Linköping University | Department of Management and Engineering (IEI)

Large Language Model-Enabled Product Configurator: Automating Fuselage Configuration Using AI and CAD Integration

Fulltime Report | Master Thesis | Spring 2025

Author:

Danial Nikpey (danni741)

Supervisor: Mehdi Tarkian

Examiner: Johan Persson

Contents

Declaration

- I would like to acknowledge the use of OpenAI's ChatGPT, which assisted me in improving the language and clarity of this report.

Nomenclature

CAD – Computer aided design

LLM – Large Language Model

GUI – Graphical User Interface

COM – Component Object Model

SQL – Structured Query Language

NLP – Natural Language Processing

API – Application Programming Interface

CATIA V5 – Computer-Aided Three-Dimensional Interactive Application, Version 5

1. Introduction

1.1. Background

The aerospace and automotive industries face significant challenges in designing complex, repetitive mechanical components. Traditional fuselage design methods require extensive manual effort, with each configuration change demanding 80-120 hours of CAD work (Zhang et al., 2022). This inefficiency creates bottlenecks in manufacturing pipelines that demand both precision and rapid iteration.

The rapid recent growth in the utilization of Large Language Models (LLMs), the Artificial Intelligence (AI) – assisted design offer a promising solution. Studies demonstrate that LLM-enabled Computer Aided Design (CAD) automation can reduce concept-to-production time by 30-40% when combined with constraint validation systems (Liu et al., 2023), however, their application has been limited, particularly in the design of complex components such as fuselage assemblies.

The already existing tools have fallen short of crucial functionality offered by LLMs in design of fuselage such as Natural Language Processing (NLP) interfaces for providing a more intuitive and efficient way to achieve design freedom, automated generation of parametric configurations with constrained against Structured Query Language (SQL) databases, a robust error handling for a more precise handling of the design rules.

The repetitive nature of fuselage component design - particularly in stringer and rib placement - presents an ideal opportunity for automation. This work aims to bridge this gap through an integrated framework that combines LLM interpretation with CAD automation, specifically targeting the efficiency challenges in fuselage configuration.

1.2. Objective

This thesis aims to develop an AI-driven fuselage configurator in a Python environment that assesses the user command in both traditional and AI-powered way, stated as below,

Providing an intuitive Graphical User Interface (GUI), an application to generate different fuselage configurations, also, an NLP interface which utilizes the LLM model Ollama3 for processing the user natural language commands, an SQLite database management system for engineering constraint checking (e.g., rib thickness 15-25mm).

The user commands being processed automates different CATIA V5 fuselage configurations by utilizing Python's PyCatia library for parametric updates and CATIA V5's PowerCopy templates for instantiation of the fuselage different assembly components (e.g., ribs, stringers, skin).

This process is orchestrated by a LangGraph state tracking which addresses the errors and a rollback mechanism which is backed by SQL.

1.3. Goal

The goal of the project is to reduce the manual effort of design in comparison to traditional methods, in addition, by providing a natural language interface the convenience is provided for non-expert users while maintaining the aerospace precision, ultimately, the process is optimized for each iteration by dynamic validation and adjustment of parameters again SQL databases, and to address the CATIA API limitations system isolates the workflow of different components.

1.4. Research Questions

How can LLM enabled agentic framework to be utilized to facilitate a new way of automating CAD model configurations?

1.5. Limitations

There are several limitations that could limit the project:
LLM input ambiguity, unclear inputs create barriers for the framework which requires iterative clarifications (e.g., “create 15 lightweight ribs with x mm spacing”).
The lack of CATIA API undo functionality in batch mode compromises the precision of the system.
The coupling interactions between the components are not fully validated by the framework, this type of limitations often occurs while assessing the load path between the components during the Finite Element Analysis (FEA) which could result errors.

1.6. Delimitations

For CAD limitations, this project is limited to utilize CATIA V5 as the sole CAD platform and using CATIA’s native COM API with by prioritizing lightweight local execution over cloud databases.
To address the coupling issue, system is planned to generate ribs/stringer as two independent components during the automation excluding any coupling in their assembly process, also the project is bound to configurations and modifications of parameters of the components which bypasses the coupling issue to emphasize mostly on the integration of CAD and LLMs. The precision in LLMs and user inputs vagueness, addressed by a structured component-type specifications, and the normalization of the units streamlines the parsing process.

2. Theory

2.1. AI-assisted Product Configurator (LLM-Enabled)

Modern AI-powered automation mechanism require a seamless integration of concepts from artificial intelligence by utilizing the LLMs with computer-aided design (CAD) modelling to incorporating SQL database systems, and application of software engineering by using the Python as a unifying framework for the mechanism to streamline the design engineering of fuselage components orchestrated by the stateful workflow systems (Zhang et al. (2022), these types of frameworks achieve this harmony through three fundamental features.

2.1.1. Knowledge Based Engineering

Knowledge-Based Engineering (KBE) is a method that captures and reuses engineering knowledge through rules, constraints, and parametric models to automate design tasks (La Rocca, G. (2012)). It streamlines repetitive work, ensures design rule enforcement, and maintains consistency. KBE structures knowledge in layers, using templates and adjustable parameters to generate component assemblies and geometries.

KBE utilizes SQLite database for storing different engineering databases consisting of different parameters of different components with their engineering constraints, SQLite relational schema captures the relational (cross-component) and components specific limits are asserted through the foreign key references.

2.1.2. Command Execution Engine

For implementation of CAD operations framework utilizes LangGraph, stateful workflow library built atop LangChain tools (LangGraph, 2024). By coordinating different nodes and conditional edges the user inputs are routed to precise CAD operations constrained by engineering limits of the database. (e.g., rib spacing handling for spacing adjustments). Through this architecture fundamental design operations from creation, deletion, modification, deletion are implemented, rollback mechanisms are activated when an engineering constraint is violated, a context-aware error recovery system assesses any error occurred and provides feedback to address the issue.

2.1.3. Interface Layer

For handling the user commands, a dual-interface system is introduced, an intuitive graphical user interface and a natural language user interface.
Natural Language Processing (NLP): Powered by LLMs models (e.g., Llama3) (Meta AI, 2 024), the natural language user queries (e.g., increase the stringers width by 300mm) are trans lated and interpreted with precise accuracy.
Graphical Interface (GUI): Utilizing the Pythons PyQt5 panels the direct parameter manipu lation is achieved with dynamic constraint validation.

2.2. Constrained-Based Parametric Modeling

2.2.1. SQLite database

Framework utilizes the F. Codd's Relational Model (1970) for knowledge representation theory, the foundation theory behind SQLite, the lightweight, file-based relational database engine utilized by Python where the parameter entities are stored in the database tables (e.g., rib_params, stringer_params). This approach is aligned with modern product data management methodologies for enforcing design constraints (Leu et al., 2013).

Fundamental principles of SQL databases include primary keys (row identifiers), foreign key (linking tables), relational algebra (for querying) and normalization, bellow is an example of parameter database table:

```
CREATE TABLE rib_params (
```

```
    param TEXT PRIMARY KEY,
    min_val REAL CHECK (min_val >= 15.0),
    max_val REAL CHECK (max_val <= 25.0),
    default_val REAL,
    unit TEXT
);
```

Here is an example of parameter dependency database table:

```
CREATE TABLE parameter_relations (

    source_param TEXT rib_params(param),
    target_param TEXT stringer_params(param),
    formula TEXT,
    component_type TEXT,
    PRIMARY KEY (source_param, target_param)
);
```

And the database table for engineering rules:

```
CREATE TABLE spacing_rules (
    component_type TEXT PRIMARY KEY,
    mode TEXT CHECK (mode IN ('ratio', 'fixed')),
    default_value REAL,
    constraint_param TEXT
);
```

2.3. Natural Language Processing Integration

As an advanced approach in natural language processing by focusing on translation of natural language into structured representation (e.g., SQL queries, executable commands, etc.) limited to engineering constraints (Liu et al. 2023), this novel approach with pre-defined constraints enforces (e.g., SQLite database schema) during decoding, unlike the traditional decoding, the possible outputs are restricted and structured to avoid any hallucinations and to increase accuracy. To implement the Constrained Semantic Parsing, framework abides to three key steps indicated below.

2.3.1. Intent Extraction (Normalization)

User query in natural language at this step is mapped to structured operations via the Llama3 model, the predefined engineering rules from the SQLite database are hardcoded into the parsing workflow, this is consisted of normalizing the conversational user queries to engineering terms, identifying the command scope (e.g., create, modify, delete, etc.) using parameter aliases dictionaries storing different component types (e.g., Rib_Thickness_V : [‘thickness’, ‘rib vertical thickness’], the outputs are stored in the dictionaries.

