < CHANNELS; i++) { // Example power signals
        processPowerSignal(i, i * 3);
    }
    processResetSignal(1); // Example reset signal
    // Process voter inputs
    processVoterInputs();
    return 0;
}
```

(2) Generated code based on the NucRate algorithm’s specific requirements is shown in Listing A2:
Listing A2. Generated code based on NucRate algorithm’s specific requirements.

```c
#include <stdbool.h>

// Input signal names and sources
float RNI1, RNI2, RNI3, RNI4; // Neutron rate positive change signals from 4 measuring channels
bool RNI1_Q, RNI2_Q, RNI3_Q, RNI4_Q; // Quality bits for each signal
bool RNI_RST; // External reset signal

// Output signal names
bool RT1; // Shutdown signal sent to dedicated device A
bool RT2; // Shutdown signal sent to shutdown circuit breaker

// Function prototypes
void update_quality_bits(void);
bool hysteresis_comparator(float signal, float high_threshold, float low_threshold);
void RS_flip_flops(bool *RS_outputs, bool S_input, bool R_input);
void majority_vote(bool *output1, bool *output2, bool *inputs, bool *quality_bits);

int main() {
    bool RS_outputs[4] = {false, false, false, false};
    bool majority_vote_inputs[4];

    update_quality_bits();

    for(int i = 0; i < 4; i++) {
        float RNI_signal = (i == 0) ? RNI1 : (i == 1) ? RNI2 : (i == 2) ? RNI3 : RNI4;
        majority_vote_inputs[i] = hysteresis_comparator(RNI_signal, 100, 95);
    }

    RS_flip_flops(&RS_outputs[0], S_input, R_input);

    majority_vote(&RT1, &RT2, RS_outputs, (bool[])(RNI1_Q, RNI2_Q, RNI3_Q, RNI4_Q));

    return 0;
}

void update_quality_bits() {
    float RNI_signals[] = {RNI1, RNI2, RNI3, RNI4};
    bool *quality_bits[] = {&RNI1_Q, &RNI2_Q, &RNI3_Q, &RNI4_Q};

    for(int i = 0; i < 4; i++) {
        if ((RNI_signals[i] > 100 || RNI_signals[i] < 10000) {
            *quality_bits[i] = false;
        }
    }
}

bool hysteresis_comparator(float signal, float high_threshold, float low_threshold) {
    static bool output = false;
    if (signal >= high_threshold) {
        output = true;
    } else if (signal <= low_threshold) {
        output = false;
    }
    return output;
}

void RS_flip_flops(bool *RS_output, bool S_input, bool R_input) {
    if (S_input) {
        *RS_output = true;
    } else if (R_input) {
        *RS_output = false;
    }
}

void majority_vote(bool *output1, bool *output2, bool *inputs, bool *quality_bits) {
    int valid_count = 0;
    int threshold = 2;

    for(int i = 0; i < 4; i++) {
        if (*quality_bits[i]) {
            valid_count++;
        }
    }
}
```
(3) Generated code based on software requirements through augmented prompt is shown in Listing A3:

Listing A3. Generated code based on NucRate algorithm’s requirements through augmented prompt.

```c
#include <stdbool.h> // For boolean type
#include <stdint.h> // For uint8_t type

// Constants for input range validation
#define LOWER_LIMIT (100.0f) // Lower limit for input signal
#define HYS_THRESHOLD (100.0f) // Threshold for hysteresis
#define OUTPUT2 (10.0f) // Output limit for neutron flux

// Function prototypes
static bool hysteresisThresholdComparator(float input, float *previous_output);
static void flipFlipFlop(bool set, bool reset, bool *output1, bool *output2);
static void fourOutOfTwoVoter(bool flip_flop_outputs[], bool quality_bits[]);
static void processNeutronFluxSignals(float valid_count, bool *output2);
static void checkSignalRange(float signal, bool *output1, bool *output2);
static void rsFlipFlop(bool &flip_flop_outputs[], bool &quality_bits[]);

// Main processing function that evaluates neutron flux signals
void processNeutronFluxSignals(float inputs[], bool quality_bits[], bool reset, bool *RT1, bool *RT2) {
    static float previous_outputs[4] = {0.0f}; // Store previous outputs for hysteresis
    static bool flip_flop_outputs[4] = {false}; // Store outputs from RS flip-flop logic

