

Augmenting Aerospace System Design Using Large Language Models

Petter Krus

Division of Fluid and Mechatronic Systems, Linköping University, Sweden

E-mail:

petter.krus@liu.se

Abstract

The democratization of artificial intelligence, exemplified by the widespread availability of ChatGPT since late 2022, presents significant opportunities for innovation in engineering system design. This paper explores how large language models (LLMs) can support and partially automate engineering design processes by generating system configuration rules and conceptual system architectures.

The methodology is demonstrated using case studies on hybrid-electric propulsion systems and actuation system design. LLMs are used to generate system configurations represented as UML component diagrams. Recognizing the non-deterministic behaviour of LLMs, a structured prompting methodology is proposed. This includes reusable context prompts with embedded examples (micro templates), allowing increased reproducibility and specificity.

Furthermore, the study shows how LLMs can be used to generate Python scripts that act as configurators, producing system descriptions within specified design freedoms. These scripts enable iterative expansion, refining the architecture over time. The outputs are exportable to simulation environments, allowing further analysis and optimisation.

Integration with LLM APIs within engineering tools enables interactive expansion of system diagrams into subsystems and detailed components, streamlining complexity management.

These findings highlight the potential of LLMs to transform aerospace system design by improving efficiency, traceability, and early-phase exploration of system architectures.

Keywords: large language models, aerospace engineering, system configuration, hybrid-electric propulsion, LLM-API, system modelling

1 Introduction

The emergence of generative artificial intelligence (AI), particularly through large language models (LLM) such as OpenAI's ChatGPT, marks a paradigm shift in engineering system design. Since its public release in late 2022, ChatGPT has democratized access to advanced AI capabilities, enabling domain experts rather than AI specialists to apply natural language models directly within their own engineering workflows [1, 2].

LLMs have proven effective as few-shot or even single-shot learners [3], making them particularly suited for conceptual and architectural design tasks where the available information may be imprecise or incomplete. Their ability to generate human-readable output, such as UML component diagrams and Python scripts, bridges the gap between textual requirements and formal system models [1].

This paper explores how LLMs can support system configuration and modeling activities in aerospace applications. The focus is on hybrid electric propulsion and actuation systems as representative domains [1, 2]. By combining reusable context prompts including structures termed *micro-templates* with task specific prompts, it is possible to guide LLMs toward producing useful and repeatable outputs despite their inherent stochasticity [2].

Moreover, instead of asking LLMs to generate full systems directly, a more reliable approach is to use them to generate code—typically in Python—that acts as a configurator. This allows the resulting design space to be constrained and iteratively refined across several levels of abstraction [4]. These scripts can be further extended using LLM assistance and eventually exported to simulation environments such as Hopsan or Modelica for system simulation and analysis.

Finally, LLMs can be integrated into engineering tools using APIs, facilitating interactive expansion of system models and integration of domain knowledge, requirements, and design rules. In this way, LLMs enable a new workflow where early-phase design exploration is more rapid, creative, and informed by both domain logic and learned language patterns [1].

The application used throughout this paper is the conceptual design of Hybrid Electric aircraft propulsion system. Hybrid electric aircraft has been studied intensely during especially the last decade as a way forward towards low-emission aviation. A review of these systems can be found e.g. in [5].

2 Methodology

This section outlines the structured methodology for employing large language models (LLMs), specifically ChatGPT, in the conceptual and system-level design of aerospace systems. The approach rests on leveraging prompt engineering, reusable prompt templates ("micro-templates"), and iterative expansion via script generation. The core methodology can be divided into five key steps [2, 1]:

2.1 Prompt Engineering and Interaction Modes

The first step involves formulating prompts that communicate the design task effectively to the LLM. For using an LLM it is useful to think of its function as black-box systems, where an input string produces an output string in response. The effectiveness depends primarily on how well the task is described. A useful strategy is to divide the prompts into two parts:

- **Context prompt:** Contains general instructions, definitions, design rules, and small examples as micro-templates. This is a prompt for domain, e.g., aircraft propulsion systems, actuation systems, etc, needed for the design task.
- **Specific prompt:** Defines the specific design task (e.g., "Generate a concept for a hybrid electric aircraft propulsion system with fuel cells, for a commuter aircraft under FAR 25 rules").

This two-tier structure ensures that outputs are consistent and aligned with engineering expectations. The use of micro-templates within a generic-domain-prompt is critical to ensure the LLM's output format and quality [2]. The generic-domain-prompt, GDP, contains general instruction related to a domain of interest and can be re-used for other similar tasks related to this domain. With this approach, a specific-prompt, SP, can then be added that can be short and only contain specifics for the particular task, while a considerable effort can be put into refining the GDP. Here it is good practice to use a structured format using, e.g., such as Markdown, JSON, or a Prompt Declarative Language (PDL) based on the YAML format for readability and re-usability [6]. However, it seems that it is more important that the prompt is structured than exactly which format is chosen. In this study Markdown has been used, but no significant differences were detected when

using YAML. One drawback with YAML is that it is sensitive to indentation, which makes it complicated to use fragments of a YAML code in another code with a different structure.