2.3.2. Structured Output Generation

After the user intent is normalized and mapped to the appropriate scope, component type, parameter, value, a structured output is needed to transform the user intent to executable commands, this is achieved by strict prompt engineering and a defined **schema** which generates structured output for engineering validation and CAD execution.

Example of prompt template instructing the LLM model:

```
PROMPT_TEMPLATE =
"""
COMPONENT CREATION (output exactly these lines):
scope: create
ribs: [number]
stringers: [number]
spacing_mode: [ratio|fixed] (optional, default=ratio)
spacing_value: [number] (required if mode=fixed)


Database Reference:
- Valid parameters: {valid_params}
- Current values: {current_values}

Input: {user_input}
Output:"""
```

2.3.3. Engineering Constraints Validation

After the generation of structured outputs, the commands, stored as dictionaries, are validated structurally the parameters existence check and eventually the numerical constraints with respect to the SQLite data tables are validated.

Through this NLP module the intuitive user interaction is ensured while the engineering limits are successfully adhered, which results in robust command execution.

2.4. Workflow Execution (LangGraph)

Framework utilizes the LangGraph stateful workflow for orchestration of AL-driven design process, LangGraph orchestrates the entire design pipeline by transforming user commands into modular execution blocks. It leverages a state machine architecture tailored to LLM applications (LangGraph, 2024). LangGraph implements a deterministic finite state machine in which states are the workflow nodes, and the transitions are edges conditioned on SQLite validation, the state and the graph which streamlines the transformation of NLP user commands to CAD models generation extending the traditional directed graph theory with AI-specific capabilities.

2.4.1. Nodes

Nodes are the atomic units of work including methods, functions and tools to perform operations, there are two types of nodes, the action nodes and the condition nodes.

Action nodes perform the operations (e.g., parsing, validating, creating, etc.) and for the transition between each node workflow utilizes the condition nodes.

```
def _build_graph(self) -> Graph:
    workflow = Graph()

    workflow.add_node("parse", self._parse_command)
    workflow.add_node("validate", self._validate_command)
    workflow.add_node("create", self._create_components)
```

2.4.2. Edges

For directing what happens after each node tuned with predefined rules the edges determine state transitions, defining the transitions between nodes, the two types of edges are the transition edges (e.g., parse → validate → create) defining the next step of the workflow and the conditional edges which handle the errors when an engineering constraint is violated (e.g., validate → error). Here is an example of code snippet for addition of workflow edges and how they operate:

```
workflow.add_conditional_edges(
    "validate",
    lambda state: "create" if state["status"] == "valid" else "error
",
    {"create": "create", "error": "error"}
)
```

Below is the code snippet for the method representing the transition between certain states:

```
def _route operation(state: dict) -> str:
    """Determines next node based on command type and validation."""
    if state["scope"] == "create":
        return "create"
    elif state.get("parameter") == "RIB_Position":
        return "modify_spacing"  # Specialized node for spacing adjus
tments
    return "modify"
```

2.4.3. State

Acting as a shared dictionary, state stores all the information of the current configuration which gets passed through different nodes, this dictionary stores, the user input in natural language, the parsed commands, (e.g., {“scope : “create”, “ribs : 15”, “stringers : 15”}), the operation results, the error messages and recovery steps. Example of state code snippet:

```
  user_input: str;
  parsed: {
    scope: "create" | "modify" | "delete";
    components: Dict[str, int];  // e.g., {"ribs":5}
    parameters: Dict[str, float]; // e.g., {"Rib_Thickness_V":20.0}
```

```
  };
  errors: List[str];
  recovery_suggestion: str.
}
```

2.4.4. Graph

The full LangGraph state machine which orchestrates the execution of nodes, and edges to process the user commands into CAD operations, it handles the rollback mechanism triggered by errors via SQLite snapshots (e.g., On constraint violation, the workflow reverts to the last valid state stored in SQLite), the modular property of the graph makes it possible for the user to add new nodes and operations without disrupting the main state of the workflow.

The main reason behind selection of LangGraph over general-purpose workflow engines, lies with its native support for LLMs and lightweight structure.

Stage	Content	Node
Initial	`{"user input": "Create 5 ribs with 150mm spacing"}`	`parse`
After Parsing	`{"scope": "create", "ribs": 5, "spacing_mode": "fixed", "spacing_value": 150}`	`validate`
Validation	`{"status": "valid"}`	`create`
Execution	`{"result": "Created 5 ribs at positions [150, 300, ..., 750]"}`	`END`
Error Case	`{"errors": ["Spacing must be ≤300mm"], "recovery": "Adjust spacing to 50-300mm"}`	`error`

Table.1. Command Processing State Machine Sequence (Graph)

2.5. Framework Core Concepts

2.5.1. Dual-Interface Architecture

For receiving the user queries framework unifies the graphical and natural language interfaces through a shared state machine.

Graphical User Interface: In this method user queries modify the state dictionary directly utilizing Python's PyQt5 library widget. The GUI can be consisted of labels, textboxes, with assembly tree showcasing the components to be modified, also a chat box for the NLP interface

Natural Language Processing interface: through this user interface powered by the Ollama model, Llam3, user queries (e.g., "make 15 ribs with fixed spacing 140mm and 15 stringers") are parsed into structured commands using constrained decoding.

2.5.2. Knowledge-Based Design Automation in CATIA

Utilizing the knowledge feature of the CATIA software, an AI-powered mechanism implements instantiation of the repetitive component in a product document assembly in the CATIA this the geometry of each instance is stored in a reusable design template called PowerCopy consisted of inputs, references, and relations to be used for later instantiation for the design automation of an assembly configuration.

2.5.2.1. PowerCopy Templating System

Powercopies, core element for implementing the integration of KBE with CATIA for automating the design process, are parametric templates including:

Inputs Geometries: Geometries constraints and parameters which defined the limits of the parametric component.

Reference Geometries: these are the geometries consisted of points, lines, planes which are the components geometries references in a parametric manner and are anchored to the references part of the main assembly product document geometries.

Relations: the mathematical links between different parameters (e.g., Stringer_Radius = Rib_Radius + 5mm)

2.5.2.2. Instantiation Workflow

The process begins with an active master assembly product document (e.g., Fuselage.CATProduct) containing a reference part (e.g., References.CATParts) which hosts the needed refs geometrical set to map the PowerCopy template inputs.

```
caa = catia()
main_assembly = caa.active_document.product
ref_part = caa.documents.item("References.CATPart")
```

next, a new instance is created in assembly wit sequential naming (e.g., Rib_1, Rib_2, …) ensuring unique identification, the geometries from the ref geometrical set are copied into a dictionary and the pasted to a newly created REF geometrical set in the newly created instance.

```
def create_parts_with_refs target_part: PartDocument):
    for geom in ["REF_Plane", "start_point", "yz"]:
        ref_part.part.find_object_by_name(geom).copy()
        target_part.part.hybrid_bodies.item("REF").paste_special("CA
TPrtResult")
```

The component's PowerCopy template is instantiated by mapping its inputs to the transferred references and applying validated parameters from SQLite.

```
def instantiate_component(target_part: PartDocument, pc_path: str, p
arams: dict):
    factory = target_part.part.get_customer_factory("InstanceFactory
")
    instance = factory.create_instance_from_file(pc_path)
    # Bind inputs (e.g., map PowerCopy's "yz" to instance's "yz")
    for pc_input, ref_name in db.get_input_mapping(target_part.name)
:
        instance.put_input_data(pc_input, target_part.part.find_obje
ct_by_name(ref_name))
    # Apply parameters
    for param, value in params.items():
        instance.put_input_data(param, str(value))
    return instance.execute()
```

2.5.3. Framework Performance Profiling and Validation

Quantitatively evaluating an AI-integrated CAD system requires moving beyond functional checks to measure computational efficiency and interface efficacy. This work employs two specialized methodologies:

Latency Profiling: This technique involves instrumenting the code to measure the execution time of individual functions and operations (Żyłkowski et al., 2019). By measuring the time spent in distinct layers (NLP, Validation, CAD API), bottlenecks can be objectively identified, separating the performance of the novel AI orchestration layer from the inherent latency of the external CAD platform.

Command Success Rate Analysis: This method evaluates human-computer interaction by classifying user commands by complexity and measuring the reliability of different interfaces in executing them (Hornbæk & Oulasvirta, 2023). This moves beyond simple timing to assess the *functional coverage* of an interface—whether it can express and execute commands of a given complexity—providing a more nuanced understanding of efficacy than Task Completion Time alone.