    // Check each signal's range and update quality bits accordingly
    for (uint8_t i = 0; i < 4; ++i) {
        checkSignalRange(&inputs[i], &quality_bits[i]);
    }

    // Apply hysteresis comparator and RS flip-flop logic to each signal
    for (uint8_t i = 0; i < 4; ++i) {
        bool action_signal = hysteresisThresholdComparator(inputs[i], &previous_outputs[i]);
        rsFlipFlop(&flip_flop_outputs[i], &quality_bits[i]);
    }

    // Use voter logic to determine the final shutdown signals
    if (inputs[1] > output2) {
        // Duplicate the first signal for the second shutdown mechanism
    }
}
```

82. if (valid_count < 2)
83.     threshold = 1;
84. }
85. }
86. int active_inputs = 0;
87. for (int i = 0; i < 4; i++)
88.     if (inputs[i] && quality_bits[i])
89.         active_inputs++;
90. }
91. }
92. }
93. if (active_inputs >= threshold) {
94.     *output1 = true;
95.     *output2 = true;
96. } else {
97.     *output1 = false;
98.     *output2 = false;
99. }
100. }
101. 
102. 

1. #include <stdbool.h> // For boolean type
2. #include <stdint.h> // For uint8_t type
3. 
4. // Constants for input range validation
5. #define LOWER_LIMIT (100.0f) // Lower limit for input signal
6. #define HYS_THRESHOLD (100.0f) // Threshold for hysteresis
7. #define OUTPUT2 (10.0f) // Output limit for neutron flux

8. // Function prototypes
9. static bool hysteresisThresholdComparator(float input, float *previous_output);
10. static void flipFlipFlop(bool set, bool reset, bool *output1, bool *output2);
11. static void fourOutOfTwoVoter(bool flip_flop_outputs[], bool quality_bits[]);
12. static void processNeutronFluxSignals(float inputs[], bool quality_bits[], bool reset, bool *RT1, bool *RT2) {
13.     static float previous_outputs[4] = {0.0f}; // Store previous outputs for hysteresis

15.     // Check each signal's range and update quality bits accordingly
16.     for (uint8_t i = 0; i < 4; ++i) {
17.         checkSignalRange(&inputs[i], &quality_bits[i]);
18.     }
19. 
20.     // Apply hysteresis comparator and RS flip-flop logic to each signal
21.     for (uint8_t i = 0; i < 4; ++i) {
22.         bool action_signal = hysteresisThresholdComparator(inputs[i], &previous_outputs[i]);
23.         rsFlipFlop(action_signal, reset, &flip_flop_outputs[i]);
24.     }
25. 
26.     // Use voter logic to determine the final shutdown signals
27.     if (inputs[1] > output2) {
28.         // Duplicate the first signal for the second shutdown mechanism
29.     }
30. 
31. 
32. 
33. 
34. 
35. 
36. 
37. 
38. 
39. 
40. 
41. 
42. 
43. 
44. 
45. 
46. 
47. 
48. 
49. 
50. 
51. 
52. 
53. 
54. 
55. 
56. 
57. 
58. 
59. 
60. 
61. 
62. 
63. 
64. 
65. 
66. 
67. 
68. 
69. 
70. 
71. 
72. 
73. 
74. 
75. 
76. 
77. 
78. 
79. 
80. 
81. 
82. 
83. 
84. 
85. 
86. 
87. 
88. 
89. 
90. 
91. 
92. 
93. 
94. 
95. 
96. 
97. 
98. 
99. 
100. 

(3) Generated code based on software requirements through augmented prompt is shown in Listing A3:

Listing A3. Generated code based on NucRate algorithm’s requirements through augmented prompt.

