

# LLM Assisted No-code HMI Development for Safety-Critical Systems

## Insights of a Short Impirical Study

Matthias Harter

Faculty of Engineering

Hochschule RheinMain - University of Applied Sciences

Rüsselsheim, Germany

e-mail: matthias.harter@hs-rm.de

**Abstract**—This paper represents the outcome of an empirical study conducted with Large Language Models (LLM) on the question whether or not we can expect current and future Artificial Intelligence to assist engineers in the development of embedded software systems for safety-critical applications. Experiments with GPT-4 and other LLMs suggest that current models are capable of *assisting* developers in part in the design of Human-Machine-Interfaces (HMI) for instruments and displays used in aircrafts without the need to manually write a single line of source code (no-code development) while maintaining the highest level of safety and reliability demanded by authorities and customers. The study does not present generally accepted quantitative measures in answering the question how well suited current language models are for this task, but rather provides a qualitative assessment of the capabilities of state of the art AI. It also sheds a light on the deficiencies of today's LLMs in fully understanding technical systems in depth. Instead of completely replacing human engineers we should rather strongly rely on human-in-the-loop policies for the most critical phase of the progressively automated development process, even with more sophisticated and powerful LLMs on the horizon. It should be noted that providing objective evidence to support the argumentation and the findings in this short paper will be the subject of future work.

**Keywords**—AI; requirements engineering; safety critical embedded software; model-based software design; automatic code generation.

## I. INTRODUCTION

This section briefly presents the development process of embedded software systems typically employed these days and problem with AI when used in safety critical applications. It is then argued that AI can still be used sensibly instead of doing without it completely.

### A. State of the Art

Outsourcing certain steps in the development process of software systems to the machine assistant has recently gained popularity, since it has shown to be beneficial with respect to coding tasks (e.g., GitHub Copilot). For many years prior to the advent of large language models like Chat-GPT and others, extensive automation of the development process of complex software systems has been the goal of many tools and procedures. For instance, the translation process from the source-code in a high-level programming language to the binary executable program (machine code or object code) with compilers like the GNU Compiler Collection (GCC) is a

fairly complex and demanding task and leads to solutions that one could attribute in a certain way to an intelligent agent, even though no AI algorithm or machine learning strategy has been traditionally employed of course. This observation is typically made by students of computers science when asked to thoroughly analyze the results of the compilation process for certain single statements in the C programming language by comparing them to the equivalent in the assembly language. The machine based translation of such statements is sometimes more efficient and elegant than the human counterpart, at least for assembly language beginners. Other tasks in the development process have been automated since many years as well, e.g., documentation generation by extracting information from comments within source code snippets (e.g., Doxygen) and static code analysis to name just a few.

Now that LLMs like GPT-4 [1], CodeLlaMa [2], StarCoder [3] or CodeGen [4] are capable of generating source code for dozens of different programming languages (depending on the model and the training), it seems logical to let the AI do the coding job, at least for small portions of a program. An supposedly increasing number of software engineers is using these models as a starting point for their software projects by diving the task into smaller modules which then get completed by hand in a repeated manner (module by module). Putting everything together and making the whole software system work as expected is certainly something that must still be done by human hand. A future seems possible, in which more and more of this work can be outsourced to AI agents (chat bots specialized in coding), not only for small modules or just as an assistant for code completion.

### B. LLMs and the problem with safety

The phenomenon of hallucination (better referred to as “confabulation”) raises concerns about reliability and trust into this development process, especially when targeted at safety-critical systems in aircrafts or other machinery that must be 100% safe (e.g., medical devices). For this reason, letting today's and even future AI write the code for safe-critical systems is not advisable, no matter how good they are or how promising the outcome will be. Even using LLM generated code for certain smaller parts of a safety-critical system is not recommended, mainly for practical reasons: The source code must be qualified/certified by authorities like the Federal

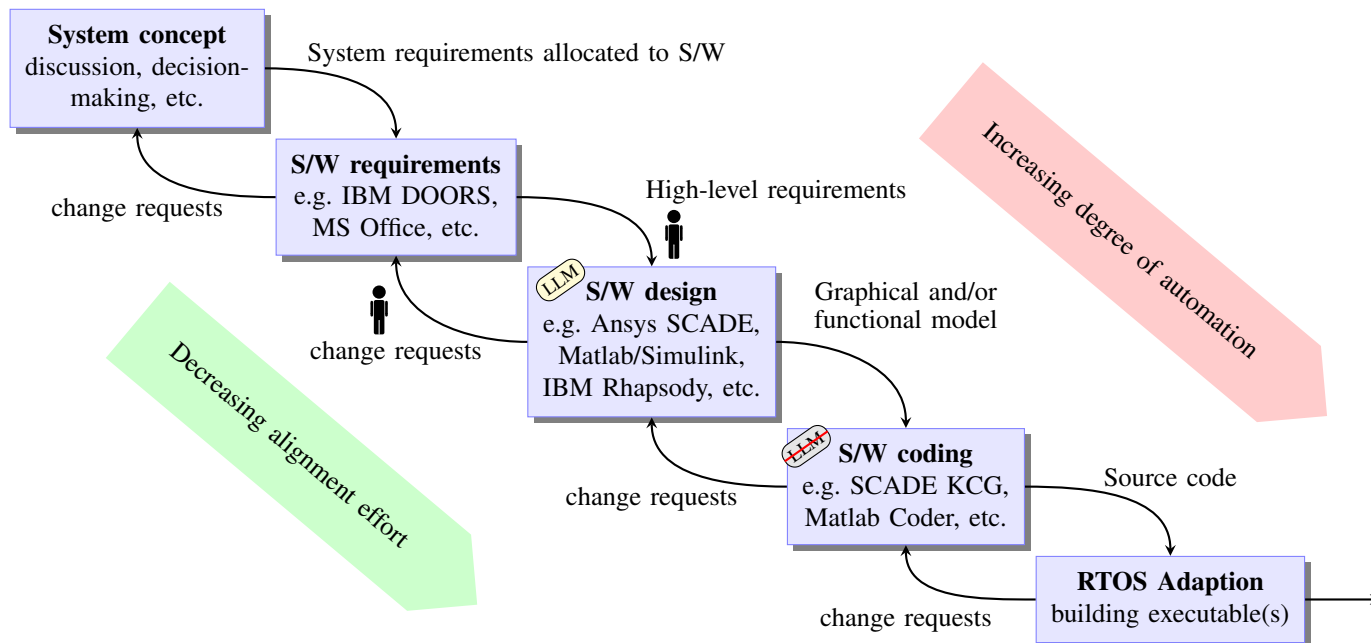


Figure 1. Development process of a safety-critical embedded software system according to DO-178C and ARP 4754A. The process resembles the left side of the famous V-model down to the implementation phase. To the right the process continues with integration, test and verification and validation (not shown).

Aviation Administration (FAA) in the US, the European Union Aviation Safety Agency (EASA) or other specialized institutions acting as certification instances. This certification process is tedious, expensive and ultimately superfluous. Instead of certifying the code every time for every software product or even every version of one and the same software product, we should move from the artifact (the software) to the process and the tools involved. If the tools in the development process are certified to be safe and to comply with safety standards and regulations (e.g., ISO 26262, DO-178C, etc.), we can rely on automation to a greater extent.

Unfortunately, such certification will be hard to achieve in case of LLMs writing source code due to their stochastic nature and the huge amount of training data that defines the capabilities and limits of the tool. The problem of confabulation stems from this design principle. In others word, neither the ever changing code generated by LLMs nor the LLM itself as a tool are good candidates for certification and the target to put trust into. Instead, we can solely rely on the code generation powers of specialized tools, such as Ansys SCADE Suite KCG and SCADE Display KCG, which produce C (ISO-C and MISRA-C [5][6] compliant) or ADA source code. According to the company’s website, Ansys SCADE Suite and SCADE Display KCG have been qualified to comply with all relevant safety standards.

