

AUTOMATING SYSTEM MODELING USING LARGE LANGUAGE MODELS

Ivan de Souza Rehder[1], Petter Krus[2] & Emilia Villani[3]

[1]Instituto Tecnológico de Aeronáutica, Brazil
[2]Linköping Universitet, Sweden
[3]Instituto Tecnológico de Aeronáutica, Brazil

Abstract

To model a complex system, it need both energy and time, specially in the beginning, when there are still some concept decisions to be made that can impact the final design. This work investigates the use of Large Language Models (LLMs) as a tool to partially lighten this burden. The main contribution is the interaction between the user and the LLM: the user describes the system in natural language, and the LLM organizes this description into a structured JSON file following an Arcadia-based schema. This JSON is then processed by a Python script that generates system diagrams, and a Streamlit interface renders them interactively. A case study using a multimodal human–machine interface and an in-flight entertainment system shows that the LLM is able to translate informal descriptions into a valid structured representation. The diagram generation script remains under development and currently covers only the initial Arcadia modeling stages.

Keywords: Systems Design, Artificial Intelligence, Modeling and Simulation, MBSE

1. Introduction

Modeling a complex system requires time and mental effort, and this is specially true in the beggining of a project, when there are still many decisions to be made and information is incomplete. Model-Based Systems Engineering (MBSE) helps to organize this process by having a system slowly growing while answers and definitons are made at different levels of abstraction [1], but even with these methods, the engineer still need to do most of the work manually.

Large Language Models (LLMs) have shown to be capable of understanding and generating structured text in different applications [2, 3]. This means that it can be possible to use them to translate informal descriptions into structured representations.

This paper proposes a way where the user describes the system to an LLM-based chatbot, and the LLM outputs a JSON file with the system ontology. The main contribution of this work is the definition of a structured JSON schema that acts as an intermediate artifact between a natural language system description and Arcadia-compliant diagrams.

The modeling activity itself is still largely manual and time-consuming [1, 4]. Recent works have explored the use of LLMs to support structured engineering tasks, such as hazard analysis [5] and hardware design from natural language descriptions [6], showing that LLMs can process some degree of engineering formalism. But, none of these works try to use LLM for automatic generation of MBSE diagrams. This is the gap that the present work aims to fill.

This paper is organized as follows. Section 2. presents background on MBSE and the Arcadia Method. Section 3. describes the proposed approach. Section 4. presents the case studies. Section 5. discusses the results and limitations. Section 6. concludes the paper.

2. Background

2.1 Model-Based Systems Engineering

Model-Based Systems Engineering (MBSE) is the use of models to support the specification, design, analysis, and validation of systems throughout their lifecycle [1]. It is a shift from document-centered to model-centered development, where the system definition is managed through interconnected and traceable models.

The main advantage of MBSE is that it makes communication across disciplines better and reduces ambiguity. By connecting requirements, architecture, and analysis it helps align stakeholder needs with design decisions and supports early validation before any physical implementation.

2.2 The Arcadia Method

Among the available approaches that uses MBSE, Arcadia is a structured method developed by Thales that is based on operational need analysis and progressive architectural refinement [4]. It has four modeling phases:

1. **Operational Analysis (OA):** describes the operational context where the system will be applied, without the presence of the proposed system itself.It focuses on the actors, activities and interactions that already exist in the operational environment;
2. **System Analysis (SA):** treats the system as a black box and defines what it does, not how. It maps the interactions between the system and the external actors without detailing the internal structure;
3. **Logical Architecture (LA):** breaks the system into its main logical components and defines the behavior and functions that each one performs to fulfill the needs identified in the SA;
4. **Physical Architecture (PA):** defines the structural components of the system and maps which physical element is responsible for each function.

These phases connect stakeholder needs to implementation, which makes design decisions verifiable throughout the life cycle. This work focuses on the first and second layers, since they are the most directly connected to the initial system description that the user provides to the LLM.

2.3 Large Language Models in Engineering

Large Language Models (LLMs) are deep neural networks trained on large datasets of human written text. They can generate, summarize, and reason over texts, and they learn from patterns in data rather than from manually programmed rules [2, 7].