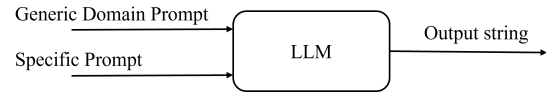


Figure 1: Illustration of prompting structure using a generic-domain-prompt and a task-specific prompt.

2.2 Micro-Templates

Micro-templates are embedded examples within the generic-domain-prompt that show representative correct outputs (e.g., component-based UML code). These guide the LLM in producing outputs with correct syntax, structure and also the desired level of abstraction. They can also be customized to define preferred subsystem configurations [1].

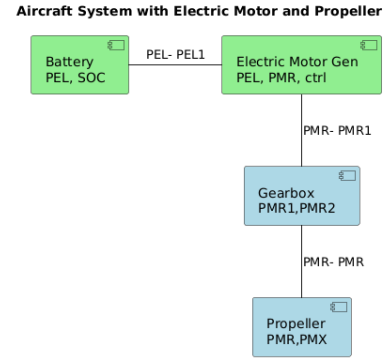


Figure 2: Example of a UML micro-template for a simple battery electric propulsion system.

2.3 From Natural Language to Structured Diagrams

Once a valid prompt is issued, the LLM can generate output such as:

- PlantUML code for UML activity diagrams representing system architecture.
- code for diagrams for function-means trees.
- Morphological matrices.

These outputs can be visually rendered or integrated into system modeling environments. This approach has been tested across different engineering domains, including fluid power and aerospace propulsion [2, 1].

2.4 Python Configurator Generation

To manage design complexity and improve repeatability, a preferred approach is to ask the LLM to generate a Python script that acts as a system configurator [2, 4]. This script:

- Encodes configuration rules.

2.5 API Integration and Interactive Expansion

- Generates specific system instances based on design parameters.
- Allows for iterative refinement and expansion.

The following process can be used to generate the Python code for the configurator:

- **User**
 - Define a prompt based on a context prompt with a specific prompt.
- **LLM**
 - Use this prompt to generate a UML diagram for validation and return it to the user.
- **User**
 - Reviews the generated UML diagram.
 - Requests Python code that implements the configuration rules and generates UML code.
- **LLM**
 - Produces the Python code according to the configuration rules.
- **Python Code**
 - Execute various input configurations and generate corresponding UML diagrams automatically.
- **User**
 - Reviews the generated UML diagrams and test configurations.
 - Provides feedback and final approval.
- **End Result:** Validated Python code with configuration rules → UML diagrams are iteratively refined with LLM and user feedback.

This approach helps contain the stochasticity of the LLM by delegating deterministic structure generation to code.

2.5 API Integration and Interactive Expansion

The final step involves integrating the LLM into engineering tool chains via APIs. This allows for:

- Context-aware prompting using in-tool state.
- Incremental design elaboration, e.g., decomposing a system into subcomponents interactively.
- Integration of requirements, standards, and functional constraints [1].

The methodology enables design engineers to work within a controlled yet creative process, where LLMs serve as assistants capable of proposing, generating, and transforming conceptual structures on demand.

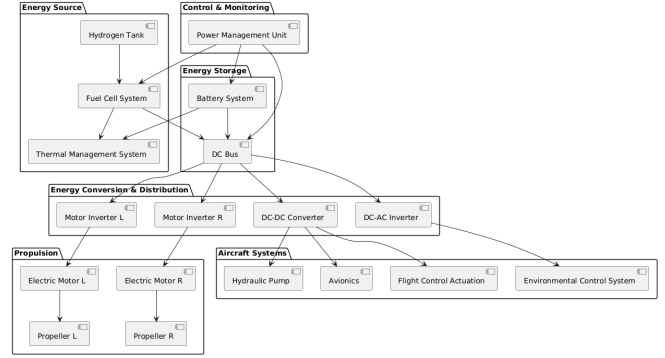


Figure 3: UML component diagram from single prompt

3 Case Study: Hybrid Electric Aircraft

To evaluate and demonstrate the proposed methodology, a representative case study from aerospace systems design of a hybrid-electric propulsion system concept is used. This example illustrates the full workflow, from prompt engineering to structured system representations and Python-based configurator, and also a behavioral diagram.

The design task was to generate an architecture for a hybrid-electric aircraft featuring two propellers and a fuel-cell-based power system.

3.0.1 Prompt-Based System Generation

The specific prompt was:

"Generate a concept for a hybrid electric aircraft for 49 passengers with fuel cells. Show the architecture as code for a component diagram in PlantUML format."

The LLM generated a PlantUML diagram describing the system structure, including fuel cells, electric motors, power converters, and a battery shown in Fig. 3.