### C. LLMs at the level of requirements

As argued in the previous subsection, we should not employ LLMs at the coding level in the development process. As their name “large language model” implies, this kind of AI is suited for interpreting natural language and for human-machine conversations (chats). In the development process

of embedded software systems natural language plays an important role at the very start of the process. Developers and management discuss the concept of the system, i.e., its capabilities, fundamental properties and limitations. At the end of this step, decisions have to be made and translated into a set of requirements that break down all characteristics of the system. In case of a software system, the requirements are a subset of the complete requirements, of course. In Figure 1, the whole process is depicted with the software requirements being the starting point for a possible integration and adaption of LLMs. The terms “requirements management” and “requirements engineering” represent the underlying tasks and procedures, which can be supported by specialized tools like IBM DOORS or just by simple office software (e.g., Microsoft Office). Most engineering activities in a company on these upper two levels are centered around requirements as more or less formal agreement upon fundamental properties of a system or product, which can get further refined (indicated by the arrows labeled “change request”), if not detailed enough, misleading or ambiguous. Engineers all over the world employ this methodology of requirements engineering as a means to establish a common understanding about the product to be developed. Of course, this understanding is achieved by natural language communication between humans. This step typically takes a significant amount of time and effort and eventually leads to a database of ideally precise, consistent, comprehensive and in terms of technicality detailed phrases in a natural language like English.

At the level of requirements, AI can come into play and support engineers taking the next step. Traditionally it was the developer’s job to translate all the requirements into a model of the system, either rather informal as a sketch inside the en-

gineer's brain or workbook or more formal as an architectural schematic, flowchart or state diagram. In the context of safety-critical applications in the mobility sector and other fields, it has become state of the art to use SysML/UML to model the system, using tools like Matlab Simulink, IBM Rational Rhapsody or Ansys SCADE Suite [7][8]. This modeling task can be quite sophisticated or even error-prone, if done by unexperienced engineers, but the hypothesis is that it can be aided by LLMs which play the role of an assistant. In this scenario the LLM is given the requirements as input (together with a certain prompt) and the output is a *first and basic version* of the model of the system (sub-system or module) to be developed. Here, the engineer is still needed not only to supervise this translation process, but also to edit, complete and essentially examine the model itself to see if it is properly aligned to the requirements defined by human. Change requests will still be needed frequently in order to keep humans in the loop and are essential to guarantee a system of check and balances.

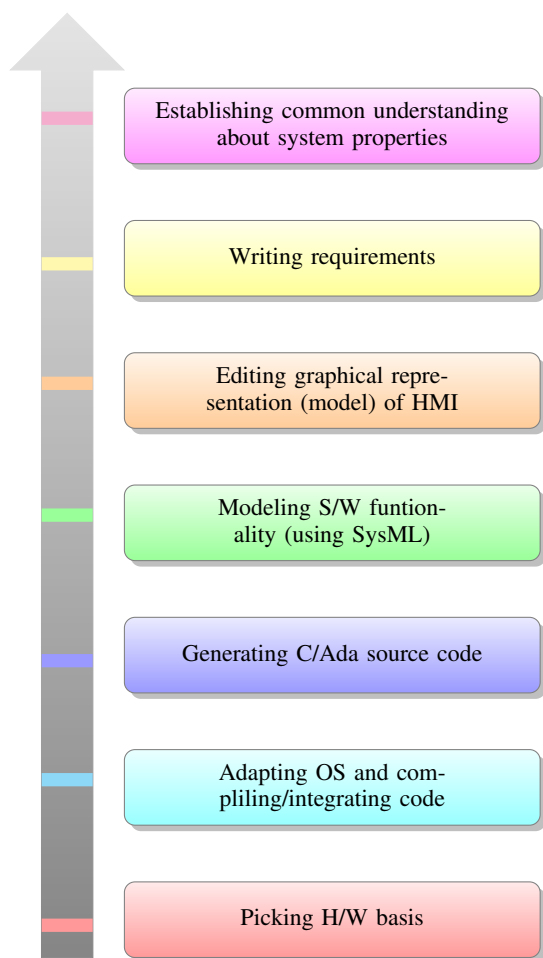


Figure 2. Leeway in decision-making for tasks in the development process of a safety-critical embedded software system. As the number of options decreases from top to bottom, the level of trust increases due to certified tools/products and proven engineering practices.

#### D. Alignment of automated processes with the human-centric perspective

One can observe a decreasing degree of alignment effort with human-formulated standards and specifications as automation is increasingly incorporated. Generally speaking, mankind should emphasize the importance of (wo)man-machine alignment at the highest level of each development process and will more and more get along with highly automated procedures and tools at the lower end of the process, even for safety-critical sectors. This increasing degree of automation does not mean that we should use LLMs to generate software code, but support us in the design or modeling phase while still keeping control of the model itself. This way engineers can focus on the human-centric viewpoint, i.e., defining what behavior is desirable, and let proven tools like traditional code generators (e.g., SCADE KCG) do the tedious and costly job. This observation is summarized in Figure 2 and further detailed for the use case which was studied for this paper.

At the top level of the whole process humans can choose between a vast variety of options and have to discuss and argue within a group and with the outside world about fundamental characteristics of the product. Writing down the requirements narrows the leeway in decision-making quite a bit, since many options and features turn out to be unfeasible, costly or otherwise undesired. In the case of a system with a Human-Machine-Interface (HMI) or User-Interface (UI) respectively, the requirements get broken down into a model of the graphical representation, which typically consists of a set of basic shapes (primitives like circles, rectangles, text and others) and their properties. This means that, again, many options are omitted and things get further concretized. Modeling the functionality in the next step using SysML is another way of narrowing the leeway. In case of the MBSE approach (Model-Based System Engineering) employed by the SCADE tool family, the SysML model needs to be technically precise and comply with certain modeling principles in order to use the SCADE Code Generator (KCG) for automated code generation. We have the choice between C or Ada and can steer the code generation process to a small extend, but the range of options is rather limited. At the bottom of this diagram we can see that compiling the source code and integrating it into a software ecosystem (usually a real-time OS/RTOS) is a matter of choosing between very few software products (e.g., VxWorks, INTEGRITY-178B or PikeOS).

The same applies to the hardware basis of the system. Obviously, the leeway of decision-making is so narrow that we cannot choose freely between all sorts of hardware and computing platforms, e.g., Raspberry Pi or Arduino. We have to get along with what has been proven to fulfill the highest standards of safety. Specialized hardware offers features and certain safety measures to ensure this (e.g., lockstep mode of operation, majority vote principle, watchdog timers, etc.).

## II. METHODOLOGY

In this section, the overall scenario is presented and the procedure for investigating the possible applications of AI in the development process of safety-critical embedded software is explained.

### A. Use case

In the scope of this work, the following scenario has been studied: Large language models which have been trained extensively with programming language code (OpenAI GPT-3.5, GPT-4 [1], Salesforce CodeGen [4], StarCoder [3] and CodeLlama [9]) are instructed to take the requirements for the display (HMI) of an aircraft instrument as input prompt (together with the system prompt, if applicable) and translate them into a model that is fed into the Ansys SCADE development environment. The model should reflect the requirements as precisely as possible and thus demonstrate the capabilities in understanding natural language from the standpoint of a technical assistant. The aircraft instrument to model is a so-called Primary Flight Display (PFD), which is used by pilots as an indication of the aircraft's attitude in relation to the horizon. This instrument is therefore also called "artificial horizon". It usually also provides information about the aircraft's speed and its altitude (above mean sea level).