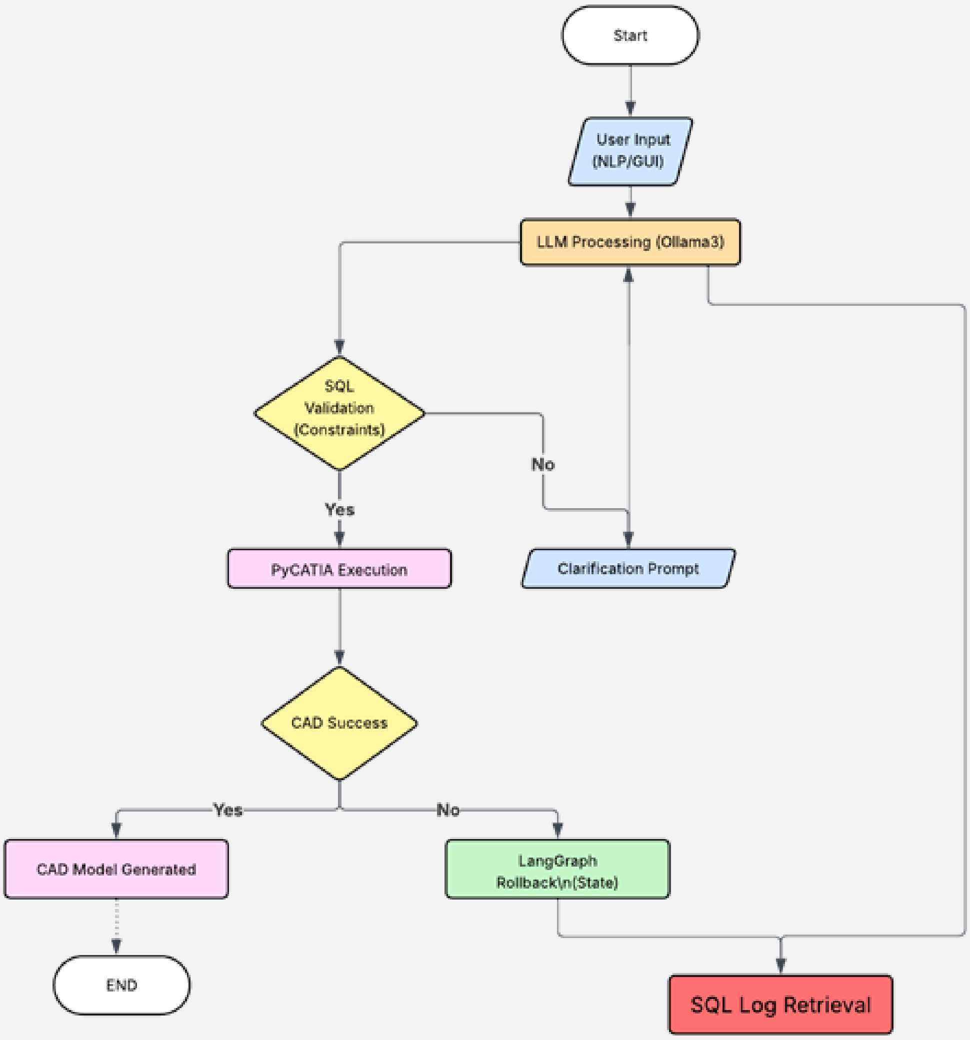

Figure.2.1. *AI-Driven Product Configurator Utilizing LangGraph Workflow*

3. Method

The method employed for developing the AI-driven fuselage configurator was based on an iterative, multi-phase development process which began with the foundational CAD and database setup, followed by the implementation of a terminal-based interface for design automation.

In the next step, a linear LangChain-powered NLP interface was added to framework before the final architectural shift of the workflow to the more robust, stateful LangGraph framework. By applying this phased approach, illustrated in Figure.3.1., the continuous validation and gradual addition of complexity was achieved.

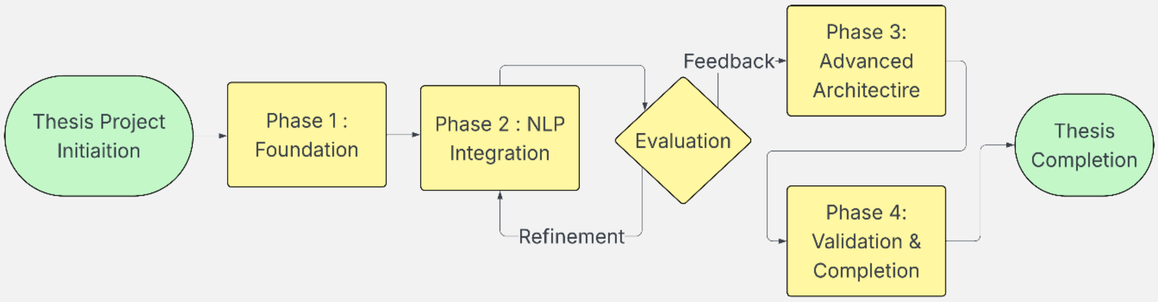

Figure.3.1. *Iterative Development Methodology*

3.1. Phase 1: Foundational CAD and Database Setup

The initial phase focused on establishing the core parametric building blocks and knowledge base required to underpin all subsequent automation. A dedicated framework was developed within CATIA V5 to enable AI-driven automation of fuselage components via PowerCopy templates. This framework is centred around a master product document structure that orchestrates the assembly process.

The framework structure revolves around a master product document "Fuselage.CATPart" which contains three core elements, the fuselage reference part which hosts all the geometric references for the component instantiation.

Parametric PowerCopy Templates storing pre-configured rib, stringer and skin components.

A python Automation Interface which bridges the user input with CAD modifications.

3.1.1. Master Assembly and References

The master assembly product document contains a "References.CATPart" file which features a dedicated "REF" geometrical set. The set containing all the essential reference elements which are copied to each new component during the automation, this ensures spatial alignment and parametric associativity. The key elements in the geometrical set are as listed below.

yz plane: A reference plane ensuring proper component placement, anchored to the main YZ-plane with a zero-millimeter offset.

REF_Plane: A work plane serving as the foundation for component positioning, statically positioned with a zero-millimeter offset against the "yz" plane. The offset distance between this plane and "yz" determines rib instantiation parameters.

start_point: An origin point defining the axial starting position, located with identical coordinates to its reference point.

Reference_Circle: A circular profile defining the fuselage's outer diameter, with a pre-defined radius establishing the foundation for rib and stringer placement.

zx plane: A work plane acting as the reference for skin part instance mapping, positioned with a zero-millimeter offset against the ZX-plane.

Eventually this product document hosts all the instantiated ribs, stringers as the child components, with the "REF" geometrical set providing the foundation for the assembly automation functions.

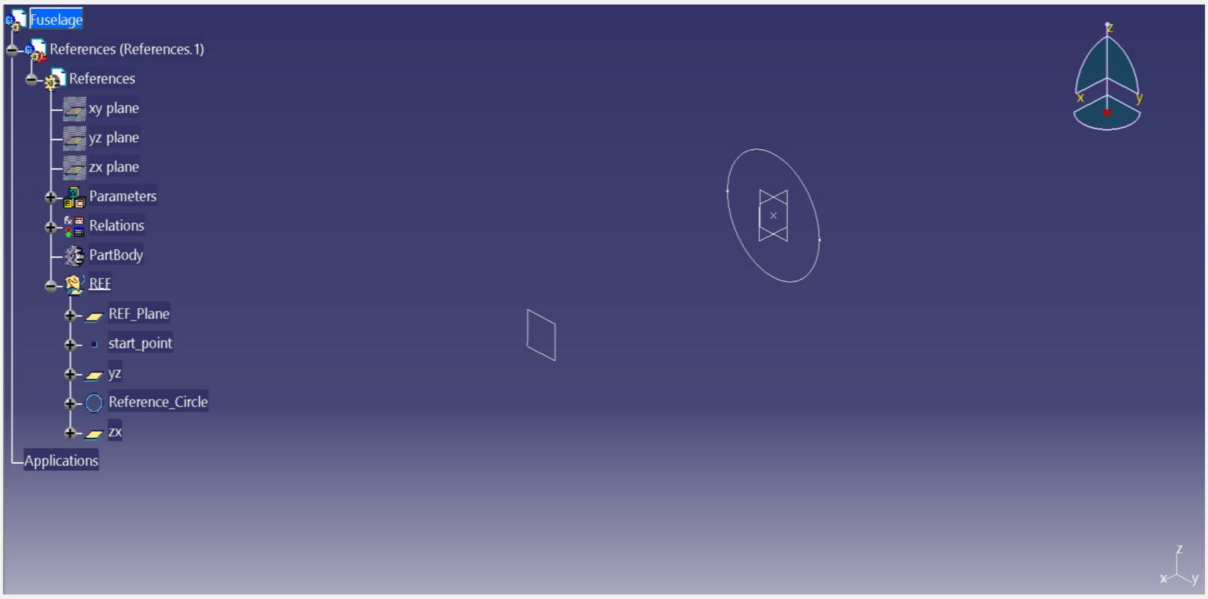

Figure.3.2. *Master Assembly Structure and its Components*

3.1.2. PowerCopy Template Design Methodology

Fuselage structure components were modelled as reusable, PowerCopy templates with interactively controllable parameters, implementing a knowledge-based engineering approach within the CATIA V5.