This system lacks some details about redundancy, e.g., regarding battery and fuel cells, and have no well defined port definitions that would be necessary for translation into e.g., a simulation model. Furthermore, the output varies greatly between chats with identical prompts. However, the output provides also additional textual description about the system and can definitely serve as a starting point that could be used for further refinement. It also includes systems that was not mentioned in the instructions, such as thermal management system and aircraft systems (avionics, flight control).

This reflects the typical performance of LLMs in such tasks, as also observed in [1], although the quality of the output has improved greatly from ChatGPT3.5 to ChatGPT-4o, to GPT o3 and GPT o4-mini and to ChatGPT 5.

To improve the result and to make the response more consistent, the context Prompt in the Appendix is used. Using this, a more concise system is obtained with port definitions and naming. Using ChatGPT 5 this results in the following plantUML code:

3.1 Behaviour, UML Activity Diagram

```
@startuml
' Top-down layout
top to bottom direction

' === Controller at the Top ===
component "Controller\nctrl" as Controller #LightYellow

' === Left Energy Chain ===
component "Hydrogen Storage L\nQM" as H2StorageL #Turquoise
component "Fuel Cell L\nEL" as FuelCellL #LightGreen
component "DCDC Converter L\nEL" as DCDCL #LightGreen
component "Battery L\nEL" as BatteryL #LightGreen
component "EMG L\nEL, MR" as EMGL #LightGreen
component "Gearbox L\nMR" as GearboxL #LightBlue
component "Propeller L\nMR, PMX" as PropellerL #LightBlue

' === Right Energy Chain ===
component "Hydrogen Storage R\nQM" as H2StorageR #Turquoise
component "Fuel Cell R\nEL" as FuelCellR #LightGreen
component "DCDC Converter R\nEL" as DCDCL #LightGreen
component "Battery R\nEL" as BatteryR #LightGreen
component "EMG R\nEL, MR" as EMGR #LightGreen
component "Gearbox R\nMR" as GearboxR #LightBlue
component "Propeller R\nMR, PMX" as PropellerR #LightBlue

' === Aircraft Body at Bottom ===
component "Aircraft Body\nPMX, Aero, states" as AircraftBody #LightBlue

' === Positioning (parallel left-right) ===
H2StorageL -down-> FuelCellL
FuelCellL -down-> DCDCL
DCDCL -down-> BatteryL
BatteryL -down-> EMGL
EMGL -down-> GearboxL
GearboxL -down-> PropellerL
PropellerL -down-> AircraftBody

H2StorageR -down-> FuelCellR
FuelCellR -down-> DCDCL
DCDCL -down-> BatteryR
BatteryR -down-> EMGR
EMGR -down-> GearboxR
GearboxR -down-> PropellerR
PropellerR -down-> AircraftBody

' === Controller Connections ===
Controller -[dashed]-> FuelCellL : ctrl
Controller -[dashed]-> FuelCellR : ctrl
Controller -[dashed]-> DCDCL : ctrl
Controller -[dashed]-> DCDCL : ctrl
Controller -[dashed]-> EMGL : ctrl
Controller -[dashed]-> EMGR : ctrl
BatteryL -[dashed]-> Controller : SOC
BatteryR -[dashed]-> Controller : SOC

' === Title ===
title Vertical Layout - Redundant Hybrid-Electric
Propulsion (Fuel Cell)
@enduml
```

This code results in the diagram in fig. 4.

The result is now more consistent when repeating the prompt in a new chat. Also the level of abstraction and the types of components and types of connections are now more consistent and reliable. As another example, a parallel hybrid propulsion system is generated using the generic domain prompt together with the following prompt:

"Generate a concept for a parallel hybrid electric

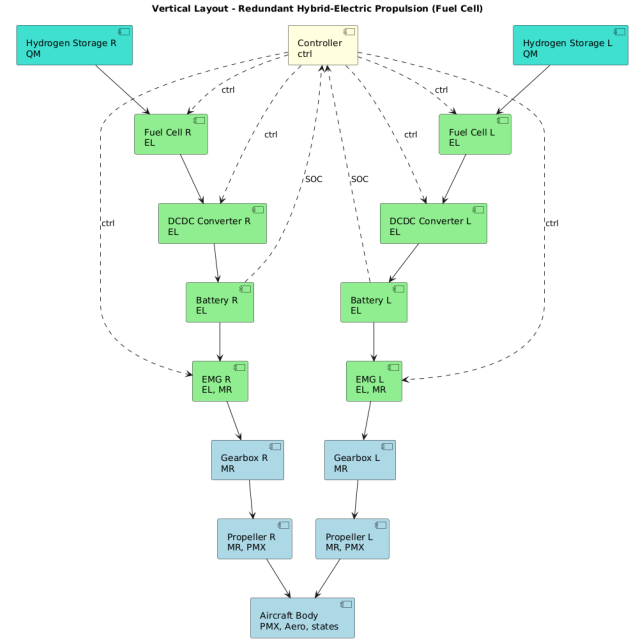


Figure 4: PlantUML diagram generated directly from a LLM prompt describing a hybrid-electric aircraft with fuel cells.

aircraft for 49 passengers with gas turbines, batteries and electric motors. Show the architecture as code for a component diagram in PlantUML format."