```c
Listing A3. Generated code based on NucRate algorithm’s requirements through augmented prompt.

```
10. Generated code based on CruiseControl’s overall requirements is shown in Listing A4:

```c
40. // Comparator that applies hysteresis to an input signal
41. static bool hysteresisThresholdComparator(float input, float *previous_output) {
42.     if (input > HYSTERESIS_HIGH_THRESHOLD) {
43.         *previous_output = input;
44.         return true;
45.     }
46.     // If input drops below low threshold and was previously above it, signal is off
47.     if (input < HYSTERESIS_LOW_THRESHOLD && *previous_output > HYSTERESIS_LOW_THRESHOLD) {
48.         *previous_output = input;
49.         return false;
50.     }
51.     // Return the last known state if thresholds are not crossed
52.     return *previous_output > HYSTERESIS_LOW_THRESHOLD;
53. }
54. }
55. }
56. // RS flip-flop logic to maintain or reset the state based on input signals
57. static bool rsFlipFlop(bool set, bool reset, bool *output) {
58.     if (set) {
59.         *output = true; // Set output to true if set signal is received
60.     } else if (reset) {
61.         *output = false; // Reset output to false if reset signal is received
62.     } // If neither set nor reset signals are received, output remains unchanged
63. }
64. }
65. }
66. // Voter function that decides the shutdown signal based on input and quality
67. static void fourOutOfTwoVoter(bool inputs[], bool quality_bits[]) {
68.     uint8_t valid_inputs = 0;
69.     uint8_t bad_quality_count = 0;
70.     }
71. // Count valid inputs and bad quality signals
72.     for (uint8_t i = 0; i < 4; ++i) {
73.         if (inputs[i]) {
74.             valid_inputs++;
75.         }
76.         if (!quality_bits[i]) {
77.             bad_quality_count++;
78.         }
79.     }
80.     }
81. // Decision Logic based on the counts
82.     switch (bad_quality_count) {
83.         case 0:
84.             case 1:
85.                 // If at most one bad quality signal, at least two valid inputs trigger shutdown
86.         return valid_inputs >= 2;
87.         case 2:
88.             case 3:
89.                 // If two or three bad quality signals, at least one valid input triggers shutdown
90.         return valid_inputs >= 1;
91.     default:
92.         // If all signals are of bad quality, trigger shutdown
93.         return true;
94.     }
95. }
96. }
97. // Check if the signal is within the acceptable range and update quality bit
98. static void checkSignalRange(float *value, bool *quality) {
99.     // Signal is out of range if it's below the lower limit or above the upper limit
100.    if (*value < LOWER_LIMIT || *value > UPPER_LIMIT) {
101.        *quality = false; // Set quality bit to false for out of range signal
102.    } else {
103.        *quality = true; // Set quality bit to true for signal within the range
104.    }
105. }
```

**Appendix B. Generated Code for CruiseControl Software**

1. Generated code based on CruiseControl’s overall requirements is shown in Listing A4:
Listing A4. Generated code based on CruiseControl’s overall requirements.

```c
#include <stdio.h>

// CC parameters
#define MIN_SPEED 20
#define MAX_SPEED 120
#define SPEED_INCREMENT 5
#define MAX_THROTTLE_SATURATION 100
#define MIN_PEDAL_PRESS_DETECTION 10

// CC states
enum CCState { OFF, ON, STDBY};

// CC variables
int cruiseSpeed = 0;
int carSpeed = 0;
int throttleCommand = 0;
enum CCState ccState = OFF;
int acceleratorPedal = 0;
int brakePedal = 0;

// Function to update CC state
void updateCCState()
{
    if (ccState == OFF) {
        if (carSpeed >= MIN_SPEED && carSpeed <= MAX_SPEED && acceleratorPedal < MIN_PEDAL_PRESS_DETECTION) {
            ccState = ON;
            cruiseSpeed = carSpeed;
        }
    }
    else if (ccState == ON) {
        if (acceleratorPedal >= MIN_PEDAL_PRESS_DETECTION || carSpeed < MIN_SPEED || carSpeed > MAX_SPEED) {
            ccState = STDBY;
        }
        else if (brakePedal >= MIN_PEDAL_PRESS_DETECTION) {
            ccState = OFF;
        }
    }
    else if (ccState == STDBY) {
        if (acceleratorPedal < MIN_PEDAL_PRESS_DETECTION && carSpeed >= MIN_SPEED && carSpeed <= MAX_SPEED) {
            ccState = ON;
        }
    }
}