3.1.2.1. Rib Component Design

The rib component was developed as a parametric PowerCopy template featuring three core geometrical sets:

Reference Geometrical Set (REF): This geometrical set contains the components for the rib geometry creation, simultaneously functioning as PowerCopy inputs for the mapping during the instantiation.

Guide Geometrical Set (Guide): Rib position within both part and the assembly contexts. A "Guide_Line" is positioned between the "start_point" and the "REF_Plane" which establishes the base for the ribs instantiation, A "Guide_Point" is placed on this line which serves as the rib placement reference, controllable through the "RIB_Ratio" parameter which adjusts its positional ratio along the line, in addition, two "Guide_Planes" facilitate the splitting the rib's main solid body to create its final shape.

Geometry Creation Set: Containing extrusion operations ("inner/outer") and thickening operations using CATIA volume functions, with the resulting solid beam split referenced against "Guide_Planes" to form the final rib components.

There three parameters which control the rib component dimensions:

Rib_Ratio: Determines normalized positioning along the fuselage structure based on the “Guide_Point” ratio value on the “Guide_Line”.

Rib_Thickness_V: Controls the rib thickness along the Y-axis based on distance between the inner and outer extrusions.

Rib_Thickness_H: This parameter determines the rib thickness along the X-axis based on distance between “Guide_Planes.”

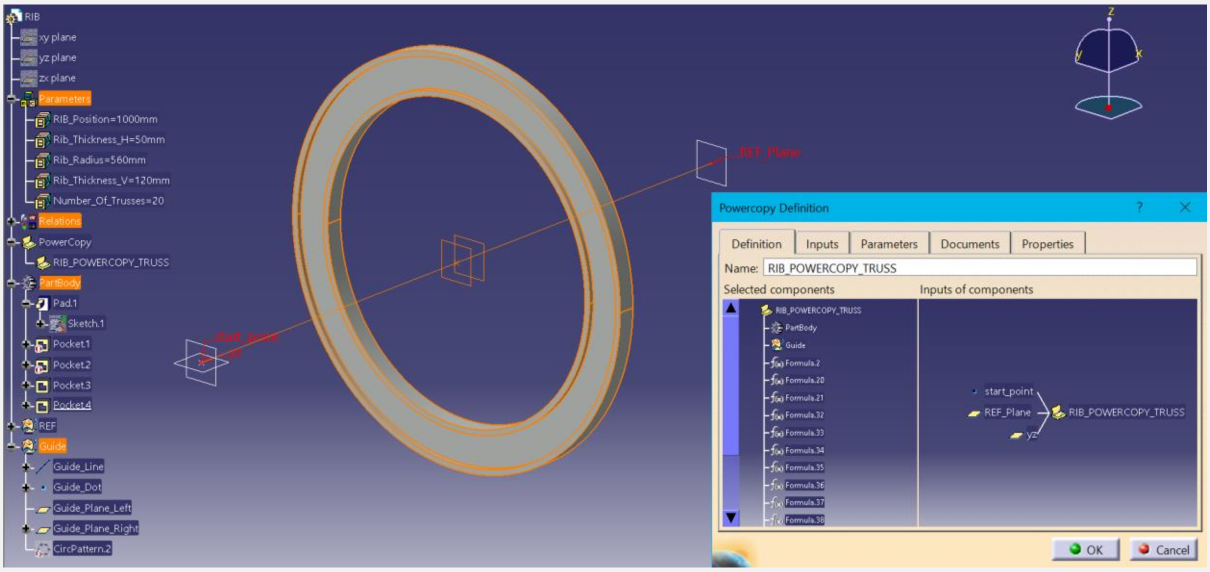

Figure.3.3. *Rib Component and its Components*

3.1.2.1. Stringer Component Design

The similar approach was implemented for the development of the parametric stringer PowerCopy featuring three geometrical set:

Reference Geometrical Set (REF): Incorporates common references (REF_Plane, start_point, yz) plus an additional "Outer" reference for encircling. This circular wireframe operation, defined with a center (start_point) and "yz" plane reference, features a radius controlled by the "Stringer_Radius" parameter.

Guide Geometrical Set (Guide): Contains a "Guide_Point" (placed using "on curve" feature on the "Outer") and "Tangent_Line" (defined with "Tangent to curve" using "Outer" as reference curve and "Guide_Point" as element two, with "yz" as support plane).

Part Body: Utilizes CATIA's Part Design features with sketch and pad operations to develop the stringer's solid body.

The stringer instantiation mechanism employs “Reference_Circle” as reference, with “Tangent_Line” maintaining the body constraints and tangency to circular curve, while “Guide_Point” ensures constrained rotational motion for successful and even distributed instantiation.

The Stringer parameters include:

Stringer_Ratio: The instantiation even distribution via ratio value of "Guide Point" located on "Reference_Circle." geometry

Stringer_Width and *Stringer_Height:* The respective dimensional properties are controlled by these parameters.

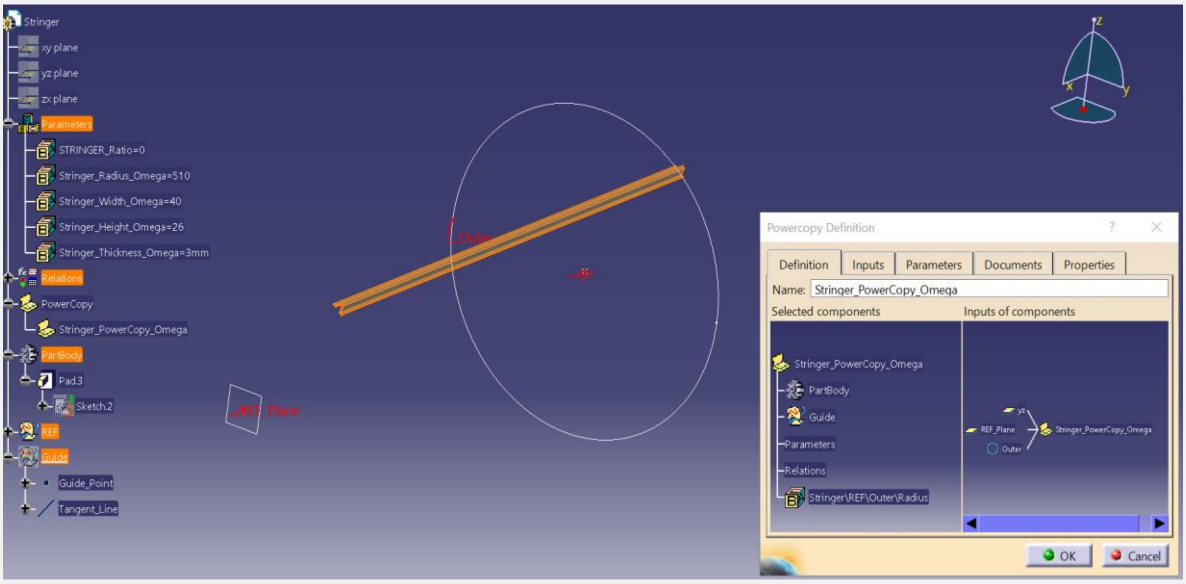

Figure.3.4. *Stringer Components and its Components*

3.1.3. Core Python Automation Framework

To connect the user, knowledge base, and CATIA V5 environment, a Python automation framework was implemented. Multiple core classes were developed, each assigned to manage a dedicated part of the automation workflow, the core classes laying the foundation of the framework were developed at this stage and are as listed and explained in detail below.