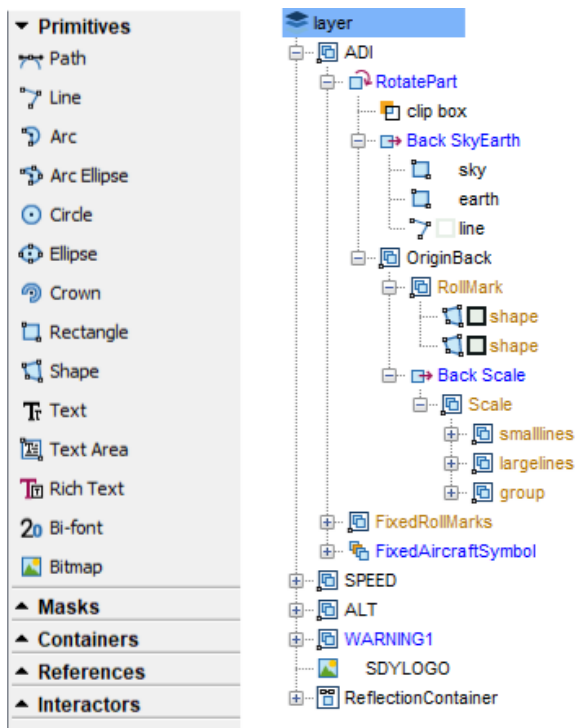


Figure 3. Hierarchy of layers (on the right) with graphical primitives like lines and groups of shapes in SCADE Display for the PFD in Figure 4. These shapes originate from the list of primitives on the icon bar on the left.

### B. Limitations

Two limitations apply to this use case as it has been tested for this paper:

- the model is limited to the visual display, i.e., no functionality should be part of the model
- the model should be written in Python using an appropriate UI framework or in PGF/TikZ (i.e.,  $\LaTeX$ ).

The latter is a limitation that results from the limitation of the LLMs type of output. Even though GPT-4 has some multi-modal capabilities and can also process images as input, it cannot produce SysML models directly, but only text-based descriptions of a model. In SCADE and other tools such SysML models look like schematics in the editor of the tool, but the underlying database is saved in XML format. By this means, LLMs could output the model as an XML file, rather than outputting a graphics file representing the SysML model. For this to work, a prerequisite would be that the LLM has been trained for outputting XML and using domain-specific (tool specific) designators, naming rules and other rules to obey. It seems to be plausible that such training could be accomplished in principle. However, there are much simpler ways to connect the LLMs to the MBSE development environment: In Ansys SCADE, users can write Python code to automate all kinds of tasks using an Application Programming Interface (API). This way the LLM can hook into the development environment and generate the model directly without any file based detour. Using this API simplifies the modeling task for an AI that cannot generate images or draw schematics on its own. For the scope of this paper, even using the API would not be feasible, since using the API would afford pre-training or at least fine-tuning of the LLM. Instead, the LLMs was instructed to use Python and let it generate the graphics with the help of an appropriate framework like Qt or Tkinter that contain all visual elements (widgets) needed. As an alternative, also PGF/TikZ instructions for processing with  $\LaTeX$  have been proposed to the LLM.

The first limitation further simplifies the task for the LLMs: In order to generate a graphical model of the HMI of the PFD, no knowledge of SysML is needed. Instead, the LLM can use the API in SCADE Display to instantiate graphical primitives from a palette of basic shapes and edit their graphical properties and appearance (see Figure 3). SCADE Display also allows to implement basic functionality using logic expressions and setting properties of the shapes (e.g., visibility, color or text strings) in a way that resembles conditional statements in a programming language, but this feature cannot be known to the LLM without proper training and thus was not expected to be used by the LLM in this study.

### C. Example

Figure 4 shows an example of an PFD as provided by SCADE Display for demonstration purposes. It contains some of the elements typically associated with the digital version of an artificial horizon (so-called "glass cockpit"), but there is no standardized layout and no mandatory information to be shown besides the horizon and scales for the attitude of the airplane. In those days of analog cockpits the artificial horizon did not provide information about altitude and speed

and was far less cluttered with additional flight information and warning/caution indicators. Nowadays a variety of such information can be found additionally on a digital PFD.

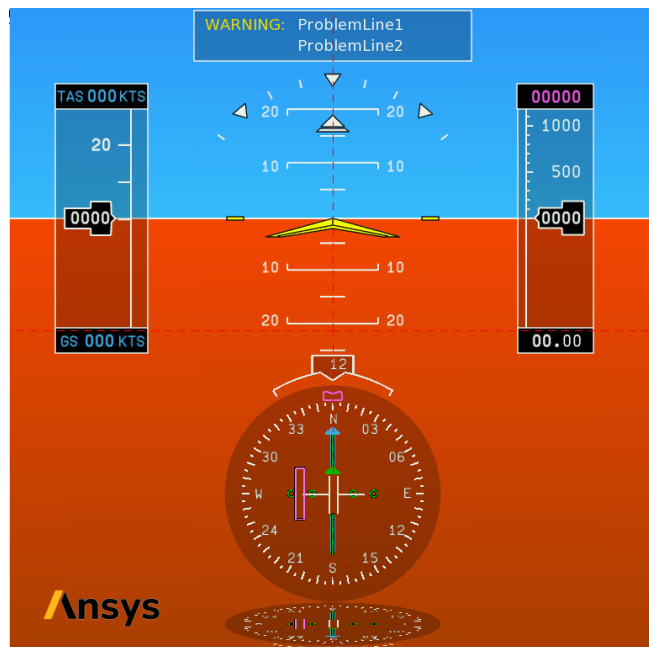


Figure 4. Graphical model of the PFD in SCADE Display as provided by the tool as an example.

In this paper, the example in Figure 4 serves as a benchmark for evaluating the degree of precision the LLMs achieve when comparing it to each generated display model. As stated in the beginning of this paper, no generally accepted quantitative measures can be used to rate the quality of the outcome. The evaluation is based on the visual assessment on how closely related the generated results are if compared to the PFD example and serves as a performance *estimate*. Generally accepted benchmarks for evaluating the performance of LLMs like HELM [10] or ARC [11] are not suited for this kind of evaluation. Even the very broad and comprehensive collection of tests and benchmarks BIG-bench [12] does not provide a method of measuring the model building capabilities from requirements as needed for this paper. In [13] the author examines over 100 benchmarks for commonsense reasoning in AI. His conclusion is that many of them are incomplete or contain flaws. As of today, there is no proven method of measuring the skills of LLMs reliably (moreover, commonsense-reasoning is different from model-building).

#### D. Requirements

The starting point for the LLMs is the set of requirements that specify the display of the PFD HMI as described in the previous subsection. These requirements should be made by the human engineer, as depicted in Figure 1. These requirements have been tailored to more or less match the design and structure of the PFD example in Figure 4. It represents the instrument in the supposedly simplest form and omits those

types of information which may be specific to certain models of aircraft or manufacturers.

a) *Full list of requirements:* This list has been used for GPT-3.5 and GPT-4, but not in full length for CodeLlama, StarCoder and CodeGen. Only OpenAIs leading-edge products could use such a long list of requirements as *single, contiguous* input (together with the system prompt or instruction). For the other LLMs a shorted version has been used (see below).

##### 1) General Layout & Dimensions:

- The PFD shall have a rectangular aspect ratio suitable for installation in standard cockpit instrument panels.
- The sky and earth shall be perfectly aligned at the horizon line.
- The horizon line shall be centered horizontally on the PFD, and its vertical placement shall adjust based on the aircraft's pitch angle.

##### 2) Color and Appearance:

- The PFD shall represent the sky in blue.
- The PFD shall represent the earth in brown.
- The horizon line shall be a distinct, bold white line for easy visibility against both the sky and earth backdrops.