Recent models like GPT-4 have shown good performance in a range of tasks, including code generation, document writing, and even some forms of structured reasoning [3, 8].Despite this, LLMs still have some limitations as they can produce incorrect information and their output quality depends on how the prompt is written [2, 8].

In design and systems engineering, LLMs have been explored as collaborative tools that support ideation, documentation, and structuring [9, 10]. However, their integration into formal MBSE workflows, and in particular into methods like Arcadia, remains largely unexplored. This is the gap that this work addresses.

Similar approaches have been explored for fluid power and aircraft system design, where micro-templates embedded in prompts were shown to improve the repeatability and consistency of LLM outputs [11, 12]

3. Proposed Approach

The idea of this work is simple: instead of the engineer filling a modeling tool directly, they first describe the system to an LLM-based chatbot. The core contribution of this approach is the definition of a structured JSON schema that serves as an intermediate artifact between the natural language description and the Arcadia diagrams. This schema decouples the LLM interaction from the diagram generation step, making the pipeline modular and extensible to additional Arcadia phases. The LLM

organizes the system description into a JSON file following this schema, and a Python [13] script reads this file and generates the diagrams.

3.1 Workflow Overview

The proposed workflow is divided in three interactions, as shown in Figure 1.
In the first one, the user describes the system to an LLM-based chatbot. The chatbot is pre-configured with a previous prompt that contains instructions on how to interpret the description and the JSON template, and uses them to structure the description into a JSON file. This file is then sent back to the user.
In the second interaction, the user runs the Python script passing the JSON file as input. The script reads the system elements and interactions from the file, classifies them as nodes or edges, and uses the Cytoscape library to generate a set of HTML files. Each file is one of these Arcadia diagrams: a Block Definition Diagram (BDD) showing the system actors, a Use Case diagram (UC) showing the system capabilities, an Activity diagram (ACT) showing the system functions, and a Sequence diagram (SD) showing the usage scenarios.
In the third interaction, the user runs Streamlit [14], which reads the generated HTML files and renders them as an interactive web interface, allowing the user to navigate through the diagrams and inspect the system model.