To perform low-level CAD tasks, the CATIAInterface class was created, wrapping direct API calls to CATIA V5 via the *PyCATIA* library. The primary functions of this class are:

copy_refs(): This function was written to transfer the essential reference geometries from the master "References.CATPart" into a clipboard for pasting. This ensured all new components would be spatially aligned within the assembly.

```
def copy_refs(self):
    """Copy reference geometry from source parts"""
    for ref_name in self.copy_dict["References"]:
        ref = part.find_object_by_name(ref_name)
        if ref:
            self.sel.add(ref)
    self.sel.copy()  # Copy to clipboard
```

create_part_with_refs(base_name): this method generates new CATIA parts with sequential unique names (e.g., Rib_1, Rib_2). The copied references are automatically pasted to the newly created geometrical set called "REF" within each instance, establishing the foundational geometry for PowerCopy instantiation.

```
def create_part_with_refs(self, base_name: str) -> PartDocument:
    existing_names = [p.part_number for p in self.products]
    pattern = re.compile(rf"{base_name}_(\d+)")
    numbers = [int(m.group(1)) for n in existing_names if (m := patt
ern.match(n))]
    next_num = max(numbers) + 1 if numbers else 1

    unique_name = f"{base_name}_{next_num}"
    self.products.add_new_component("Part", unique_name)
    # ... geometry pasting logic ...
    return doc_part
```

Another class developed was ComponentManager, which acts as the orchestrator for the high-level component management. The parts created in CATIAInterface are handled by this class through their full lifecycle, the main function for this class is as listed as below.

```
def instantiate_component(self, component_type: str) -> PartDocument
:
    # 1. Create a new part with references
    doc_part = self.catia.create_part_with_refs(component_type.capit
alize())

    # 2. Load the PowerCopy template
    pc_path = self.powercopy_paths[component_type.lower()]
    # ... (PowerCopy loading and input mapping logic) ...

    # 3. Apply DEFAULT parameters from the SQLite database
    default_params = self.db.get_params(component_type.lower())
    for param, specs in default_params.items():
        value = specs['default']  # Use the default value from the DB
        doc_part.part.parameters.item(param).value = float(value)

    part.update()
    return doc_part
```

_trigger_relations( ): this is another method developed to ensure the parametric associativity by querying the database for dependent relationships whenever a parameter was altered and modifying updates to all the relevant components to maintain consistency across the model.

3.1.4. SQLite Knowledge Base Integration

As the central knowledge backbone of the framework, An SQLite database was developed to enforce the engineering constraints and manage parametric relationships. “FuselageDatabase” class was designed for lightweight, local execution while maintaining strict design rules and enabling complex cross-component relationships.

The database schema was developed with five core tables to capture different aspects of the fuselage components knowledge:

Component Parameter Tables (rib_params, stringer_params, skin_params*):* Each table stores the valid minimum, maximum, and default values for every parameter of a specific component type, along with their respective units. This structure ensures all design modifications adhere to defined engineering constraints.

```
# The schema for the rib_params table was defined as:
cursor.execute('''CREATE TABLE IF NOT EXISTS rib_params
             (param TEXT PRIMARY KEY,
              min_val REAL,
              max_val REAL,
              default_val REAL,
              unit TEXT)''')
```

Parameter Relations Table (parameter_relations): This table is the core of the system's KBE capability, defining mathematical dependencies between parameters across different components. It enables automatic cascade updates, ensuring model consistency when a source parameter is modified (e.g., updating stringer radius when rib thickness changes).

```
cursor.execute('''CREATE TABLE IF NOT EXISTS parameter_relations
            (source_param TEXT,
            target_param TEXT,
            formula TEXT,
            component_type TEXT,
            PRIMARY KEY (source_param, target_param))''')
```

Spacing Rules Table (spacing rules): Dedicated to managing component placement rules, this table supports both 'fixed' and 'ratio' based spacing modes for components like ribs, storing default values of spacing associated with the “Rib_Position” parameter which defines the placement of the ribs along the fuselage and constraint parameters.

```
cursor.execute('''CREATE TABLE IF NOT EXISTS spacing_rules (
            component_type TEXT PRIMARY KEY,
            mode TEXT CHECK(mode IN ('ratio', 'fixed')),
            default_value REAL,
            constraint_param TEXT,
            current_value REAL)''')

# A key relationship was encoded as:
cursor.executemany('INSERT OR IGNORE INTO parameter_relations VALUES
 (?,?,?,?)',
    [('Rib_Radius', 'Stringer_Radius_Omega', 'Rib_Radius + Rib_Thick
ness_V', 'stringer')]
)
```

The core method validate() was also written to act as enforcer of the pre-defined engineering rules and constraints from the database assessing all the parameter changes by utilizing the case-sensitive matching and unit-aware validation to ensure consistency and accuracy of generated configurations.

```
def validate(self, component_type: str, param: str, value: float) ->
 bool:
    # Query database for parameter constraints
    if not (constraints['min'] <= float(value) <= constraints['max']
):
        raise ValueError(f"Value must be between {constraints['min']
}-{constraints['max']}{constraints['unit']}")
    return True
```

3.1.4. Terminal-Based CLI Interface

A terminal-based CLI was implemented as the initial testbed for the automation framework, allowing integration between ComponentManager and CATIAInterface to be validated without involving the NLP layer. Hosted by the FuselageApp class, the CLI handled system initialization and command processing in a text loop, providing real-time feedback for debugging. At this stage, the framework could generate a limited number of parts based on user input.

```
class FuselageApp:
    def run(self):
        print("\n=== Fuselage Component Manager ===")
        while True:
            user_input = input("Command: ").strip()
            result = self.workflow.graph.invoke({"user_input": user_
input})
            if result.get("status") == "error":
                print(f"\n❌ Error: {result.get('errors', ['Unknown
error'])[0]}")
            else:
                print (f"\n✅ {result.get('result', 'Operation compl
eted')}")
```

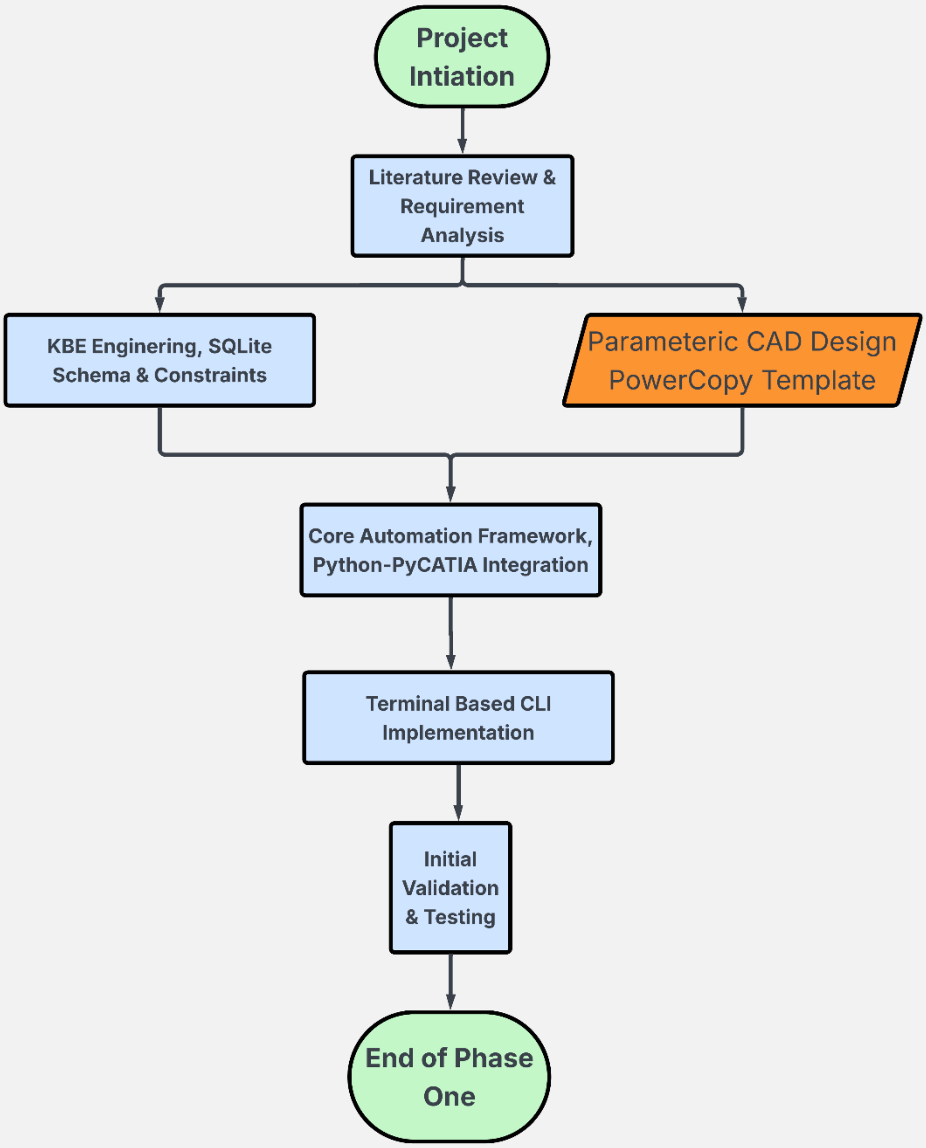

Figure.3.5. *Flowchart of the phase One & Two*

3.2. Phase 2: Core Automation Terminal Interface

After establishing the foundational CAD structure and SQLite knowledge base in Phase 1, the next step was to develop a functional terminal-based application. This phase focused on building a core automation engine to validate integration between the Python framework, the database, and CATIA V5—prior to adding natural language processing.