// Function to update throttle command
void updateThrottleCommand()
{
    if (ccState == ON) {
        throttleCommand = (cruiseSpeed - carSpeed) * MAX_THROTTLE_SATURATION / (MAX_SPEED - MIN_SPEED);
        if (throttleCommand > MAX_THROTTLE_SATURATION) {
            throttleCommand = MAX_THROTTLE_SATURATION;
        }
    }
    else {
        throttleCommand = 0;
    }
}

// Main function
int main()
{
    // Simulate car and driver actions
    carSpeed = 50;
    acceleratorPedal = 0;
    brakePedal = 0;
    updateCCState();
    updateThrottleCommand();
    printf("CC State: %d, Cruise Speed: %d, Throttle Command: %d\n", ccState, cruiseSpeed, throttleCommand);
    return 0;
}
```

(2) Generated code based on CruiseControl’s specific requirements is shown in Listing A5:
Listing A5. Generated code based on CruiseControl's specific requirements.

```c
1. #include <stdbool.h>
2. #include <stddef.h>
3. // Input signal names and sources
4. typedef enum {
5.     CC_OFF,  
6.     CC_ON,  
7.     CC_RESUME,  
8.     CC_SET,  
9.     CC_QUICK_DECEL,  
10.    CC_QUICK_ACCEL  
11. } CC_Commands;
12. 
13. typedef struct {
14.     CC_Commands command;
15.     float speed;
16.     float accel;
17.     float brake;
18. } CC_Input;
19. 
20. // Output signal names
21. typedef struct {
22.     float cruise_speed;
23.     bool throttle_cmd;
24.     enum {
25.         CS_OFF, CS_ON, CS_STDBY, CS_INT
26.     } cruise_state;
27. } CC_Output;
28. 
29. // Function prototypes
30. void cruise_control(CC_Input input, CC_Output *output);
31. void cruise_speed_management(CC_Input input, CC_Output *output);
32. void car_driving_control(CC_Input input, CC_Output *output);
33. void pedals_pressed_detection(CC_Input input, CC_Output *output);
34. 
35. int main()
36. {
37.     CC_Input input;
38.     CC_Output output;
39.     // Update input signals here
40.     cruise_control(input, &output);
41.     return 0;
42. }
43. 
44. // Main cruise control function
45. void cruise_control(CC_Input input, CC_Output *output) {
46.     cruise_speed_management(input, output);
47.     car_driving_control(input, output);
48.     pedals_pressed_detection(input, output);
49. }
50. 
51. // Cruise speed management functionality
52. void cruise_speed_management(CC_Input input, CC_Output *output) {
53.     const float SpeedMin = 30.0;
54.     const float SpeedMax = 150.0;
55.     const float SpeedInc = 2.5;
56.     if (output->cruise_state == CS_ON || output->cruise_state == CS_STDBY ||
57.         output->cruise_state == CS_INT) {
58.         if (input.command == CC_SET) {
59.             output->cruise_speed = input.speed;
60.         } else if (input.command == CC_QUICK_ACCEL) {
61.             output->cruise_speed = (output->cruise_speed + SpeedInc <= SpeedMax) ?
62.                 output->cruise_speed + SpeedInc : SpeedMax;
63.         } else if (input.command == CC_QUICK_DECEL) {
64.             output->cruise_speed = (output->cruise_speed - SpeedInc >= SpeedMin) ?
65.                 output->cruise_speed - SpeedInc : SpeedMin;
66.         }
67. }  
```
Generated code based on software requirements through augmented prompt is shown in Listing A6:
Listing A6. Generated code based on CruiseControl's requirements through augmented
prompt.