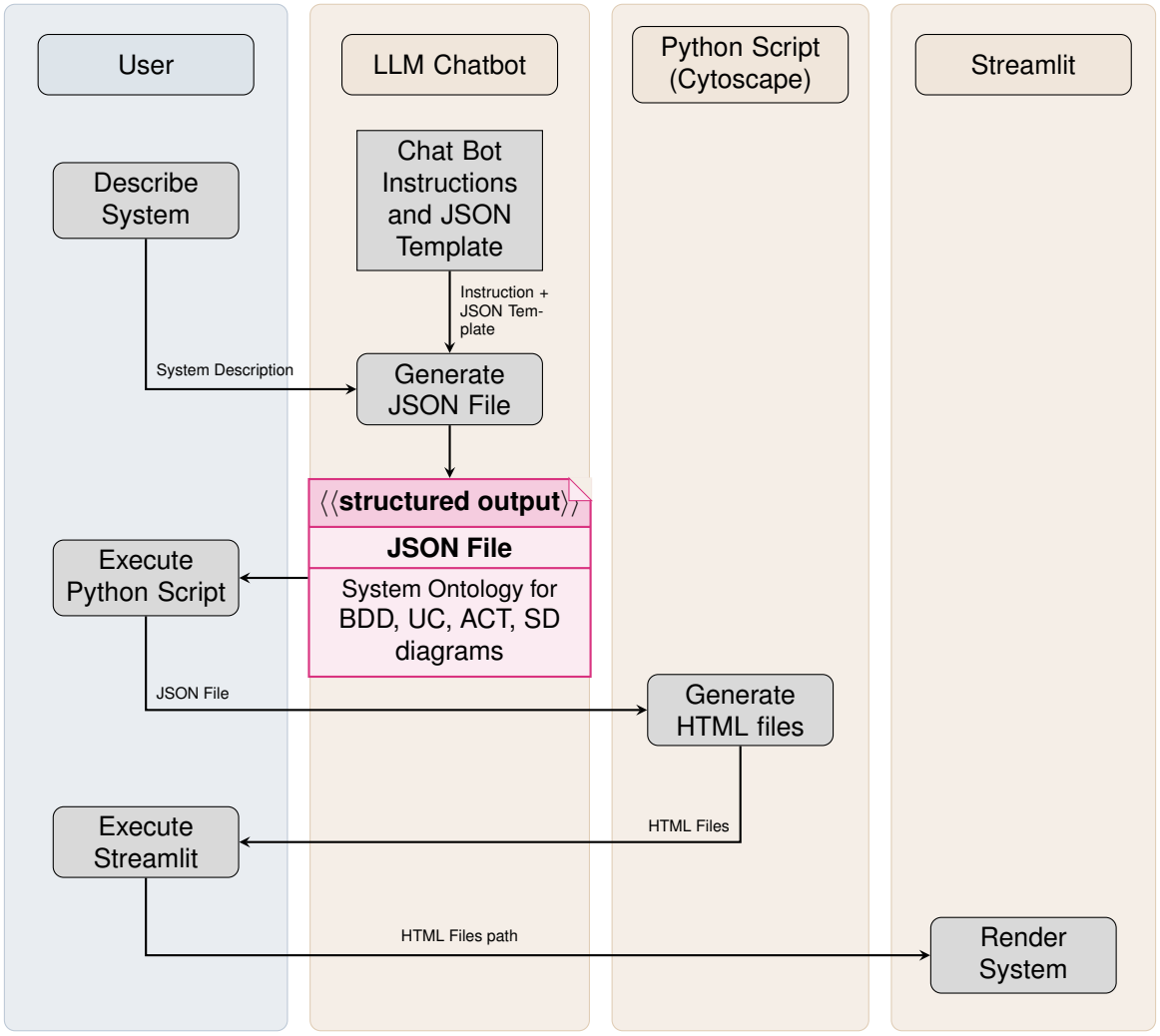


Figure 1 – Overview of the proposed LLM-assisted MBSE workflow.

3.2 LLM Interaction and Prompt Design

Before the user provides any input, the chatbot is configured with instructions that defines its behavior and to organize the system description into a structured JSON file based on a template file.Based

on the JSON template file, the LLM exactly which schema, fields, and naming conventions to follow when generating the output. This is analogous to the micro-template approach proposed in [11, 12], where predefined code snippets are embedded in the prompt to constrain the LLM output format.
In the interaction with the LLM, the user writes a free-form description of the system, but this description has to contain some key information to allow the LLM to generate a valid JSON file, besides an introduction telling the chatbot what the description is about with the system name and a general overview of what it does and what is its purpose.
The description covers the Operational Analysis, listing the system capabilities and, for each one, the associated use cases. Each use case is described step by step, citing the actor that performs each action, the function that is performed, and what that function produces or triggers. It is important to mention in the description for the chatbot that in this part that the system does not appear here. After that, the description covers the System Analysis, following the same structure but now the system appears as a black box that interacts with the external actors.
The LLM then uses this description, together with a JSON template that is already attached inside its system, to generate a structured JSON file that serves as input to the Python script.

3.3 JSON Schema and Intermediate Artifact

The JSON file generated by the LLM follows a template that is attached to the chatbot. The file is organized in two higher levels: a `meta` section with the system name and description, and a `phases` section that contains the modeling elements for each Arcadia phase.
The structure that repeats across phases is shown below. Each phase contains a set of lists where each element has an `id`, a `label`, and a `description`, and some phase-specific fields.

```
{
  "meta": {
    "name": "System Name",
    "description": "Short system description"
  },
  "phases": {
    "OA": {
      "actors": [...],
      "entities": [...],
      "capabilities": [...],
      "activities": [...],
      "interactions": [...],
      "scenarios": {
        "lane_order": [...],
        "scenarios": [...]
      }
    },
    "SA": {
      "system": [...],
      "actors": [...],
      "missions": [...],
      "capabilities": [...],
      "functions": [...],
      "interactions": [...],
      "scenarios": {
        "lane_order": [...],
        "scenarios": [...]
      }
    }
  }
}
```

The same file can be extended in the future to support additional Arcadia phases like Logical Architecture (LA) and Physical Architecture (PA), without changing the overall pipeline.