##### 3) Aircraft Attitude Indicator:

- An aircraft symbol, representing the relative pitch and roll of the aircraft, shall be fixed centrally on the PFD.
- The aircraft symbol shall be displayed in a contrasting color (e.g., white) to ensure it is distinct against both sky and earth.

##### 4) Altitude Tape:

- The PFD shall display an altitude tape vertically on the right side, showing the current altitude of the aircraft.
- The altitude values shall be displayed in white digits with a black outline for easy readability.
- An arrow or pointer shall indicate the current altitude on the tape.

##### 5) Airspeed Tape:

- The PFD shall display an airspeed tape vertically on the left side, showing the current airspeed of the aircraft.
- The airspeed values shall be displayed in white digits with a black outline.
- An arrow or pointer shall indicate the current airspeed on the tape.

##### 6) Heading Indicator:

- The PFD shall display a horizontal heading tape or compass rose at the bottom of the display.
- The current heading shall be indicated by a fixed pointer or triangle, with the tape/rose rotating behind it.

##### 7) Turn Coordinator:

- The PFD shall incorporate a turn coordinator, represented by a curved line or other suitable graphical representation, to show the rate and direction of turn.

##### 8) Additional Flight Information:

- The PFD shall display other pertinent flight data such as vertical speed, angle of attack, and barometric pressure.
- This information should be arranged in a manner that does not clutter the primary attitude information.

## 9) Warning and Caution Indicators:

- The PFD shall have provisions for displaying warning (red) and caution (amber) indications for critical flight parameters, such as stall warnings or autopilot disengagement.

*b) Short list of requirements:* The full list of requirements was too long for CodeLlama, StarCoder and CodeGen. CodeLlama did not finish the code generation process properly and stopped the output in the middle of the code - unfinished and not ready to run on the Python interpreter. StarCoder and CodeGen did not output anything, the process stopped with a time-out error. For this reason, a short list of requirements was used. This way, the modeling task was shorter also and could be finished with less tokens for the output. The short list consists of all requirements from above up to and including requirement no. 5.

### E. Prompt engineering

Of course, the raw list of requirements is not enough to instruct LLMs to generate any code. Even though the requirements made up the greatest part of the input prompt, the LLMs needed to get instructions on how to code and in what language. The prompt was also used to describe the scenario and the role the LLM was expected to play when generating the code. The prompt was therefore separated into a first part which was labeled as “Instruction” and a second part which was named “Requirements”. Such labeling and structuring is considered to be good practice and generally improves the outcome. Substantially better results could be expected, if instead of this “zero-shot learning” approach, at least a single example of the code to be generated would be presented to the LLM (“few-shot learning”) as part of the input prompt [14]. This would have meant that a corpus of instructions was used along with an example of HMI related graphic routines. However, it was the focus of this work to only study the potential of LLMs in understanding typical (traditional) requirements in natural language and to add only a minimal amount of instructions beforehand (often referred to as “system prompt”). The following paragraph depicts the input prompt used:

```
### Instruction ###
You are a software developer who writes
code for the user interface of a Primary
Flight Display (PFD) used in an airplane's
cockpit.
Your language of choice is Python. Use
the following list of requirements as
a specification of the properties and
appearance of the user interface.
All requirements must be met. Output the
code for generating the graphics of the
user interface.
### Requirements ###
...
```

Minor changes to this input prompt were used occasionally, e.g., to instruct the LLM to use a different programming language instead of Python. For instance, GPT-4 was asked to output the code for the HMI of the primary flight display using the TikZ package of  $\text{\LaTeX}$ . The advantage of this variation was, that it was perfectly clear that only static code to generate the visuals was asked for, instead of functional code that would compute changes in the aircraft's attitude from sensor inputs. In fact, the code generated by CodeLlama for the right PFD in Figure 7 comprises function calls like `getPitchAngle()`, `getRollAngle()`, `getAltitude()` and `getAirspeed()`, which are supposed to provide sensor data from real-time measurements. For the complete code, refer to listing 10 in the appendix. Such functional code was not part of the assignment and therefore the Python interpreter aborts execution after drawing the basic layout of the display, thus omitting any adjustments to be made to the indicated attitude of the aircraft or changes in altitude or speed.

No instruction finetuning was used to further improve the outcome. Chain-of-thought finetuning was also not employed, even though it should lead to substantially better results [15]–[20], given that the task of model building from requirements requires engineers to also think in a “divide and conquer” fashion and the build the system step-by-step from bottom-up. A single requirement (single sentence in natural language) in this way could be quite challenging and sophisticated and require many complex technical considerations, but be still quite feasible, if divided conceptually into sub-tasks and solved sequentially.

## III. FINDINGS

This section explains the results of a comparison of the suitability of different language models for the use case presented.

### A. GPT-3.5 and GPT-4 are ahead

GPT-3.5 and GPT-4 could handle the full list of requirements, whereas the other LLMs tested in this work failed. Besides these two, only CodeLlama (CodeLlama-34b-Instruct) could at least handle the shorter list of requirements, primarily due to the limited context length of the LLM (4096 tokens for CodeLlama, see [2]).

The StarChat LLM is advertised to be the “fine-tuned versions of the StarCoder family to act as helpful coding assistants” (taken from Hugging Face website). And further: “The base model has 16B parameters and was pretrained on one trillion tokens sourced from 80+ programming languages.” As StarCode offers a context window of 8K tokens [3], it was expected to actually generate some code, irrespective of the quality and the achievements. The same applies to CodeGen (codegen25-7b-instruct) from Salesforce [4]. However, running the models on Hugging Face playground led to extremely long runtimes and eventually was aborted on the server side. For this reason, the table shows 0% fulfillment rate. It remains unclear of these two LLMs would be able to process the full

(or shorted) list of requirements if run on a dedicated, powerful server. Answering this question is left for future work.

In a paper titled “Sparks of Artificial General Intelligence: Early experiments with GPT-4” [21] the authors examine the capabilities of GPT-4 in graphical user interface programming. They claim that “... GPT-4 is also an expert in GUI programming, knowing how to create an accurate layout and handle complicated input events”.

With this in mind, expectations were high that at least the leading-edge LLM GPT-4 could satisfactorily fulfill the task of translating requirements written in natural language into programming language code for the static display of HMI showcases. As Table I shows for the full list of requirements listed in the preceding section, GPT-4 is indeed capable of completing this task in a way that it can *assist* a human engineer in building a first, basic graphic model of a HMI for further processing with tools like SCADE Display and subsequent code generation with SCADE Display KCG or similar. In Figure 5, two of the best results are shown. The code for the PFD on the right is given in Figure 8 (code for the other on the left omitted to save space).

The metrics in Table I should be understood as meaning that the respective language model was used for several runs under the same conditions, resulting in different code variants for each run. These were then analyzed in terms of their degree of fulfillment and the dispersion characterized by the lower and upper bounds as well as the median.

The numbers indicate that in at least one case GPT-4 could successfully meet all requirements listed in Section II-D and the minimum number of requirements that could be satisfied is twice as high as in the case of GPT-3 (37% vs. 16%). The median is also almost twice as high and GPT-4 always produced code that could be run by the interpreter (Python or  $\LaTeX$ ) right from the start (no code fiddling needed). GPT-3.5 produced code that was erroneous in one case, but it could be corrected by the LLM itself after being instructed to do so.