```c
#include <stdbool.h>
#include <stdint.h>

// Constants for cruise control system
#define SPEED_MIN 30.0f // Minimum cruise speed
#define SPEED_MAX 150.0f // Maximum cruise speed
#define SPEED_INC 2.5f // Speed increment/decrement for quick accel/decel
#define KP 0.113f // Proportional gain
#define KI 0.5f // Integral gain
#define THROTTLE_SAT_MAX 45.0f // Maximum throttle saturation value
#define PEDALS_MIN 3.0f // Minimum pedal value to detect pedal press

typedef enum {CRUISE_OFF, CRUISE_ON, CRUISE_STDBY, CRUISE_INT} CruiseState;

// Car driving control function calculates throttle command based on speed error and integral
static float car_driving_control(float speed_error, float *integral, bool is_on, bool is_saturated)
{
    if (!is_on)
    {
        // Reset integral when cruise control is off
        *integral = 0.0f;
        return 0;
    }
    if (!is_saturated)
    {
        // Update integral if throttle is not saturated
        *integral += speed_error;
    }
    // Calculate control output using proportional and integral gains
    float output = (KP * speed_error) + (KI * (*integral));
    return output;
}

// Cruise speed management function updates cruise speed based on user inputs
static float cruise_speed_management(float current_speed, CruiseState state, bool set, bool quick_acc, bool quick_decel)
{
    static float cruise_speed = 0.0f; // Maintain the current cruise speed

    if ((state != CRUISE_ON) && (state != CRUISE_STDBY) && (state != CRUISE_INT))
    {
        return cruise_speed;
    }
    // Update cruise speed according to user inputs
    if (set)
    {
        cruise_speed = current_speed; // Set cruise speed to current speed
    }
    else if (quick_acc && (cruise_speed + SPEED_INC) <= SPEED_MAX)
    {
        cruise_speed += SPEED_INC; // Increase cruise speed
    }
    else if (quick_decel && (cruise_speed - SPEED_INC) >= SPEED_MIN)
    {
        cruise_speed -= SPEED_INC; // Decrease cruise speed
    }
    return cruise_speed;
}

// Pedals pressed detection function checks if a pedal is pressed based on its value
static bool is_pedal_pressed(float pedal_value)
{
    return pedal_value > PEDALS_MIN; // Returns true if pedal value is greater than // the minimum threshold
}

// Cruise control behavior function determines the new cruise control state based on // provided inputs
static CruiseState cruise_control_behavior(CruiseState current_state, bool on, bool off, bool resume, bool brake_pressed, bool accel_pressed, float speed)
{
    CruiseState new_state = current_state; // Initialize new_state with current_state
```
Appendix C. The Test Code for Robustness Testing

Listing A6. Generated code based on CruiseControl’s requirements through augmented prompt.

```c
79. // Update cruise control state based on inputs
80. if (off)
81. { new_state = CRUISE_OFF;
82. }
83. else if (on)
84. { new_state = CRUISE_ON;
85. }
86. else if (current_state == CRUISE_ON)
87. { if (accel_pressed || (speed < SPEED_MIN) || (speed > SPEED_MAX))
88. { new_state = CRUISE_STDBY;
89. }
90. }
91. else if ((current_state == CRUISE_STDBY) && (accel_pressed) &&
92. (speed >= SPEED_MIN) && (speed <= SPEED_MAX))
93. { new_state = CRUISE_ON;
94. }
95. else if (brake_pressed)
96. { new_state = CRUISE_INT;
97. }
98. else if (current_state == CRUISE_INT) resume)
99. { if (!accel_pressed && (speed >= SPEED_MIN) && (speed <= SPEED_MAX))
100. { new_state = CRUISE_ON;
101. }
102. else
103. { new_state = CRUISE_STDBY;
104. }
105. return new_state;
106. }
107. // Top-Level function for Cruise Control
108. void cruise_control(const bool CC_INR_01, const bool CC_INR_02, const bool CC_INR_03, const bool CC_INR_04, const bool CC_INR_05, const float CC_INR_06,
109. const float CC_INR_07, const float CC_INR_08, float *CC_OUTR_01, bool *CC_OUTR_02, CruiseState *CC_OUTR_03)
110. {
111. // Manage cruise control behavior based on inputs
112. CruiseState new_state = cruise_control_behavior(*CC_OUTR_03, CC_INR_01, !CC_INR_01, CC_INR_02, is_pedal_pressed(CC_INR_07), is_pedal_pressed(CC_INR_06), CC_INR_08);
113. // Manage cruise speed based on inputs and current cruise control state
114. float cruise_speed = cruise_speed_management(CC_INR_08, new_state, CC_INR_03, CC_INR_05, CC_INR_04);
115. // Calculate speed error between the current speed and desired cruise speed
116. float speed_error = cruise_speed - CC_INR_08;
117. // Determine if throttle is saturated (reached maximum limit)
118. bool is_saturated = *CC_OUTR_02 >= THROTTLE_SAT_MAX;
119. // Calculate throttle command based on speed error, integral, cruise control state, and saturation
120. static float integral = 0.0f;
121. float throttle_cmd = car_driving_control(speed_error, &integral, new_state == CRUISE_ON, is_saturated);
122. // Update output signals with calculated values
123. *CC_OUTR_01 = cruise_speed; // Output cruise control
124. *CC_OUTR_02 = throttle_cmd; // Output throttle command
125. *CC_OUTR_03 = new_state; // Output updated cruise control state
126. }
```
## Appendix C. The Test Code for Robustness Testing

### Listing A7. The test code for robustness testing.