This results in the diagram shown in fig. 5.

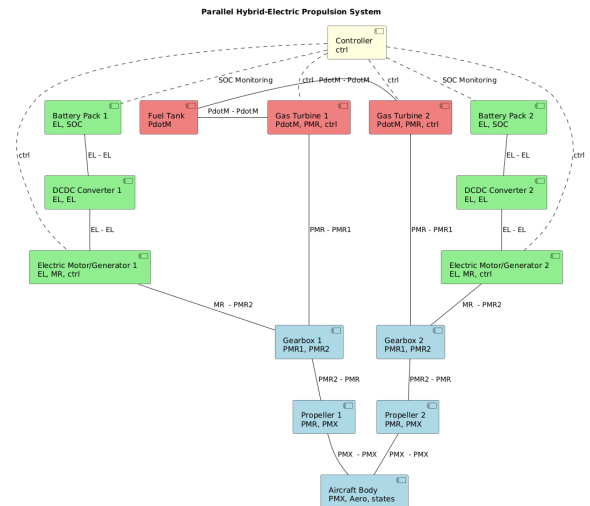


Figure 5: Parallel hybrid propulsion configuration.

3.1 Behaviour, UML Activity Diagram

Another aspect of the system design is to describe the system behaviour and function. This can be done using a UML Activity Diagram. Here, the LLM can be used to generate a UML activity diagram of a flight referring to the previous dialog.

Prompt: "Make an UML activity diagram for PlantUML of a flight of the regional fuel cell electric aircraft in the example"

The result is shown in fig. 6. Note that for this result no generic domain prompt was used, but that this could be used to have more control of the type of information, the level of detail, and appearance of the diagram.

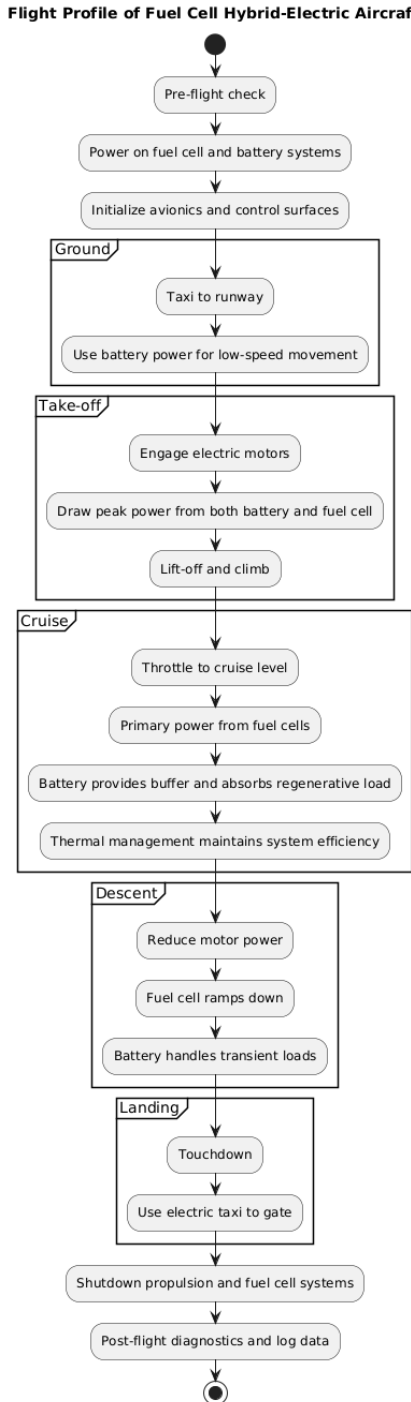


Figure 6: UML activity diagram generated directly by code from an LLM prompt.

3.1.1 Python-Based Configurator Generation

In order to have a more reliable result, especially for more complex systems. The LLM can be asked to generate a Python program that in turn can generate the system giving a set of degrees of freedom, as indicated in Fig. ??.

The process for this example was started off using the prompt below that followed the chat where the parallel hybrid was generated.

Prompt: "Generate Python code that programmatically produces PlantUML component-diagram text for a variety of aircraft propulsion architectures. For example:

- Fuel-cell hybrid-electric systems - Mixed (parallel) hybrids, where both the primary source and batteries can drive propellers (distributed propulsion)

The code should allow me to specify:

- **num_propellers**: the number of propellers (e.g. 2-8)
- **hybrid_mode**: either "series" or "parallel"
- **primary_source**: either "hydrogen" (fuel cell) or "kerosene" (gas turbine)

After tests and a few corrections, the configurator could generate the diagrams in Figs. 7 and 8.