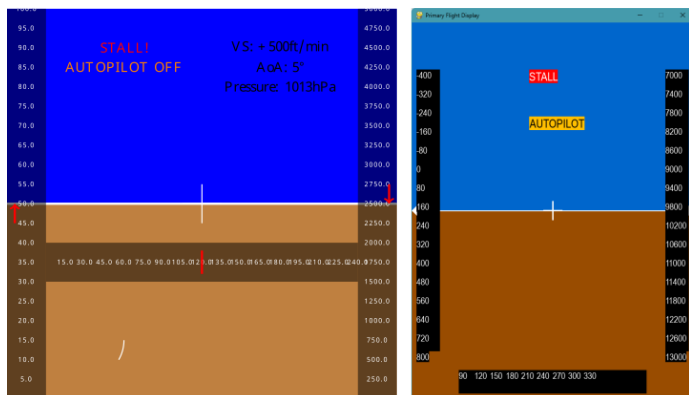


Figure 5. PFD with highest degree of fulfillment (100%, left) for the full list of requirements as generated by GPT-4 using the TikZ ( $\LaTeX$ ) language (see Table I). The PFD on the right side achieved 84% fulfillment.

It should be noted that the variability of the code is quite high considering all instances, despite the fact that no input

TABLE I  
EVALUATION OF LLMs FOR THE FULL LIST OF REQUIREMENTS

	Full list of requirements (no. 1 to 9)		
	Degree of fulfillment	# of error-free code variants	# of correctable code variants
GPT-4	Min. 37% Median 74% Max. 100%	14 of 14	N/A
GPT-3.5	Min. 16% Median 39% Max. 68%	7 of 8	1 of 8 <sup>a</sup>
CodeLlama	0%	0 of 2 <sup>b</sup>	0 of 2 <sup>b</sup>
StarChat	0%	0 of 2 <sup>c</sup>	0 of 2 <sup>c</sup>
CodeGen2.5	0%	0 of 2 <sup>c</sup>	0 of 2 <sup>c</sup>

<sup>a</sup>contained errors that GPT-3.5 corrected after being instructed

<sup>b</sup>code output ended after approx. 5000 characters

<sup>c</sup>timeout after several minutes without any output

prompt changes had been made and the requirements also were kept untouched. The LLMs was presented one and the same input repeatedly and the code was analyzed by comparing each and every requirement to what could actually be seen on the display of the (virtual) instrument when the code was executed. The number of satisfied requirements on the *static* display was counted and led to the percentage measure.

#### B. CodeLlama: Shorted list of requirements

The shorted list of requirements comprises requirement no. 1 to no. 5 and represents a very basic PFD. By this means the number of code lines for the output was essentially reduced. This enabled CodeLlama to become part of the game, i.e. it could finish the code which was otherwise aborted. Interestingly, all six runs in which CodeLlama came into operation produced code with the same type of error concerning the proper usage of the UI framework Qt. Correcting this error required the manual replacement of a line of code with three additional lines.

The complete code for the PFD on the left side of Figure 6 is given by the listing in Figure 9 in the appendix. It represents the original code from CodeLlama without the corrections. The same Qt related error produced CodeLlama in all six instances of output in Table II and could be resolved analogously.

TABLE II  
EVALUATION OF LLMs FOR THE SHORT LIST OF REQUIREMENTS

	Short list of requirements (no. 1 to 5)		
	Degree of fulfillment	# of error-free code variants	# of correctable code variants
GPT-4	100%	8 of 8	N/A
GPT-3.5	Min. 64% Median 84% Max. 96%	4 of 4	N/A
CodeLlama	Min. 14% Median 29% Max. 86%	0 of 6	6 of 6 <sup>a</sup>
StarChat	0%	0 of 2 <sup>b</sup>	0 of 2 <sup>b</sup>
CodeGen2.5	0%	0 of 2 <sup>b</sup>	0 of 2 <sup>b</sup>

<sup>a</sup>repeatedly the same error using Qt, but code corrected by GPT-4

<sup>b</sup>timeout after several minutes without any output

In the table it can be seen that CodeLlama is far less powerful in GUI programming (i.e., HMI generation from requirements) than GPT-3.5 and certainly GPT-4, with a median degree of fulfillment of 29% vs. 84% for GPT-3.5. GPT-4 could meet all requirements and produced error-free code, as expected.

### C. Visual examination and oddities

In Table II the noticeable spread between the best result from CodeLlama and the worst corresponds to the great variability that can be seen in the output of CodeLlama in Figure 6 and Figure 7. The Qt window on the left in Figure 6 is almost empty and seems to be a complete failure, but this stems from the fact that on the upper half of the window an image of the sky should be loaded from a file and in the lower half an image of the earth. The two Python instructions for loading these two files had been commented out, because they were not readily available. With those images included, the window would not look that defective and an artificial horizon would be noticeable at the boundary between the images.

Another point to mention is the aircraft symbol, which can hardly be seen in the upper center of the window. It is quite small and represented by the unicode symbol “rocket” (U+1F680), which was included in the source code as text string.

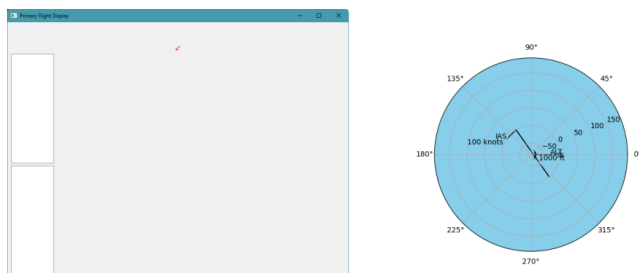


Figure 6. PFD with lowest degree of fulfillment (14%, left side) for the short list of requirements as generated by CodeLlama. CodeLlama also generated variants of the PFD on the basis of a polar diagram (29%, right side).

The variability of the results from CodeLlama is quite remarkable, as can be seen on the right side of Figure 6. In this case the LLM tried to use a polar diagram to fit the PFD in, but with this approach it sacrificed many requirements so that only 29% could be met.

In those cases in which CodeLlama “decided” to chose the right UI/GUI framework and a suitable graph paradigm, the outcome was not so bad as can be seen in Figure 7. On one hand, the aircraft symbol was merely a circle in the middle of the display, but the requirements did not specify how it should look like well enough on the other hand. It can be stated that *what is not specified thoroughly, precisely and comprehensively in the requirements can be implemented by the LLM modeling assistant freely and with little common-sense knowledge and engineering experience, it will be*. The more common-sense knowledge an advanced LLM has, the better it can fill those gaps in the specification and thereby interpret the human will and serve the intended purpose.

For this reason, CodeLlama generated the “rubber bands” (airspeed tape and altitude tape, requirements no. 4 and 5) on the left and right side in such a way that it is hard to use from a practical standpoint, but it also fulfilled the needs written down in the requirements. It was just lacking the knowledge, that a cluttered display with multiple symbols and text snippets overlaying each other is basically useless.

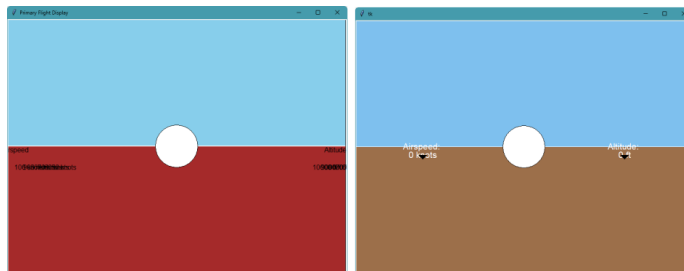


Figure 7. PFD variants with highest degree of fulfillment (57% and 86%) for the short list of requirements as generated by CodeLlama. Note that GPT-3.5 and GPT-4 were still much better (see Table II).

GPT-4, however, seems to be much more aware of the implications of certain design and layout related decisions it makes for the practical usability, as can be clearly seen in Figure 5. It knows that the altitude is typically displayed in quantities of 100 feet and the speed in a finer resolution (here 5 knots). The compass rose (heading tape) in PFD on the right correctly shows values of degree ranging from 90 to 330, but not surpassing 360 degree. All of this was not specified by the requirements.