```cpp
1. #include <stdio.h>
2. #include <stdbool.h>
3. #include <stdlib.h>
4. #include <time.h>
5. #include <assert.h>

6. // Function to generate a random float within a specified range
7. float generateRandomFloat(float min, float max) {
8.     float random = ((float)rand()) / (float)RAND_MAX;
9.     return min + (random * (max - min));
10. }

11. // Test case for boundary value testing
12. void boundaryValueTesting() {
13.     float inputs[4] = {LOWER_LIMIT - 1.0f, LOWER_LIMIT, UPPER_LIMIT, UPPER_LIMIT + 1.0f};
15.     bool reset = false;
16.     bool RT1, RT2;
17.     processNeutronFluxSignals(inputs, quality_bits, reset, &RT1, &RT2);
18.     // Verify the expected outputs
19.     assert(RT1 == true);
20.     assert(RT2 == true);
21. }

22. // Test case for random input testing
23. void randomInputTesting() {
24.     srand(time(NULL));
25.     float inputs[4];
26.     bool quality_bits[4];
27.     bool reset = false;
28.     bool RT1, RT2;
29.     for (int i = 0; i < 4; i++) {
30.         inputs[i] = generateRandomFloat(LOWER_LIMIT, UPPER_LIMIT);
31.         quality_bits[i] = true;
32.     }
33.     processNeutronFluxSignals(inputs, quality_bits, reset, &RT1, &RT2);
34.     // Verify the expected outputs
35.     assert(RT1 == true);
36.     assert(RT2 == true);
37. }

38. // Test case for exception input testing
39. void exceptionInputTesting() {
40.     float inputs[4] = {15.0f, -20.0f, 100.0f, atof("invalid")};
41.     bool quality_bits[4] = {true, true, true, true};
42.     bool reset = false;
43.     bool RT1, RT2;
44.     processNeutronFluxSignals(inputs, quality_bits, reset, &RT1, &RT2);
45.     // Verify the expected outputs
46.     assert(RT1 == true);
47.     assert(RT2 == true);
48. }

49. // Test case for security testing
50. void securityTesting() {
51.     float inputs[4] = {0.000000.0f, -5000.0f, 15000.0f, 30000.0f};
52.     bool quality_bits[4] = {true, false, true, true};
53.     bool reset = false;
54.     bool RT1, RT2;
55.     processNeutronFluxSignals(inputs, quality_bits, reset, &RT1, &RT2);
56.     // Verify the expected outputs
57.     assert(RT1 == false);
58.     assert(RT2 == false);
59. }
```

Listing A7. The test code for robustness testing.

```c
85.     assert(RT1 == true);
86.     assert(RT2 == true);
87. }
88. 
89. int main()
90. {
91.     boundaryValueTesting();
92.     exceptionInputTesting();
93.     randomInputTesting();
94.     forcedErrorTesting();
95.     securityTesting();
96.     printf("All tests passed successfully!\n");
97. 
98.     return 0;
99. }
```

Appendix D. The Standardized XML for Generalized Requirements

(1) The standardized XML template for generalized requirements is shown in Listing A8.

Listing A8. The standardized XML template for generalized requirements.