This Python configurator encodes configuration rules for a class of hybrid-electric aircraft architectures. This script includes parametric definitions for the number and type of energy sources (e.g., batteries, fuel cells), power distribution logic (e.g., series or parallel hybrid), and propulsion arrangements. When executed, the script can automatically produce consistent PlantUML system descriptions for different mission profiles or technology configurations [1, 2]. The calls to the configurator to generate the two diagrams are simply:

```

uml_series = generate_plantuml(
    num_props=6,
    hybrid_type='series',
    primary_energy='hydrogen')

uml_parallel = generate_plantuml(
    num_props=4,
    hybrid_type='parallel',
    primary_energy='aviation_fuel')

```

3.2 API Integration and Interactive Expansion

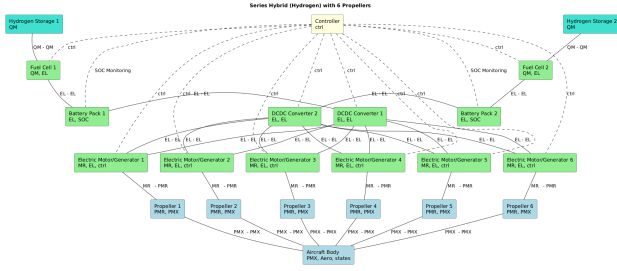


Figure 7: PlantUML output of a hybrid-electric propulsion configuration generated using a Python-based LLM-generated configurator. Here it is a distributed propulsion system with eight electric motors have been specified.

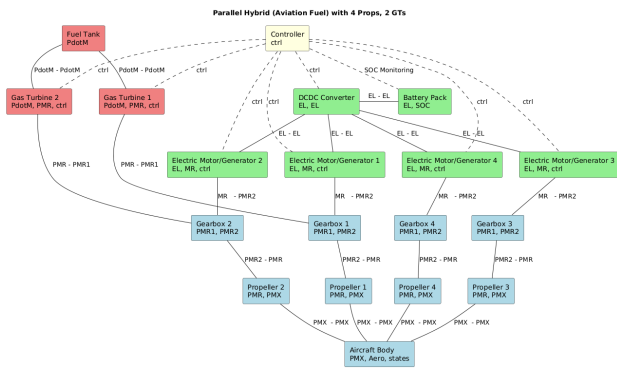


Figure 8: PlantUML output of a hybrid-electric propulsion configuration generated using the same Python-based LLM-generated configurator as in 7. Here it is a system with four electric machines and two gas turbines in a parallel-hybrid configuration.

This Python-based configurator approach has several advantages:

- It constrains the LLM output to a validated structural template, improving reproducibility.
- It allows scalability and reuse by encoding architectural degrees of freedom in the script.
- It decouples the probabilistic behavior of LLMs from the deterministic process of system instantiation.

The configurator can also be extended through dialogue with the LLM, which is instructed to add new capabilities, e.g., support for solar panels, and auxiliary systems. The output UML diagrams can be used for architecture visualization, system documentation, or simulation initialization. Such configurators could e.g., be used to configure simulation models for different classes of systems.

3.1.2 Comparison: Direct LLM output vs. Configurator-Driven Generation

The difference between directly generating system architectures using LLM prompts and using an LLM-generated Python configurator lies in their trade-off between creativity and

control. Prompt-based generation allows for broad exploration of possible system configurations but suffers from non-determinism and occasional structural errors. In contrast, the configurator approach offers a deterministic and repeatable output bounded by a predefined design space.

- **Prompt-Based Output:** Flexible, exploratory, but variable in quality.
- **Configurator Output:** Consistent and scalable, but limited to encoded rule sets.

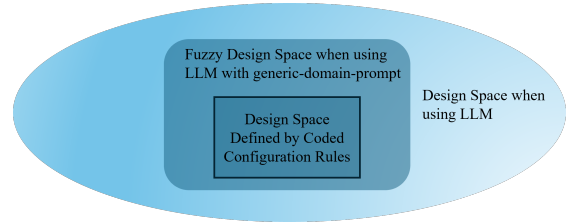


Figure 9: Comparison between system generation using direct LLM prompting and system generation via Python configurator. LLM prompting can go outside what is defined in the generic-domain-prompt.

The demonstrated results demonstrates the capacity of LLMs to assist with early-stage engineering design tasks that require conceptual abstraction, functional decomposition, and architectural reasoning.

3.2 API Integration and Interactive Expansion

A useful application that has been enabled by LLM API integration, is the automatic generation and refinement of *Function-Means Trees*. Here, a top function can be defined, e.g., "Provide air transport for 70 passengers". This then goes into the LLM through the API as a prompt. This, generates a set of alternative means as shown in fig. 10. In addition to the name, also a comment is generated that together with the name can go into the next level to generate the next level of functions. Here some of the alternative means are not viable, but only one means should be selected from all the alternatives, and chances are that there is at least one viable solution, in the unlikely event there is none, the system can be set to generate more alternatives.