Interestingly, GPT-4 did not realize that it would be better to let the altitude increase towards the sky and sketched the numbers the other way around (higher altitude towards the earth). A human engineer would have done it opposed for sure, even without a corresponding requirement. Such implicit attributes must be derived from the requirements as part of the engineering task. This is something that GPT-4 is not able to do in each and every case. Its common-sense knowledge is incomplete, otherwise GPT-4 would not provide numbers for negative speeds on the left airspeed tape. Numbers can be negative in many cases, but for speed it makes no sense. It is speculative, but maybe GPT-4 would not have included negative values if the tape and its purpose was titled “speed” instead of “airspeed”.

Taking the speculation one step further, a single example of a PFD with negative speed values can be found on the internet if a search (using Google) for images for “primary flight display” is performed. The image with negative speed values leads to GitHub: In 2019, under the name “kouky” an author published a project for a PFD to be used in micro UAVs (Unmanned Aerial Vehicles) [22]. Such aerial vehicles or multicopters can indeed exhibit negative speeds if flown backwards. If GPT-4 included the negative speeds intentionally, the LLM either learned from the code of this project on GitHub or from the corresponding image.



#### IV. CONCLUSION AND FUTURE WORK

The following section examines the steps that need to be taken to advance the concept presented and summarizes the findings of this study.

##### A. Next Steps

The work presented in this paper will be the starting point for building a modelling assistant for HMI generation from requirements based on open source LLMs like Llama 2 (i.e. CodeLlama), StarCoder (StarChat) or comparable LLMs, provided that these can be fine-tuned to achieve similar results as GPT-4. In [23] and [24], the authors give practical hints and instructions on how to fine-tune OpenLlama and StarCoder, respectively. It is an open question, whether or not these open LLMs can be fine-tuned in that way or training from scratch is needed. The fact that StarChat (and CodeGen) could not be included in the comparative study in this paper in a proper way due to the long runtimes and the timeout on HuggingFace does *not* imply that these LLMs should be not suited at all. On specialized hardware with enough capacity for running the LLMs exclusively, it should be possible to actually produce an outcome.

The finetuning task should in the intermediate run also include the training on the proper usage of the Python API of the SCADE MBSE framework (Model Based Systems Engineering) or similar development platforms (IBM Rational Rhapsody or Mathworks MATLAB/Simulink, depending on API suitability). This way the LLMs could produce the code for the HMI model directly. In SCADE Display this model would then be revised by the human engineer as depicted in Figure 1, before the code generator KCG would generate the actual C or Ada code. In the case of SCADE Display KCG, the tool uses the OpenGL SC subset of the graphic library explicitly targeted at safety critical applications (“SC” standing for “safety critical”).

In the long run the whole concept should be rolled out on model building in general, i.e., SysML based models of functional components, not only HMI/GUI/UI use cases. This will be the hardest part and requires a deep understanding of the system or subsystem to be developed. In in the scope of this work it was not possible to show the feasibility of such a modeling task. Presumably, it is a very long way from graphical model models for displaying purposes to functional models in a broad sense. Future work should examine the chances that such AI based assistants could support tomorrow’s engineers in the development process for software systems in general. Safety critical applications will not impose barriers if the AI assistant comes into play in the right phase of the process, as suggested by this paper.

##### B. Summary

In this paper it was shown that an AI/LLM assisted software development process without the need for manual coding is possible, if - instead of generating the final C/Ada source code - the LLM is instructed to create a *model* of the system. The term “system” in this respect refers to a specific

system for displaying information and human interaction as in HMI applications or for GUI/UI use cases. No modeling capabilities for functional components were included in this study, primarily due to the lack of an appropriate output format for the model itself (XML/SysML). In this scenario, the LLM was given a collection of requirements for the *visual* component of a software system and then instructed to translate these requirements into a graphical model of the HMI to be displayed, including basic colored shapes (rectangles, circles, etc.), text insets, call-outs, etc. - all arranged and adjusted to the fulfill the requirements. The results five from different LLMs were studied. However, only three of the LLMs produced comparable results (due to limitations concerning the computing platform).

Comparing the results, it was shown that GPT-4 is superior in performance and accuracy. The outcome in general shows a great amount of variability including visual forms of confabulation if details are left out or specified in an unprecise manner. GPT-4 does not produce contradictory objects with respect to the requirements, but also needs more self-explanatory, technically explicit guidelines than the requirements typically used by today’s engineers provide. The LLMs have deficiencies in common-sense reasoning and fill gaps in background knowledge either by figures stemming from misleading training data or by other unknown influencing factors. For this reason, a set of requirements with a fine granularity is important, i.e., the level of detail is crucial. However, the number of requirements that can be processed by the LLM in one single run is very limited. Because of this, techniques like Chain-Of-Thought (COT) prompt engineering should be employed in the instructions in which the requirements are embedded. LLMs benefit from a large context window (tokens to be processed in a single run) and the capabilities to process long documents (the requirements) must be improved by current techniques and future enhancements of the language models in order to fulfill the expectations and the practical usability of this whole concept.

It has been argued that the usage of LLM assistants for a no-code software development process is not prohibitive even for safety critical fields of application like cockpit instruments in aircrafts. The phase in which the LLM is employed and what task it is instructed to perform (i.e., model building instead of source code generation) is crucial and the human engineers always needs to stay in the loop by checking and revising the models. The use case studied was a Primary Flight Display (PFD) as used by pilots and served as an indicative measure for the performance of selected LLMs with coding capabilities. The study was performed on an empirical basis on this single use case, so no universal validity for other scenarios is claimed.

#### REFERENCES

- [1] OpenAI, “Gpt-4 technical report,” *ArXiv*, vol. abs/2303.08774, 2023. [Online]. Available from: <https://arxiv.org/abs/2303.08774> [retrieved: Oct., 2023]
- [2] H. Touvron *et al.*, “Llama 2: Open foundation and fine-tuned chat models,” 7 2023. [Online]. Available from: <https://www.semanticscholar.org/paper/104b0bb1da562d53cbda87aec79ef6a2827d191a> [retrieved: Oct., 2023]