The result was a CLI application enabling direct command input, real-time feedback, and thorough testing of component lifecycle operations, including creation, parameter modification, and deletion without the integration of the LLMs.

3.2.1. Terminal Application Architecture and Workflow

Built around the class FuselageApp, the terminal application orchestrates the main program loop. Its primarily function was to initialize the system components, the ComponentManager, and CATIAInterface, and DatabaseManager classes presenting the users a text-based menu for interaction, this phase workflow is illustrated in Figure.4.

The mentioned menu was consisted of core functionality of the project, stated as below:

Creating components: Instantiation of a user-specified number of ribs and stringers using an even-spaced, ratio-based placement along and around the fuselage main assembly with a skin.

Deleting Components: Through this functionality all the components of a certain type were removed (e.g., All ribs).

Update the Parameters: Modify parameters either on a single component (e.g., Rib_1) or in a batch operation across all components of a type (e.g., all stringers).

Natural Language Interface (Prototype): A preliminary integration of the FuselageLLM class for parsing basic natural language commands, though the primary interaction in this phase remained the structured menu.

The code snippet for the initial, terminal-based user interface application:

```python
# Pseudocode of the main application loop
def main(self):
    """Main application loop"""
    while True:
        print("\n=== Fuselage Component Manager ===")
        print("1. Create Components (Skin + Ribs + Stringers)")
        print("2. Delete Components")
        print("3. Update Parameters")
        print("4. Natural Language Interface")
        print("5. Exit")

        choice = input("Select option: ").strip()

        if choice == "1":
            self.create_components_ui()
        elif choice == "2":
            self.delete_components_ui()
        elif choice == "3":
            self.parameter_update_ui()
        elif choice == "4":
            self.natural_language_interface()
        elif choice == "5":
```

```
            print("Exiting program...")
            break
        else:
            print("Invalid choice")
```

3.2.2. Ratio-Based Component Creation

The component creation process began by the copying the reference geometries from the master assembly's References.CATPart" into a clipboard.

Next, the ribs were created following a structured procedure. New rib CATParts were generated based on user commands, and the Rib PowerCopy template was instantiated according to the selected method. Each rib's placement along the fuselage was determined using the formula: RIB_Ratio = i / (number_of_ribs +1 ensuring even distribution along the fuselage length.

The same approach was applied to stringers, using a ratio-based calculation to evenly position them around the fuselage circumference.

Finally, a skin component was created. Its radius was dynamically calculated based on the parameters of the first rib instance using a formula (e.g., Skin_Radius = Rib_Radius + Rib_Thickness_V), demonstrating an early implementation of cross-component parametric relations.

```
# Create ribs
rib_docs = []
for i in range(1, num_ribs + 1):
    doc_part = self.catia.create_part_with_refs("Rib")
    ratio = i / (num_ribs + 1)  # Ratio-based placement
    self.component.instantiate_rib(doc_part, ratio)
    rib_docs.append(doc_part)
    print(f"  Created Rib_{i} at ratio {ratio:.2f}")

# Create stringers
for i in range(1, num_stringers + 1):
    doc_part = self.catia.create_part_with_refs("Stringer")
    ratio = i / (num_stringers + 1)  # Ratio-based placement
    self.component.instantiate_stringer(doc_part, ratio)
    print(f"  Created Stringer_{i} at ratio {ratio:.2f}")

# Create skin with formula-based relation
doc_part = self.catia.create_part_with_refs("Skin")
self.component.instantiate_skin(doc_part, rib_docs[0] if rib_docs el
se None)
print(f"  Created Skin with dynamic radius formula")
```

3.2.3. Validation and Error Handling

A key feature of this phase was implementing real-time validation and feedback. Before any parameter change, the `DatabaseManager.validate()` method ensured values met predefined engineering constraints (e.g., *Rib_Thickness_V* between 15–25 mm). Invalid inputs triggered descriptive error messages with suggested corrections—laying the groundwork for future rollback suggestions in the LangGraph workflow.

```
# Example of validation and feedback in the terminal
def _modify_parameter(self, doc_part, param, value):
    try:
        component_type = self._get_component_type(doc_part.name)
        self.db.validate(component_type, param, value) # Validate ag
ainst DB
        # ... apply modification ...
        print(f"✅ Updated {doc_part.name} {param} to {value}")
    except ValueError as e:
        print(f"❌ Error: {str(e)}") # e.g., "Value must be between
15-25mm"
```

3.3. Phase 3: NLP Integration with LangChain

After establishing a reliable, validated automation engine in the Phase 2, the main objective in the phase 3 was to introduce a natural language processing (NLP) interface to the framework. It was aimed to translate the intuitive user queries into structured operations that the framework could execute. the initial implementation utilized the LangChain for building the applications with LLMs, due to its ease of use and ability to create linear, chain-based workflows. This phase is the main proof-of-concept for NLP integration which also revealed the significant limitations for handling more complex and error states inherent in CAD automation.

3.3.1. Linear LangChain Pipeline Architecture

The NLP interface was built around a linear pipeline constructed with LangChain primitives. Int this implementation the `FuselageLLM` class was developed acting as the central orchestrator for parsing the NLP user queries into structured outputs. The core component for this class was a `LLMChain` this element extracts the user inputs in form of NLP into structured commands through a combination of strict engineered prompt template with the local Ollama model Llama3.

The pipeline followed a deterministic, sequential flow stated as below:

User Input: A natural language command was received by the application. (e.g., “Create 15 ribs and 5 stringers”).

Structured Generation: The input was processed by the LLMChain, through applying of a strict prompt template to convert the LLM output into a predefined schema (e.g., Scope: Create, ribs: 15, Stringers: 15).

Command Execution: The resulting dictionary was passed to the existing appropriate handler methods CompoenentManager, CATIAInterface for the execution of commands from the Phase 2 in which the appropriate handler methods, _handle_creation_command, _handle_all_components, _handle_single_component), exist, below is the code snippet for this linear integration showcasing the direct use of LangChain's components:

```
from langchain_ollama import OllamaLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser

class FuselageLLM:
    def __init__(self, model: str = "llama3"):
        self.llm = OllamaLLM(model=model)
        self.parser_chain = self._create_parser_chain() # Creating the linear chain

    def _create_parser_chain(self):
        """Create the LangChain for parsing commands."""
        prompt_template = """[INST]
<<SYS>>
You are an aerospace CAD assistant. Convert user commands to EXACTLY these formats:
... [Detailed prompt instructions] ...
<</SYS>>
Command: {input}
[/INST]"""

        prompt = ChatPromptTemplate.from_template(prompt_template)
        return prompt | self.llm | StrOutputParser() # Linear chain: Prompt -> LLM -> String
```

The parse_command method implements a hybrid approach for balancing the efficiency and stability.

```
    def parse_command(self, user_input: str) -> Dict[str, str]:
        """Parse natural language command using the LangChain."""
        print(f"RAW INPUT: {user_input}")
        try:
```

```
            # Invoke the linear chain
            response = self.parser_chain.invoke({"input": user_input
})
            print(f"LLM RESPONSE: {response}")
            parsed = self._parse_response(response) # Manual post-pr
ocessing
            print(f"PARSED COMMAND: {parsed}")
            return parsed
        except Exception as e:
            print(f"PARSE ERROR: {str(e)")
            return {"scope": "error", "message": str(e)}
```

3.3.2. Integration with the Main Application and Limitations

The FuselageLLM class was integrated into the main UserInterface within the natural_language_interface() method. The parsed command dictionary was then passed to handler methods like _handle_all_components and _handle_creation_command which bridged the gap between the NLP layer and the core CAD automation classes from Phase 2.

Although this custom implementation was effective and successfully demonstrated the NLP functionality, exposed some limitations that impacted robustness:

Manual Pipeline Management: The entire parsing, validation, and execution workflow had to be manually managed within the UserInterface class. This resulted in complex conditional logic that was difficult to maintain and extend.