```xml
<requirements>
<input_signals>
<!-- Input signals -->
<signal>
<req_id></req_id>
<source></source>
</signal>
</input_signals>
<input_signal_types>
<!-- Input signal types -->
<signal_type>
<req_id></req_id>
<type></type>
<channels>
</channels>
</signal_type>
</input_signal_types>
<input_signal_ranges>
<!-- Input signal ranges -->
<signal_range>
<req_id></req_id>
<limits>
<lower_limit></lower_limit>
<upper_limit></upper_limit>
</limits>
<channels>
</channels>
</signal_range>
</input_signal_ranges>
<output_signals>
<!-- Output signals -->
<signal>
<req_id></req_id>
<destination></destination>
</signal>
</output_signals>
<output_signal_types>
<!-- Output signal types -->
<signal_type>
<req_id></req_id>
<type></type>
</signal_type>
</output_signal_types>
<output_signal_ranges>
<!-- Output signal ranges -->
<signal_range>
<req_id></req_id>
</signal_range>
</output_signal_ranges>
```

The standardized XML generalized requirements of the NucRate algorithm are shown in Listing A9.

(2) The standardized XML generalized requirements of the NucRate algorithm are shown in Listing A9.
Listing A9. The standardized XML generalized requirements of the NucRate algorithm.

```xml
<requirements>
  <input_signals>
    <signal>
      <req_id>RNI_INR_01</req_id>
      <source>Positive rate of change of neutron flux from four measurement channels, RNI1, RNI2, RNI3, RNI4.</source>
    </signal>
    <signal>
      <req_id>RNI_INR_03</req_id>
      <source>External reset signal, RNI_RST.</source>
    </signal>
  </input_signals>

  <input_signal_types>
    <signal_type>
      <req_id>RNI_INR_03</req_id>
      <type>Analog</type>
      <channels>
        <channel>RNI1</channel>
        <channel>RNI2</channel>
        <channel>RNI3</channel>
        <channel>RNI4</channel>
      </channels>
    </signal_type>
    <signal_type>
      <req_id>RNI_INR_05</req_id>
      <type>Digital</type>
      <channels>
        <channel>RNI_RST</channel>
      </channels>
    </signal_type>
  </input_signal_types>

  <input_signal_ranges>
    <signal_range>
      <req_id>RNI_INR_06</req_id>
      <limits>
        <lower_limit>10</lower_limit>
        <upper_limit>10000</upper_limit>
      </limits>
      <channels>
        <channel>RNI1</channel>
        <channel>RNI2</channel>
        <channel>RNI3</channel>
        <channel>RNI4</channel>
      </channels>
    </signal_range>
  </input_signal_ranges>

  <output_signals>
    <signal>
      <req_id>RNI_OUTR_01</req_id>
      <destination>Dedicated location A</destination>
    </signal>
    <signal>
      <req_id>RNI_OUTR_02</req_id>
      <destination>Shutdown circuit breaker</destination>
    </signal>
  </output_signals>
</requirements>
```
Listing A9. The standardized XML generalized requirements of the NucRate algorithm.

```xml
<output_signal_types>
  <signal_type>
    <req_id>RNI_OUTR_01</req_id>
    <type>Digital</type>
  </signal_type>
  <signal_type>
    <req_id>RNI_OUTR_02</req_id>
    <type>Digital</type>
  </signal_type>
</output_signal_types>
<output_signal_ranges>
  <signal_range/>
</output_signal_ranges>
<program_parameters/>
<functional_requirements>
  <requirement>
    <req_id>RNI_FUNR_01</req_id>
    <description>High signals of positive rate of change of neutron flux from the four channels are first checked against the signal range. Signals that exceed the range will have their corresponding quality bits set to bad.</description>
  </requirement>
  <requirement>
    <req_id>RNI_FUNR_02</req_id>
    <description>High signals of positive rate of change of neutron flux from the four channels are compared with a hysteresis threshold comparator to generate action signals.</description>
  </requirement>
  <requirement>
    <req_id>RNI_FUNR_03</req_id>
    <description>The hysteresis threshold comparator is a threshold comparator with hysteresis function. The hysteresis high threshold is 100, and the hysteresis low threshold is 95.</description>
  </requirement>
  <requirement>
    <req_id>RNI_FUNR_04</req_id>
    <description>The action signal is input to the S terminal of an RS flip-flop, and the external reset signal is input to the R terminals of the four flip-flops.</description>
  </requirement>
  <requirement>
    <req_id>RNI_FUNR_05</req_id>
    <description>The RS flip-flop has S priority.</description>
  </requirement>
  <requirement>
    <req_id>RNI_FUNR_06</req_id>
    <description>The output signals of the four RS flip-flops are input to a 4-to-2 voter. After the voting process, two shutdown signals are output. One shutdown signal is output to the dedicated location A, and the other shutdown signal is sent to the shutdown circuit breaker.</description>
  </requirement>
  <requirement>
    <req_id>RNI_FUNR_07</req_id>
    <description>The 4-to-2 voter triggers an action when at least two out of the four inputs are valid. The voter has a degradation feature where, if one input's quality bit is bad, it degrades to a 3-to-2 voter, and then to a 2-to-1 voter. When all quality bits of the signals are bad, the voter outputs an action signal.</description>
  </requirement>
</functional_requirements>
<generalized_domain_requirements>
Listing A9. The standardized XML generalized requirements of the NucRate algorithm.