## APPENDIX

## C. Source code examples

- [3] R. Li *et al.*, “Starcoder: may the source be with you!” 2023. [Online]. Available from: <http://arxiv.org/abs/2305.06161> [retrieved: Oct., 2023]
- [4] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, “Codegen2: Lessons for training llms on programming and natural languages,” *arXiv preprint*, 2023.
- [5] R. Bagnara, A. Bagnara, and P. M. Hill, “The misra c coding standard and its role in the development and analysis of safety- and security-critical embedded software.” *CoRR*, vol. abs/1809.00821, 2018. [Online]. Available from: <http://dblp.uni-trier.de/db/journals/corr/corr1809.html#abs-1809-00821> [retrieved: Oct., 2023]
- [6] R. Bagnara, M. Barr, and P. M. Hill, “Barr-c: 2018 and misra c: 2012: Synergy between the two most widely used c coding standards.” *CoRR*, vol. abs/2003.06893, 2020. [Online]. Available from: <http://dblp.uni-trier.de/db/journals/corr/corr2003.html#abs-2003-06893> [retrieved: Oct., 2023]
- [7] J. Holt, *SysML for Systems Engineering: A Model-Based Approach*, ser. Computing. Institution of Engineering and Technology, 2018. [Online]. Available from: <https://digital-library.theiet.org/content/books/pc/pbpc020e> [retrieved: Oct., 2023]
- [8] D. Iqbal, A. Abbas, M. Ali, M. U. S. Khan, and R. Nawaz, “Requirement validation for embedded systems in automotive industry through modeling,” *IEEE Access*, vol. PP, pp. 1–1, 01 2020.
- [9] H. Touvron *et al.*, “Llama: Open and efficient foundation language models,” 2023. [Online]. Available from: <http://arxiv.org/abs/2302.13971> [retrieved: Oct., 2023]
- [10] P. Liang *et al.*, “Holistic evaluation of language models,” 2022. [Online]. Available from: <https://arxiv.org/abs/2211.09110> [retrieved: Oct., 2023]
- [11] F. Chollet, “On the measure of intelligence,” 2019. [Online]. Available from: <http://arxiv.org/abs/1911.01547> [retrieved: Oct., 2023]
- [12] A. Srivastava *et al.*, “Beyond the imitation game: Quantifying and extrapolating the capabilities of language models,” 2023. [Online]. Available from: <http://arxiv.org/abs/2206.04615> [retrieved: Oct., 2023]
- [13] E. Davis, “Benchmarks for automated commonsense reasoning: A survey,” 2023. [Online]. Available from: <https://arxiv.org/abs/2302.04752> [retrieved: Oct., 2023]
- [14] T. B. Brown *et al.*, “Language models are few-shot learners,” 2020. [Online]. Available from: <https://arxiv.org/abs/2005.14165> [retrieved: Oct., 2023]
- [15] H. W. Chung *et al.*, “Scaling instruction-finetuned language models,” 2022. [Online]. Available from: <https://arxiv.org/abs/2210.11416> [retrieved: Oct., 2023]
- [16] T. Z. Zhao, E. Wallace, S. Feng, D. Klein, and S. Singh, “Calibrate before use: Improving few-shot performance of language models,” 2021. [Online]. Available from: <https://arxiv.org/abs/2102.09690> [retrieved: Oct., 2023]
- [17] H. Su *et al.*, “Selective annotation makes language models better few-shot learners,” 2022. [Online]. Available from: <https://arxiv.org/abs/2209.01975> [retrieved: Oct., 2023]
- [18] S. Diao, P. Wang, Y. Lin, and T. Zhang, “Active prompting with chain-of-thought for large language models,” 2023. [Online]. Available from: <https://arxiv.org/abs/2302.12246> [retrieved: Oct., 2023]
- [19] M. Chen *et al.*, “Evaluating large language models trained on code,” 2021. [Online]. Available from: <https://arxiv.org/abs/2107.03374> [retrieved: Oct., 2023]
- [20] L. Ouyang *et al.*, “Training language models to follow instructions with human feedback,” 2022. [Online]. Available from: <https://arxiv.org/abs/2203.02155> [retrieved: Oct., 2023]
- [21] S. Bubeck *et al.*, “Sparks of artificial general intelligence: Early experiments with gpt-4,” 2023. [Online]. Available from: <https://arxiv.org/abs/2303.12712> [retrieved: Oct., 2023]
- [22] kouky (nickname), “Primary flight display graphics framework for mac + ios,” 2022. [Online]. Available from: <https://github.com/kouky/PrimaryFlightDisplay> [retrieved: Oct., 2023]
- [23] M. Thissen, “Feintuning großer sprachmodelle – so gehts,” *iX*, vol. 10, pp. 50–59, 2023.
- [24] R. Wartala, “Eigene coding-assistenten trainieren und nutzen,” *iX*, vol. 10, pp. 44–49, 2023.

## GPT-4

```

import pygame
import sys

# Initialize pygame
pygame.init()

# Set dimensions
WIDTH, HEIGHT = 600, 800

# Colors
BLUE = (0, 102, 204)
BROWN = (153, 76, 0)
WHITE = (255, 255, 255)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
AMBER = (255, 191, 0)

# Create screen and clock
screen = pygame.display.set_mode((WIDTH, HEIGHT))
pygame.display.set_caption("Primary Flight Display")
clock = pygame.time.Clock()

def draw_horizon(pitch_angle=0):
    pygame.draw.rect(screen, BLUE, (0, 0, WIDTH, HEIGHT // 2 -
    pitch_angle))
    pygame.draw.rect(screen, BROWN, (0, HEIGHT // 2 - pitch_angle,
    WIDTH, HEIGHT))
    pygame.draw.line(screen, WHITE, (0, HEIGHT // 2 - pitch_angle,
    (WIDTH, HEIGHT // 2 - pitch_angle), 2)

def draw_aircraft_symbol():
    pygame.draw.line(screen, WHITE, (WIDTH//2 - 20, HEIGHT//2),
    (WIDTH//2 + 20, HEIGHT//2), 3)
    pygame.draw.line(screen, WHITE, (WIDTH//2, HEIGHT//2 - 20),
    (WIDTH//2, HEIGHT//2 + 20), 3)

def draw_altitude_tape(altitude=10000):
    start_y = 100
    end_y = 700
    pygame.draw.rect(screen, BLACK, (WIDTH - 60, start_y, 50, end_y
    - start_y))
    pygame.draw.polygon(screen, WHITE, [(WIDTH-10, HEIGHT//2-10),
    (WIDTH-10, HEIGHT//2+10), (WIDTH, HEIGHT//2)])
    for y in range(start_y, end_y + 1, 40):
        altitude_value = altitude - (HEIGHT//2 - y) * 10
        text_surface = pygame.font.SysFont('arial',
        20).render(str(altitude_value), True, WHITE, BLACK)
        screen.blit(text_surface, (WIDTH - 60, y))

def draw_airspeed_tape(speed=200):
    start_y = 100
    end_y = 700
    pygame.draw.rect(screen, BLACK, (10, start_y, 50, end_y -
    start_y))
    pygame.draw.polygon(screen, WHITE, [(10, HEIGHT//2-10), (10,
    HEIGHT//2+10), (0, HEIGHT//2)])
    for y in range(start_y, end_y + 1, 40):
        speed_value = speed - (HEIGHT//2 - y) * 2
        text_surface = pygame.font.SysFont('arial',
        20).render(str(speed_value), True, WHITE, BLACK)
        screen.blit(text_surface, (10, y))

def draw_heading_indicator(heading=90):
    pygame.draw.rect(screen, BLACK, (100, HEIGHT - 60, WIDTH - 200,
    50))
    for i in range(0, 360, 30):
        x_offset = (i - heading) * (WIDTH-200) / 360
        if 100 + x_offset > 90 and 100 + x_offset < WIDTH - 90:
            text_surface = pygame.font.SysFont('arial',
            20).render(str(i), True, WHITE, BLACK)
            screen.blit(text_surface, (100 + x_offset, HEIGHT -
            60))

def display_warning(message):
    text_surface = pygame.font.SysFont('arial', 25).render(message,
    True, WHITE, RED)
    screen.blit(text_surface, (WIDTH // 2 - 50, HEIGHT // 8))

def display_caution(message):
    text_surface = pygame.font.SysFont('arial', 25).render(message,
    True, BLACK, AMBER)
    screen.blit(text_surface, (WIDTH // 2 - 50, 2 * HEIGHT // 8))

# Main loop
while True:
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            sys.exit()

    screen.fill(BLACK)

    draw_horizon()
    draw_aircraft_symbol()
    draw_altitude_tape()
    draw_airspeed_tape()
    draw_heading_indicator()

    display_warning("STALL")
    display_caution("AUTOPILOT")

    pygame.display.flip()
    clock.tick(60)

```

Figure 8. Python code generated by GPT-4 for the full list of requirements for the HMI of a PFD. It represents the code for the right PFD in Figure 5.