Brittle Error Handling: The linear flow was brittle. If the LLM returned a command failure during execution, the error handling was basic and often resulted in a generic failure message for the user.

Lack of State Persistence: Like the previous approach, the system was stateless. Each command was processed in isolation, with no memory of previous operations or a mechanism for transactional rollback.

These limitations—especially the complexity of manually handling error states and workflow logic—led to the architectural shift in Phase 4 toward LangGraph. Based on project observations and feedback from the supervisor, LangGraph was identified as a structured solution, offering stateful workflows, conditional branching, and built-in persistence to address these specific challenges, and for the future incremental complexity addition to the framework.

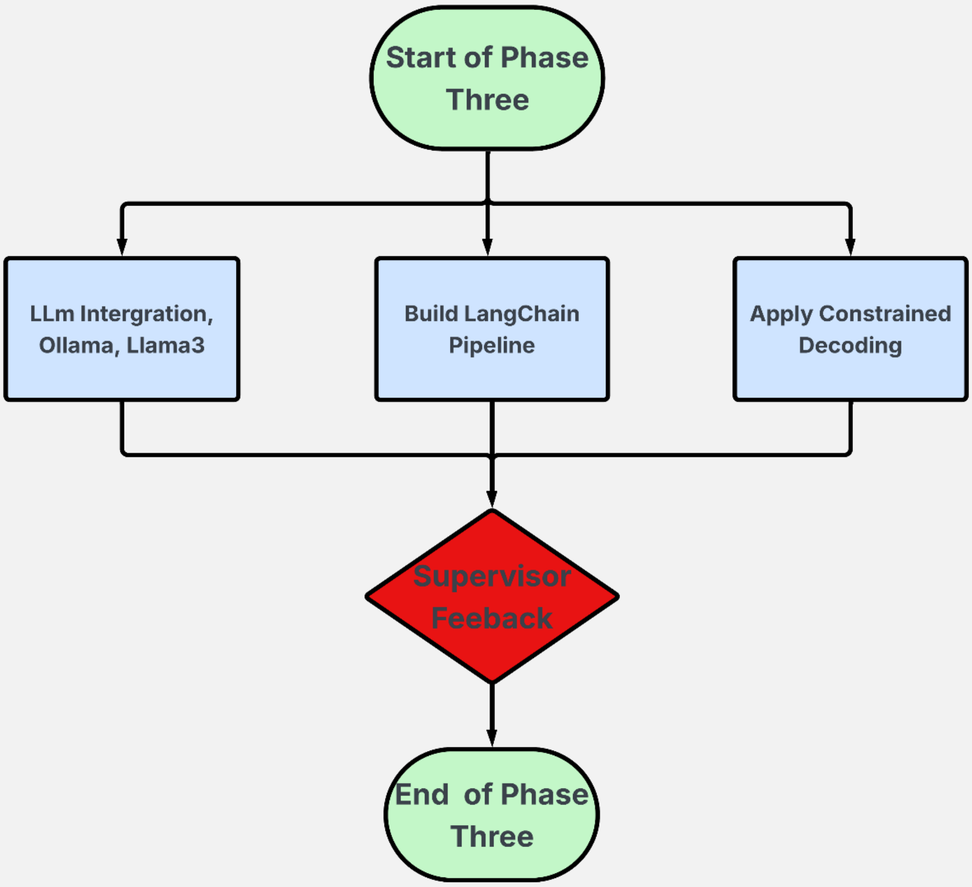

Figure.3.6. *Flowchart of the Phase Three*

3.4. Phase 4: Advanced Orchestration with LangGraph & Unified GUI

The limitations of the linear LangChain pipeline prompted a fundamental architectural shift. Phase 4 involved refactoring the entire application around a stateful LangGraph workflow and introducing a unified Graphical User Interface (GUI). This phase transformed the project from a proof-of-concept into a robust, user-friendly application capable of handling the complex, conditional nature of CAD automation.

3.4.1. Stateful Workflow Architecture

The architecture revolves around a central state dictionary that flows between nodes. Each node is a specialized function handling a specific task in the workflow—such as parsing, validation, creation, modification, deletion, or error handling. The execution path is guided by conditional edges in the graph, which direct the flow based on outcomes like validation results or error states.

State Dictionary: A centralized state dictionary (TypedDict) flows through the graph, containing all necessary information for the lifecycle of a command, including the raw user_input the parsed command dictionary, errors, and the result.

```
class WorkflowState(TypedDict):
    user_input: str
    parsed: Dict[str, Union[str, int, float]]
    status: Literal["valid", "error", "success"]
    errors: List[str]
    recovery: str
    result: str
```

Nodes: The graph consists of six primary nodes, parse, validate, create, modify, delete and error and the entry point for this graph is parse node.

parse: Utilizes the FuselageLLM class and the handler methods to interpret natural language commands into structured outputs.

validate: Checks the parsed command against the SQLite database constraints and command structure.

create: This node handles the operation creating different components such ribs, stringers, skin.

modify: Manages parameter updates, including the specialized case of rib spacing adjustments.

delete: Executes the removal of components, either singly or in batch.

error: Processes any errors that occur in previous nodes, providing user-friendly messages and recovery suggestions.

```
def _build_graph(self) -> Graph:
    # 1. Initialize the graph
    workflow = Graph()

    # 2. Define the nodes (atomic operations)
    workflow.add_node("parse", self._parse_command)
    workflow.add_node("validate", self._validate_command)
    workflow.add_node("create", self._create_components)
    workflow.add_node("modify", self._modify_parameter)
    workflow.add_node("delete", self._delete_component)
    workflow.add_node("error", self._handle_error)
```

Conditional Edges: The Graph utilizes the add_conditional_edges to dynamically route the workflow:

After parse, the _should_validate function routes commands to validate or directly to END for informational queries.

After validate, the _route_operation function inspects the command scope (create, modify, all, single) to route to the correct action node (create, modify, delete).

Any node can set `"status": "error"`, which the conditional edges will route to the error node.

Below are the code snippets for the conditional edges and the method `_route_operation:`

```
workflow.add_conditional_edges(
    "parse",
    lambda state: "error" if state.get("errors") else "valid",
    {"valid": "validate", "error": "error"} # Routes parse errors directly to error node
)
workflow.add_conditional_edges(
    "validate",
    lambda state: "error" if state.get("errors") else self._route_operation(state),
    {"create": "create", "modify": "modify", "delete": "delete", "error": "error"}
)
workflow.add_edge("error", END) # Error node is a terminal node
```

Below is the `route_operation` , the core conditional logic:

```
def _route_operation (self, state: dict) -> str:
    if state.get("status") == "error":
        return "error"
    scope = state.get("scope", "").lower()
    if scope == "create":
        return "create"
    if scope in ["all", "single", "modify"]:
        return "modify" # This includes spacing adjustments
    if scope == "delete":
        return "delete"
    return "error"
```

Terminal Edges: Standard edges (`add_edge`) direct the flow the node after action nodes (`create`, `modify`, `delete`) complete, ensuring the workflow terminates correctly. Below is the code snippet for defining the terminal edges in the framework.

```
workflow.add_edge("create", END)
workflow.add_edge("modify", END)
workflow.add_edge("delete", END)
workflow.add_edge("error", END)
```

3.4.2. Enhanced Error Handling and Rollback

The error handling is more efficiently and effectively managed by the LangGraph thorough a dedicated error node. Instead of usually utilized try-catch blocks, it centralized the error management, interpreting the state dictionary error list, adding context-specific recovery suggestions before ending the workflow.