3.4 Diagram Generation Script

The Python script reads the JSON file and processes it. For each element in the file, the script creates a node or an edge based on its type, and uses this classification to build the diagrams.
From the actors and entities, the script builds a Block Definition Diagram (BDD) that shows the context of the system in the operational analysis and in the system analysis. From the capabilities, it builds a Use Case diagram (UC). With activities and functions, it builds an Activity diagram (ACT, but without showing interactions between the activities) that shows what activitiy is performed by each actor/entity. And finally, from the scenarios, it builds one Sequence diagram (SD) per use case scenario, showing the ordered sequence of actions for each scenario of use.
All diagrams are generated using the Cytoscape library and saved as HTML files. These files are then served by Streamlit, which renders them as an interactive web interface where the user can navigate through the diagrams and inspect the system model.

3.5 Implementation

The source code of the script and the JSON templates are publicly available at `https://github.com/IvanRehder/MBSEViewer`.

4. Case Study

4.1 System Description

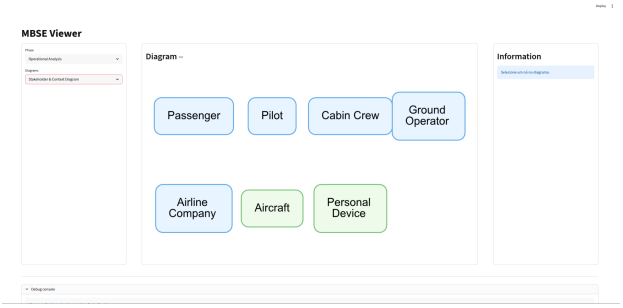
The first case study is a multimodal human–machine interface (HMI) for Intelligence, Surveillance, and Reconnaissance (ISR) missions in a Unmanned Teaming (MUM-T) context [15, 16]. In this scenario, a manned aircraft pilot flies one or more unmanned aerial vehicles (UAVs), using the HMI to monitor or control them and coordinate the mission without transferring full control of the UAVs [17, 18]. The main actors in this context are the aircraft pilot, the aircraft, the UAV, the UAV pilot, and the command and control (C2) system. The HMI is responsible for mediating all interactions between the pilot and the unmanned systems during the mission.
The second case study is an In-Flight Entertainment (IFE) system. This system is a built-in example model available in Capella, a tool for the Arcadia method. It was used because it already has a well-defined structure, which makes it easy to compare the output of the script against a known reference.

4.2 LLM Interaction

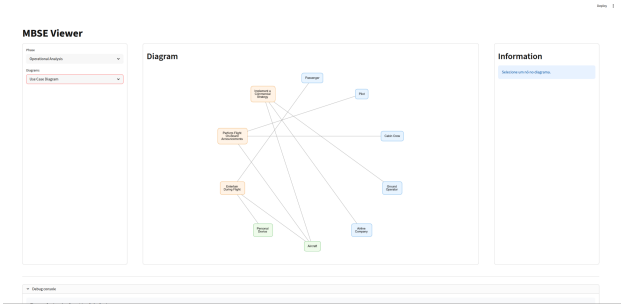
The HMI system was used as the main system during the development of the script, so the files and the code were being adjusted alongside the code. The IFE system was used afterwards to confirm that the pipeline works with a different system.
In both cases, the JSON generated by the LLM needed some manual adjustments before it could be used as input to the script. These adjustments were small, like a wrong classification of human, classifing the system itself as an actor or adjust the scenario lane order. A formal quantitative evaluation of the LLM output accuracy, such as element-level correctness rates and manual correction effort, was not conducted in this work and remains as future work.