These function-means structures can be used for ideation, comparison of architectural variants, and communication between design stakeholders. Through API integration, the LLM can dynamically update the function-means tree in response to system changes or additional requirements.

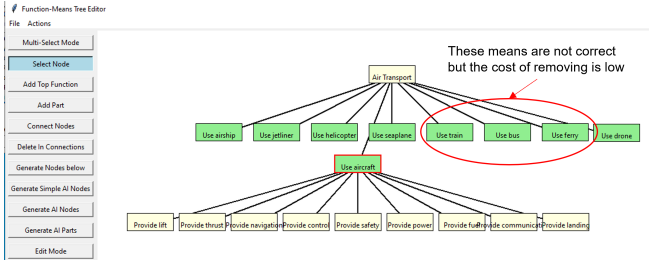


Figure 10: Function-means tree generated using LLM-API for a transport aircraft.

In addition to functional decomposition, API-assisted tools enable interactive system decomposition. Starting from a top-level architecture such as the hybrid electric aircraft with fuel cells and two propellers, the user can select a component (e.g., the fuel cell) and invoke an LLM-assisted prompt to elaborate supporting subsystems such as "air supply", "humidification", or "thermal management" [1]. This facilitates interactive system development where complexity is added incrementally through modular expansion.

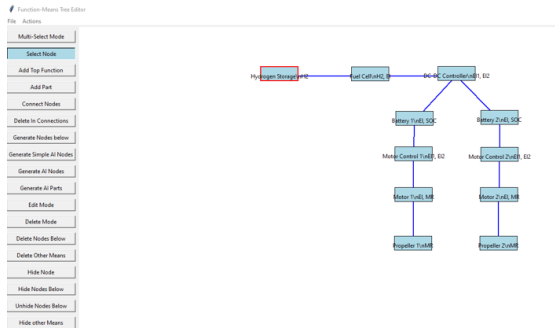


Figure 11: System imported into a system editor enabled with LLM-API for a fuel-cell-based hybrid-electric aircraft.

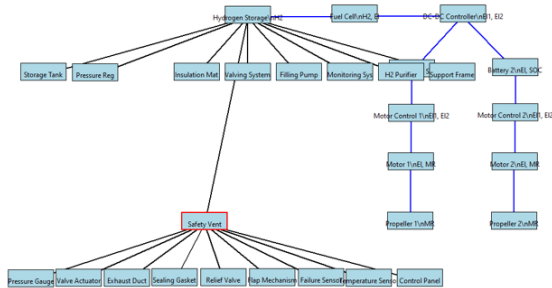


Figure 12: Decomposition of the fuel cell subsystem using interactive LLM-API to generate, auxiliary components such as air supply and thermal management.

Incorporation of LLMs via APIs transforms static system models into dynamic and evolvable design spaces. This is particularly valuable in early-stage system design, where design intent is still emerging and flexibility is essential.

4 Discussion

The case studies and integration examples discussed in this paper reveal both the promise and current limitations of using large language models (LLMs) in engineering system design.

The primary benefit of incorporating LLMs into the design process is the acceleration and support of conceptual system development. Unlike conventional CAD or simulation tools, LLMs offer:

- **Rapid synthesis of system architectures** from high-level natural language input.
- **Assistance with documentation**, such as generating structured diagrams, requirement breakdowns and activity diagrams.
- **Support for creativity and exploration**, especially in the early stages when design alternatives are numerous and loosely defined.
- **Scalable automation through code generation**, particularly via Python scripts that define configurable design spaces.

In addition, LLMs provide a flexible framework for integrating auxiliary features such as regulatory compliance checks.

4.1 Structured Process Summary

The methodology proposed in this paper follows a six-step structured process for LLM-augmented aerospace design:

1. **Prompt Engineering:** Design task-specific and reusable context prompts to structure the model interaction.
2. **Use of Micro-Templates:** Embed representative examples to guide output structure and semantics.
3. **Direct System Generation:** Generate architectures as structured code e.g., for UML diagrams.
4. **Python-Based Configurator Generation:** Create deterministic system generators encoding design rules and degrees of freedom.
5. **LLM-API-Based Interactive system editor:** Enable system refinement and decomposition in context-aware design environments.

This structured flow forms a repeatable and scalable design approach that can be adapted across different domains.

4.2 Challenges and Limitations

Despite these strengths, several limitations remain. One key challenge is the stochastic nature of LLM outputs. Even with structured prompts and context prompts, outputs can vary across different sessions or model versions. This limits reproducibility and can reduce trust in LLM-generated artifacts. It should also be noted that even if the output is correct, it can

vary since a design solution to a set of requirement, is not necessary unique.