## CodeLlama-34b-Instruct

```

import matplotlib.pyplot as plt
from PyQt5 import QtWidgets

class PrimaryFlightDisplay(QtWidgets.QMainWindow):
    def __init__(self):
        super().__init__()

        # Set up the main window
        self.setGeometry(0, 0, 800, 600)
        self.setWindowTitle("Primary Flight Display")

        # Create the layout
        self.layout = QtWidgets.QVBoxLayout()
        self.setCentralWidget(self.layout)

        # Add the sky and earth images
        self.sky_image = plt.imread('sky.png')
        self.earth_image = plt.imread('earth.png')
        self.sky_widget = QtWidgets.QLabel()
        self.earth_widget = QtWidgets.QLabel()
        self.sky_widget.setPixmap(self.sky_image)
        self.earth_widget.setPixmap(self.earth_image)
        self.layout.addWidget(self.sky_widget)
        self.layout.addWidget(self.earth_widget)

        # Add the horizon line
        self.horizon_line = QtWidgets.QFrame()
        self.horizon_line.setStyleSheet("background-color: white;
border: none; height: 2px;")
        self.layout.addWidget(self.horizon_line)

        # Add the aircraft attitude indicator
        self.aircraft_symbol = QtWidgets.QLabel()
        self.aircraft_symbol.setText("✈")
        self.aircraft_symbol.setAlignment(Qt.AlignCenter)
        self.aircraft_symbol.setFontSize(24)
        self.layout.addWidget(self.aircraft_symbol)

        # Add the altitude tape
        self.altitude_tape = QtWidgets.QScrollArea()
        self.altitude_tape.setVerticalScrollBarPolicy(
            QtCore.Qt.ScrollBarAsNeeded)
        self.altitude_tape.setHorizontalScrollBarPolicy(
            QtCore.Qt.ScrollBarAlwaysOff)
        self.altitude_tape.setWidgetResizable(True)
        self.altitude_tape.setFixedWidth(100)
        self.altitude_tape.setMinimumHeight(100)
        self.altitude_tape.setMaximumHeight(300)
        self.altitude_tape.setBackgroundRole(QtGui.QPalette.Base)
        self.altitude_tape.setAutoFillBackground(True)
        self.altitude_tape.show()
        self.layout.addWidget(self.altitude_tape)

        # Add the airspeed tape
        self.airspeed_tape = QtWidgets.QScrollArea()
        self.airspeed_tape.setVerticalScrollBarPolicy(
            QtCore.Qt.ScrollBarAsNeeded)
        self.airspeed_tape.setHorizontalScrollBarPolicy(
            QtCore.Qt.ScrollBarAlwaysOff)
        self.airspeed_tape.setWidgetResizable(True)
        self.airspeed_tape.setFixedWidth(100)
        self.airspeed_tape.setMinimumHeight(100)
        self.airspeed_tape.setMaximumHeight(300)
        self.airspeed_tape.setBackgroundRole(QtGui.QPalette.Base)
        self.airspeed_tape.setAutoFillBackground(True)
        self.airspeed_tape.show()
        self.layout.addWidget(self.airspeed_tape)

    def update_display(self, aircraft_pitch, aircraft_roll,
altitude, airspeed):
        # Update the aircraft attitude indicator
        self.aircraft_symbol.move(self.width() / 2 -
self.aircraft_symbol.width() / 2, self.height() / 2 -
self.aircraft_symbol.height() / 2 + aircraft_pitch * 10)
        self.aircraft_symbol.rotate(-aircraft_roll)

        # Update the altitude tape
        self.altitude_tape.verticalScrollBar().setValue(altitude)

        # Update the airspeed tape
        self.airspeed_tape.verticalScrollBar().setValue(airspeed)

if __name__ == "__main__":
    app = QtWidgets.QApplication([])
    pfd = PrimaryFlightDisplay()
    pfd.show()

```

Figure 9. Python code generated by CodeLlama for the short list of requirements for the HMI of a PFD. Note that it contained flaws (wrong usage of Qt and missing image files) that could be corrected manually. The (corrected) code refers to the left side in Figure 6.

## CodeLlama-34b-Instruct

```

import tkinter as tk
from math import sin, cos, radians

# Define constants for the dimensions and colors of the PFD
WIDTH = 800
HEIGHT = 600
SKY_COLOR = "#7ec0ee" # light blue
EARTH_COLOR = "#9c6f4a" # dark brown
HORIZON_LINE_COLOR = "white"
ALTITUDE_TAPE_COLOR = "white"
AIRSPEED_TAPE_COLOR = "white"
ARROW_COLOR = "black"

# Create a canvas object to draw on
canvas = tk.Canvas(width=WIDTH, height=HEIGHT, bg="gray")
canvas.pack()

# Draw the sky and earth
sky = canvas.create_rectangle(0, 0, WIDTH, HEIGHT//2,
fill=SKY_COLOR)
earth = canvas.create_rectangle(0, HEIGHT//2, WIDTH, HEIGHT,
fill=EARTH_COLOR)

# Draw the horizon line
horizon_line = canvas.create_line(0, HEIGHT//2, WIDTH, HEIGHT//2,
fill=HORIZON_LINE_COLOR)

# Draw the aircraft attitude indicator
aircraft_symbol = canvas.create_oval(WIDTH//2-50, HEIGHT//2-50,
WIDTH//2+50, HEIGHT//2+50, fill="#fff", outline="#000")

# Draw the altitude tape
altitude_tape = canvas.create_text(WIDTH*0.8, HEIGHT*0.5,
text="Altitude: ", font=("Helvetica", 16),
fill=ALTITUDE_TAPE_COLOR)
altitude_value = canvas.create_text(WIDTH*0.8, HEIGHT*0.5 + 20,
text="0 ft", font=("Helvetica", 16), fill=ALTITUDE_TAPE_COLOR)
arrow = canvas.create_polygon((WIDTH*0.8 - 10, HEIGHT*0.5 + 20),
(WIDTH*0.8 + 10, HEIGHT*0.5 + 20), (WIDTH*0.8, HEIGHT*0.5 + 30),
fill=ARROW_COLOR)

# Draw the airspeed tape
airspeed_tape = canvas.create_text(WIDTH*0.2, HEIGHT*0.5,
text="Airspeed: ", font=("Helvetica", 16),
fill=AIRSPEED_TAPE_COLOR)
airspeed_value = canvas.create_text(WIDTH*0.2, HEIGHT*0.5 + 20,
text="0 knots", font=("Helvetica", 16), fill=AIRSPEED_TAPE_COLOR)
arrow = canvas.create_polygon((WIDTH*0.2 - 10, HEIGHT*0.5 + 20),
(WIDTH*0.2 + 10, HEIGHT*0.5 + 20), (WIDTH*0.2, HEIGHT*0.5 + 30),
fill=ARROW_COLOR)

def update():
    # Update the position of the aircraft symbol based on the
    aircraft's pitch and roll angles
    pitch = getPitchAngle() * radians(180)/pi
    roll = getRollAngle() * radians(180)/pi
    x = WIDTH//2 + 50 * sin(pitch)
    y = HEIGHT//2 + 50 * cos(roll)
    aircraft_symbol.coords(x, y)

    # Update the altitude value on the altitude tape
    altitude = getAltitude()
    altitude_value.set_text("%d feet" % altitude)
    arrow.moveTo(WIDTH*0.8, HEIGHT*0.5 + 20)

    # Update the airspeed value on the airspeed tape
    airspeed = getAirspeed()
    airspeed_value.set_text("%d knots" % airspeed)
    arrow.moveTo(WIDTH*0.2, HEIGHT*0.5 + 20)

root.after(1000, update)

```

Figure 10. Variation of the Python code generated by CodeLlama for the short list of requirements. The code refers to the right side in Figure 7 and gets aborted by the Python interpreter after initial drawing of the basic layout due to missing sensor data.