4.3 Generated Artifacts

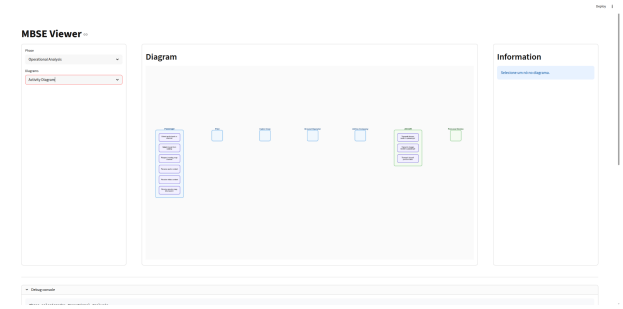
Figures 2 and 3 show the diagrams generated by the script for the IFE system.
For the IFE system, the script generated one BDD, one UC diagram, one ACT diagram per phase, and onthree SD. The OA phase produced three scenarios (one per use case in the Entertain During Flight capability), and the SA phase produced nine scenarios. Figure 2 shows one diagram of each type generated for the OA phase of the IFE system. For the HMI system, each phase produced 19 Sequence diagrams, one per scenario. Figure 3 shows the four diagram types generated for the SA phase of the HMI system.



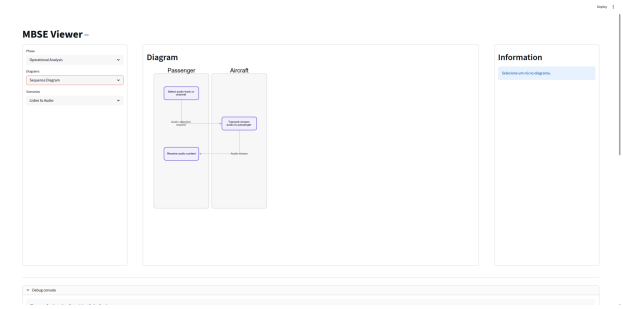
(a) Block Definition Diagram (OA) with the system actors and entities.



(b) Use Case diagram (OA) with the system capabilities and their supporters.



(c) Activity diagram (OA) with the system functions allocated to each actor.



(d) One of the sequence diagrams (OA) with the ordered steps of a usage scenario.

Figure 2 – Diagrams automatically generated by the script for the IFE system (OA phase).

5. Results and Discussion

The diagrams show different levels of quality depending on the diagram type. All nodes in all diagrams can be moved interactively in the Streamlit interface.

In both case studies, the BDD correctly represented all actors and entities present in the system description, with the only observed error being actor type misclassification.

The UC diagram works well for systems with a small number of actors and capabilities. When this number grows, the diagram can become hard to read, even with the navigation features provided by Streamlit. However, any diagram with a large number of elements would face the same issue.

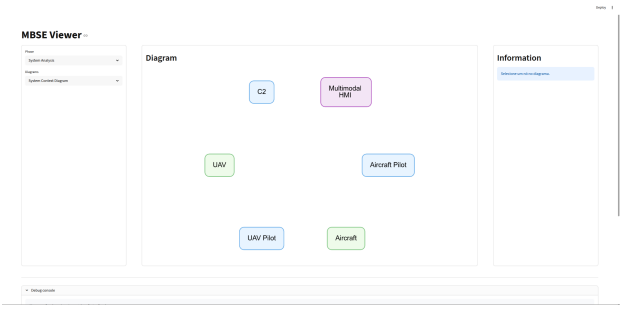
The AD ideally, it would show actors, activities, and interactions together, but including interactions without making the diagram too similar to the SD proved to be difficult with the Cytoscape library. So, the current version has only actors and activities. Besides that, actors that have no activities are still shown in the diagram, when they could be hidden. This is show in Figure **??**, where some actors appear without any activity. This also reflects a conceptual issue in the IFE example model, which defines actors that do not perform any activity in the OA scenarios.

In the SD, the main limitation is the order of the swimlanes, which has to be verified/corrected manually in the JSON file before running the script.