Second, LLMs often produce incomplete or partially incorrect outputs when handling high system complexity. In such cases, prompt refinement or post-processing is necessary.

LLMs are generally quite forgiving if the prompt is not fully complete or correct, which in general, is an advantage. However, a prompt with mistakes, will result in a result of lower quality, something that is not immediately recognized when using the prompt. Therefore, when designing a context prompt that should be reused, it can be hard to detect mistakes.

There is also the aspect of the limitation of the knowledge in the LLM. Working at a general level of design, the general knowledge stored in the LLM can be sufficient, however as we go deeper into details domain specific knowledge is needed that may not be generally available. Here, we have used the context prompt to feed additional information to some degree. However, to take full advantage of existing knowledge in e.g., a company it is necessary to use alternative approaches such as fine tuning or *Retrieval Augmented Generation*, RAG, to make this knowledge accessible to the LLM.

Finally, validation of generated content remains a manual task. Domain knowledge is essential to filter, adapt, and correct LLM-generated structures. As such, LLMs should be seen as collaborators rather than as autonomous designers.

5 Conclusion

This paper has explored the integration of large language models (LLMs), specifically ChatGPT, into the conceptual and system-level design of aerospace engineering applications. Through structured prompting, the use of micro-templates in context prompts, and script-based configurators, LLMs have been shown to support and partially automate early design tasks that traditionally require extensive manual effort and expert knowledge.

A case study involving a hybrid-electric propulsion system demonstrated how LLMs can generate useful system representations, including UML diagrams and Python-based configurators.

Key contributions of this work include:

- A structured methodology for LLM-based engineering design using reusable context prompts with embedded examples.
- Demonstration of system configuration workflows supported by Python script generation.
- Integration strategies using APIs for interactive, tool-supported system decomposition and refinement.

Overall, the findings highlight the value of LLMs as assistants rather than replacements for engineers, capable of accelerating design iteration and supporting complex system synthesis within constrained design spaces.

References

- [1] Petter Krus. Large language models in aircraft system design. In *33rd Congress of the International Council of the Aeronautical Sciences (ICAS)*, Florence, Italy, 2024.
- [2] Petter Krus. Ai-augmented design of fluid power systems using large language models. In *Proceedings of the JFPS International Symposium on Fluid Power*, Okayama, Japan, 2024.
- [3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:1877–1901, 2020.
- [4] Alejandro Pradas, Petter Krus, Panarotto Massimo, and Oskar Isaksson. Large language models in complex system design. In *Design 2024 – 18th International Design Conference*, Dubrovnik, Croatia, 2024.
- [5] Ye XIE, AI SAVVARISAL, Antonios TSOURDOS, Dan ZHANG, and Jason GU. Review of hybrid electric powered aircraft, its conceptual design and energy management methodologies. *Chinese Journal of Aeronautics*, 34(4):432–450, 2021.
- [6] Qianyu Zhou, Shaofeng Huang, Jiasen Wu, Yuxuan Zou, Aman Agrawal, Rohit Singh, and Percy Liang. Prompt-as-declaration: A declarative language for prompt programming. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Appendix

Here is an example of a context Prompt, GDP, for hybrid electric aircraft. For clarity and re-usability it is written in Markdown format.

```
# Aircraft Propulsion System Configuration Guide
*Hybrid-electric architectures with gas turbines, fuel cells, batteries, and electric machines.*

---

## Components

### Gas Turbine
- **Alias:** 'GasTurbine'
- **Ports:** 'MR'
- **Color:** LightCoral
- **Description:**
  Converts chemical energy ('dotPM') from a fuel source (Fuel Tank or Hydrogen Storage) into rotational power ('MR'). Each 'MR' port can connect to only one component.

---

### Fuel Cell
- **Alias:** 'FuelCell'
- **Ports:** 'EL'
- **Color:** LightGreen
- **Description:**
  Converts chemical energy ('dotPM') from Hydrogen Storage into electric power ('EL').

---

### Propeller
- **Alias:** 'Propeller'
- **Ports:** 'MR', 'MX'
- **Color:** LightBlue
- **Description:**
  Converts rotational power ('MR') into thrust (linear power output).

---

### Gearbox
- **Alias:** 'Gearbox'
- **Ports:** 'MR'
- **Color:** LightBlue
- **Description:**
  Combines or distributes rotational power ('MR') from multiple sources to a single output ('MR').

---

### Electric Motor Generator (EMG)
- **Alias:** 'EMG'
- **Ports:** 'EL', 'MR'
- **Color:** LightGreen
- **Description:**
  Converts electric power ('EL') from a Battery into mechanical rotational power ('MR') or vice versa. Each 'MR' port can connect to only one component.

---

### Electric PowerManagement
- **Alias:** 'EPM'
- **Ports:** 'EL', 'MR'
- **Color:** LightGreen
- **Description:**
  Converts and regulates electric power ('EL').

---

### Battery
- **Alias:** 'Battery'
- **Ports:** 'EL'
- **Color:** LightGreen
- **Description:**
  Stores and provides electric power ('EL'). Allows multiple connections.
```

Connection Principles

- **Port Matching:**
 - Connect ports of the same type between components.
 - 'EL' ? 'EL'
 - 'QM' ? 'QM'
 - Use '-down-' for direction.
- **Dashed Connections:**
 - Use dashed lines ('-[dashed]-') for control signals.
- **Mandatory Connections:**
 - Only include relevant components in a configuration.