<req_id>RNI_FUNR_07_01</req_id>
<description>The 4-out-of-2 voting logic takes action when at least two out of the four inputs are valid. However, the voting result is influenced by the quality bits of the inputs.</description>
</requirement>

<req_id>RNI_FUNR_07_02</req_id>
<description>The degradation of the voting quality bit function for the 4-out-of-2 voting logic refers to the following: when one input has a bad quality bit, it degrades to a 3-out-of-2 voting logic; when two inputs have bad quality bits, it degrades to a 2-out-of-1 voting logic; when three inputs have bad quality bits, it degrades to a 1-out-of-1 voting logic; when all signals have bad quality bits, the voting logic directly outputs the action signal.</description>
</requirement>

<constraints_requirements>
<requirement>
<req_id>RNI_NFUNR_01</req_id>
<description>Programming language constraint: C99 standard language.</description>
</requirement>

<requirement>
<req_id>RNI_NFUNR_02</req_id>
<description>Code standard limitation: Compliance with MISRA-C:2004 standard.</description>
</requirement>

<requirement>
<req_id>RNI_NFUNR_03</req_id>
<description>Code comment rate: Greater than 20%.</description>
</requirement>

<requirement>
<req_id>RNI_NFUNR_04</req_id>
<description>Program structure requirement (readability): Implement the hysteresis threshold comparator, 4-out-of-2 voting logic, and RS flip-flops as separate functions.</description>
</requirement>

<requirement>
<req_id>RNI_NFUNR_05</req_id>
<description>Safety and security requirement: Static memory allocation.</description>
</requirement>

<requirement>
<req_id>RNI_NFUNR_06</req_id>
<description>Safety and security requirement: No use of interrupts.</description>
</requirement>

<requirement>
<req_id>RNI_NFUNR_07</req_id>
<description>Performance requirements: Execution time less than 100us.</description>
</requirement>
</constraints_requirements>
Appendix E. The Detail Procedures for Conducting the Experiments

Listing A10. The detailed procedures for conducting the experiments.

(1) Prepare software domain requirements.
(2) Itemize software requirements and break them down into smaller granularity requirements.
(3) Translate software requirements into standardized XML requirements documents.
(4) Carefully analyze the functional requirements of the software and generalize content that contains domain-specific terminology. The transformed requirements are in common vocabulary.
(5) Provide the transformed software requirements as prompts to the GPT-4 dialog box and make it automatically generate code.
(6) Review the integrity of generated code and check that code generation for all requirements is complete.
(7) Focus on checking the code of the functional part of the domain requirements. In case of missing or incorrect code, modify the conversion content of the domain requirements to make its description clearer. The modified requirements are then provided as prompts to GPT-4 to regenerate the code.
(8) Repeat until you are satisfied with the generated code.
(9) Functional testing of the generated code.
(10) Domain compliance checking of the generated code.
(11) Robustness testing of the generated code.
(12) For the problems found in the test process, the problem is provided as a new prompt to GPT-4 to make targeted changes to the code.
(13) Repeat until GPT-4 solves all test problems.


**Disclaimer/Publisher’s Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.