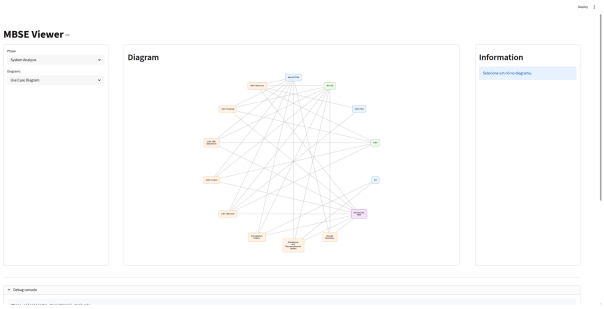
The most important result of this work is that the LLM interaction proved viable: the model was able to take a natural language description and produce a structured JSON file that correctly represented the system elements and their relationships according to the Arcadia schema. The Python script and the Streamlit interface are enablers that make this output visible, but the core of the approach lives in this translation step.

That said, the time to describe the system to the LLM is still significant. Even though the approach has worked, the description had to be very detailed to output a good JSON file. This is an important limitation, since one of the goals was to reduce the modeling effort. Improving the quality of this interaction is therefore one of the the directions for future work.

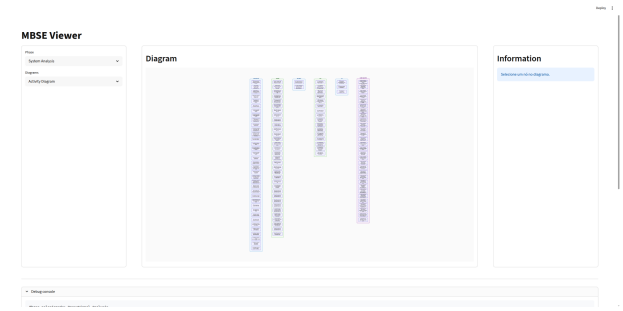
The current implementation covers only the Operational Analysis and System Analysis phases. The Logical Architecture was not implemented because its BDD needed to show the internal structure of the system, which was too difficult to implement within the time available for this publication. The



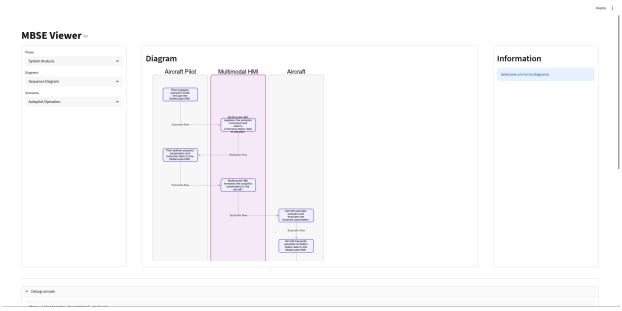
(a) Block Definition Diagram (SA) with the system actors and entities.



(b) Use Case diagram (SA) with the system capabilities and their supporters.



(c) Activity diagram (SA) with the system functions allocated to each actor.



(d) One of the sequence diagrams (SA) with the ordered steps of a usage scenario.

Figure 3 – Diagrams automatically generated by the script for the HMI system (SA phase).

Physical Architecture was not attempted either.
Despite these points, the approach shows potential. The pipeline is functional, the generated diagrams are useful for early-phase system modeling, and the tool can be used as a starting point for a new model-based systems engineering tool, where the user interacts with the system through natural language rather than through a dedicated modeling environment.

6. Conclusion

This paper presented an idea to use LLMs to help with the early phases of system modeling. The user describes the system to a chatbot, the chatbot generates a JSON file, and a Python script transforms this file into Arcadia diagrams.
The two case studies showed that this is viable. The central finding is that an LLM can translate a natural language system description into a structured MBSE representation. The script then consumed this representation to produce Operational Analysis and System Analysis diagrams. The BDD and SD showed the best results, while the AD still has some limitations. The main bottleneck identified was the effort required to write a sufficiently detailed system description for the LLM to generate a valid JSON file, which points to the LLM interaction itself as the main area for improvement in future work.
The script does not yet cover the Logical and Physical Architecture levels. The LA diagram in particular proved difficult to implement due to the complexity of showing the system components and their hierarchy in a clear way.
Future work will:

- extend the pipeline to the Logical and Physical Architecture levels;
- improve the user experience and the quality of the generated diagrams;
- investigate the automatic generation of technical documentation from the same JSON file.