Energy Storage Requirements

- System must include at least one energy storage: **Hydrogen Storage**, **Fuel Tank**, or **Battery Pack**.
- Gas turbines must connect to a chemical energy source.
- At most **one type** of chemical energy source allowed in addition to the battery.

General Requirements

- At least one continuous power path from storage to each propeller.
- Two independent propulsion drives and two propellers (per **14 CFR 25**).
- No single point of failure; ensure redundancy throughout.

Power and Control Separation

- Distinguish **power-distribution components** from **storage components**.
- Controller manages operations via 'ctrl' signals; monitors battery SOC.
- External references may be included for control parameters.

Validation Rules

- Only **one type** of chemical energy source allowed (Fuel Tank **or** Hydrogen Storage).
- **Aircraft Body** must be included and connected.
- Each **Electric Motor/Generator** must have its own dedicated Battery pack.
- Gas Turbines must connect to a valid fuel source.
- For **14 CFR Part 25** compliance:
 - System must include at least **two propellers**.
 - Two power paths from the energy source are required.
- Ensure **no single point of failure**.

Solution Steps

1. Define system components from user input.
2. Validate the complete architecture against rules.
3. Generate **PlantUML diagram** for configuration.

Micro Templates

Basic Propeller

Description: A minimal system with a propeller connected to the aircraft body.

```
PlantUML:
@startuml
' Define the components with ports
component "Aircraft Body\nPMX, Aero, states" as AircraftBody1
#LightBlue
component "Propeller\nPMR,PMX" as Propeller1 #LightBlue
' Connections
Propeller1 -down- AircraftBody1: PMX- PMX
' Diagram Title
title Aircraft System with Aircraft Body and Propeller
@enduml
```

```

### Battery to Motor
**Description:** Connection of a battery to a motor-generator and a
propeller.

PlantUML:
@startuml
component "Battery\nPEL, SOC" as Battery #LightGreen
component "Electric Motor Gen\nPEL, PMR, ctrl" as ElMotorGen
#LightGreen
component "Propeller\nPMR,PMX" as Propeller #LightBlue
component "Gearbox\nPMR1,PMR2" as Gearbox #LightBlue
Battery --right-- ElMotorGen: PEL- PEL1
ElMotorGen --down-- Gearbox: PMR- PMR1
Gearbox --down-- Propeller: PMR- PMR
title Aircraft System with Electric Motor and Propeller
@enduml

---

### Turbo Prop
**Description:** A gas turbine connected via a gearbox to a
propeller.

PlantUML:
@startuml
component "Propeller\nPMR,F" as Propeller1 #LightBlue
component "Gearbox\nPMR1,PMR2" as Gearbox1 #LightBlue
component "FuelTank\nPdotM" as FuelTank1 #LightCoral
component "GasTurbine\nPdotM, PMR, ctrl" as GasTurbine1
#LightCoral
FuelTank1 --right-- GasTurbine1: PdotM - PdotM
GasTurbine1 --down-- Gearbox1: PMR- PMR1
Gearbox1 --down-- Propeller1: PMR2- PMR
title Aircraft System with Gas-Turbine and Propeller
@enduml

---

### Hybrid Electric with Fuel Cell
**Description:** Hybrid-electric configuration with a fuel cell,
motor-generator, battery, and controller.

PlantUML:
@startuml
component "FuelCell\nEL" as FuelCell1 #LightGreen
component "Hydrogen Storage\nQM" as HydrogenStorage #Turquoise
component "Electric Motor/Generator\nEL, MR" as EMG1 #LightGreen
component "Electric PowerManagement\nEL" as EPM1 #LightGreen
component "Battery Pack\nEL" as Battery1 #LightGreen
component "Controller\nctrl" as Controller1 #LightYellow
HydrogenStorage --down-- FuelCell1: QM
FuelCell1 --down-- EPM1 : EL
EPM1 --down-- Battery1 : EL
Battery1 --down-- EMG1: EL
Controller1 -.-[dashed]- FuelCell1: ctrl
Controller1 -.-[dashed]- EMG1: ctrl
Controller1 -.-[dashed]- Battery1: SOC Monitoring
title Hybrid Electric Aircraft Propulsion System with Fuel Cells
@enduml

---

## Final Solution Steps

- Validate the complete architecture against the **validation rules**
once more. Make sure that all components used are defined in this prompt.

```