The tool also has potential to evolve into a more complete systems engineering environment, where the user interacts through natural language instead of a dedicated modeling tool.

7. Acknowledgements

The authors acknowledge the financial support of the following Brazilian Funding Agencies: FAPESP with grant 2021/11258-5; FINEP with grants 01.24.0600.00 and 01.20.0195.00; CNPq and CISB.

8. Contact Author Email Address

mailto: ivan.rehder@ga.ita.br

9. Copyright Statement

The authors confirm that they, and/or their company or organization, hold copyright on all of the original material included in this paper. The authors also confirm that they have obtained permission, from the copyright holder of any third party material included in this paper, to publish it as part of their paper. The authors confirm that they give permission, or have obtained permission from the copyright holder of this paper, for the publication and distribution of this paper as part of the ICAS proceedings or as individual off-prints from the proceedings.

References

[1] INCOSE. *INCOSE systems engineering handbook*. John Wiley & Sons, 2023.

[2] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[3] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. 2023. arXiv preprint arXiv:2303.08774.

[4] Pascal Roques. *Systems architecture modeling with the Arcadia method: a practical guide to Capella*. Elsevier, 2017.

[5] Simon Diemert and Jens H Weber. Can large language models assist in hazard analysis? In *International Conference on Computer Safety, Reliability, and Security*, pages 410–422. Springer, 2023.

[6] Kaiyuan Yang, Haotian Liu, Yuqin Zhao, and Tiantai Deng. A new design approach of hardware implementation through natural language entry. *IET Collaborative Intelligent Manufacturing*, 5(4):e12087, 2023.

[7] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. 2021. arXiv preprint arXiv:2112.11446.

[8] Sébastien Bubeck, Varun Chadrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. 2023.

[9] Yang Shi, Tian Gao, Xiaohan Jiao, and Nan Cao. Understanding design collaboration between designers and artificial intelligence: A systematic literature review. *Proceedings of the ACM on Human-Computer Interaction*, 7(CSCW2):1–35, 2023.

[10] Liuqing Chen, Haoyu Zuo, Zebin Cai, Yuan Yin, Yuan Zhang, Lingyun Sun, Peter Childs, and Boheng Wang. Toward controllable generative design: A conceptual design generation approach leveraging the function–behavior–structure ontology and large language models. *Journal of Mechanical Design*, 146(12), 2024.

[11] Petter Krus. Large language model in aircraft system design. In *33rd Congress of the International Council of the Aeronautical Sciences (ICAS), Florence, Italy*, 2024.

[12] Petter Krus. Using large language models for fluid power system design. *JFPS International Journal of Fluid Power System*, 19(2):74–79, 2026.

[13] Python Software Foundation. *Python Language Reference, version 3.11*, 2026.

[14] Streamlit. Streamlit: The fastest way to build and share data apps, 2024.

[15] Grant Taylor and Terry Turpin. Army aviation manned-unmanned teaming (mum-t): past, present, and future. In *18th International Symposium on Aviation Psychology*, page 560, 2015.

[16] Grant S Taylor, Thomas J Alicia, Terry Turpin, and Amit Surana. Controlling multiple unmanned aircraft from a manned helicopter: The need for advanced autonomy and refined pilot-vehicle interface. In *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, volume 61, pages 78–82. SAGE Publications Sage CA: Los Angeles, CA, 2017.

[17] Aditya Das, Patrik Kol, Cody Lundberg, Kris Doelling, Hakki Erhan Sevil, and Frank Lewis. A rapid situational awareness development framework for heterogeneous manned-unmanned teams. In *NAECON 2018-IEEE National Aerospace and Electronics Conference*, pages 417–424. IEEE, 2018.

[18] Samuel J Levulis, Patricia R DeLucia, and So Young Kim. Effects of touch, voice, and multimodal input, and task load on multiple-uav monitoring performance during simulated manned-unmanned teaming in a military helicopter. *Human factors*, 60(8):1117–1129, 2018.