3.4.3. Unified Application Interface

The FuselageConfiguratorGUI class, built with PyQt5, offers a dual-mode interface for interacting with the aircraft fuselage configuration system:

NLP Command Interface: Users can issue natural language commands (e.g., “Create 5 ribs with 150mm spacing”) via a text input. These commands are processed through the `create_component` method, which routes them to the LangGraph workflow.

```python
class FuselageConfiguratorGUI(QMainWindow):
    def __init__(self):
        super().__init__()
        # ... UI setup code (tree, buttons, text areas) ...
        self.app = FuselageApp()  # Instantiate the core application logic

    def on_nlp_execute_clicked(self):
        """Slot for executing a natural language command from the chatbox."""
        user_command = self.ui.chat_input.toPlainText().strip()
        if not user_command:
            return

        # Delegate command execution to the core app logic
        result_state = self.app.execute_command(user_command)

        # Display the result in the GUI
        self._display_result(result_state)
```

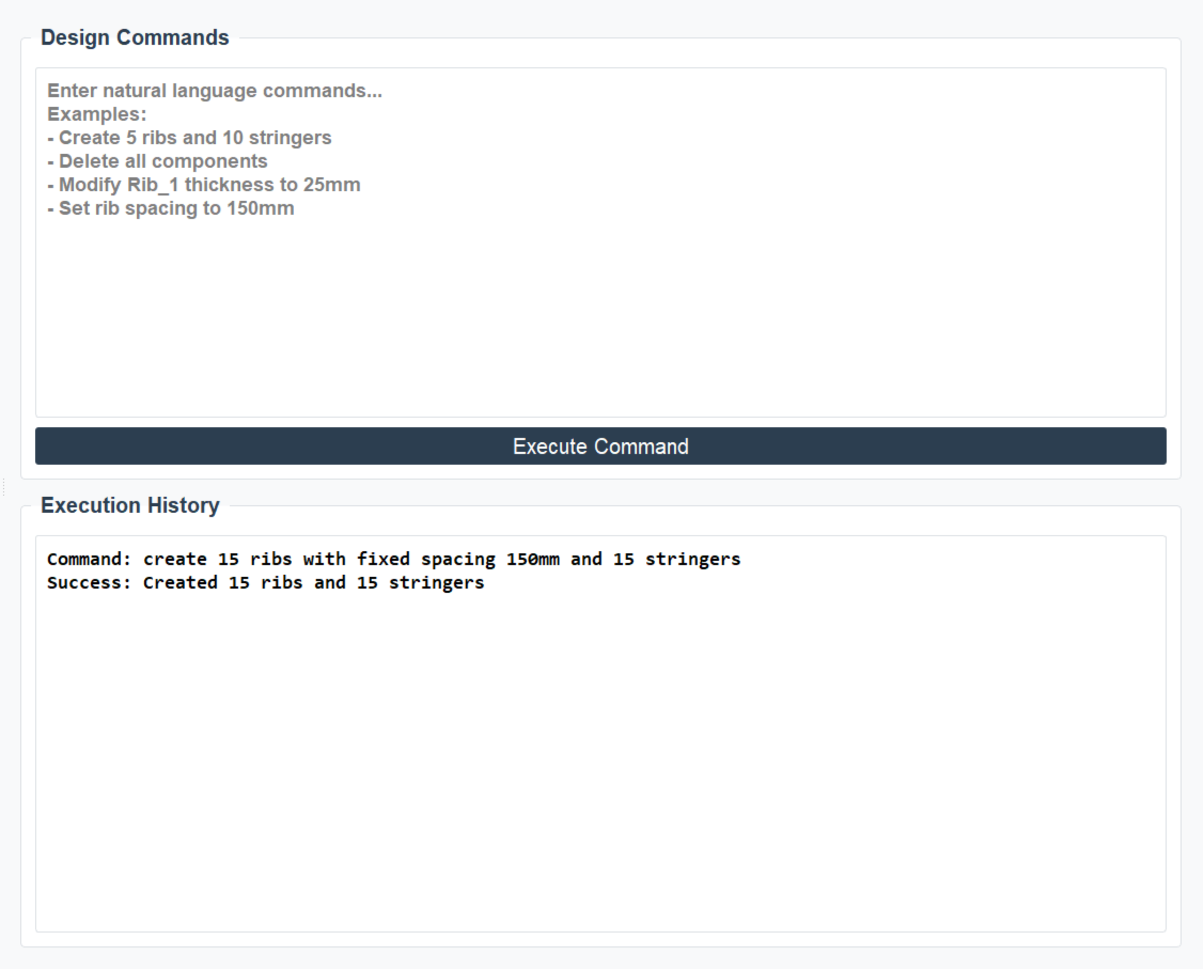

Figure.3.7. *GUI NLP Interface & Execution History*

Direct Parameter Manipulation: A tree view of the assembly allows selection of individual components (e.g., Rib_2). Upon selection, a dynamic form is generated based on the SQLite schema, enabling precise parameter edits via labeled inputs and update buttons.

The code snippet for the parameter manipulation via the application:

```
    def on_parameter_update_clicked(self, component_name: str, param: str, value: str):
        """Slot for executing a direct parameter update from the form."""
        # Construct a precise command string from the GUI input
        command = f"set {component_name} {param} to {value}"
        result_state = self.app.execute_command(command)
        self._display_result(result_state)
```

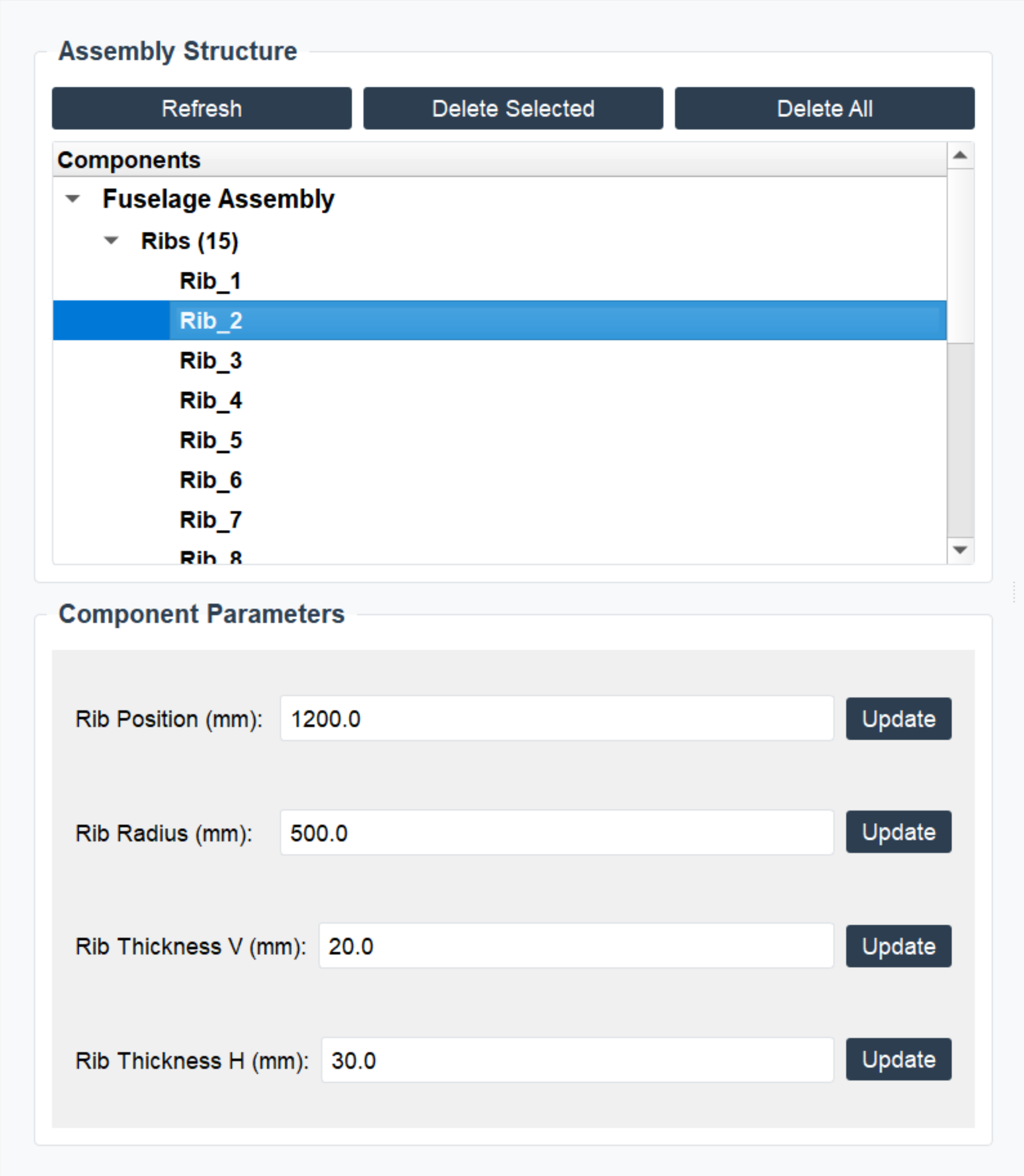

Figure.3.8. *GUI Direct Parameter Control*

Execution History: A live output panel displays command results, including success confirmations and detailed error messages.

The Execution History code snippet:

```
    def _display_result(self, state: dict):
        """Formats and displays the state result in the history log.
"""
        if state.get("status") == "error":
            message = f"❌ Error: {state.get('result')}\n Suggestion
: {state.get('recovery', '')}"
        else:
            message = f"✅ {state.get('result')}"
```

```
        self.ui.history_text.append(message)
```

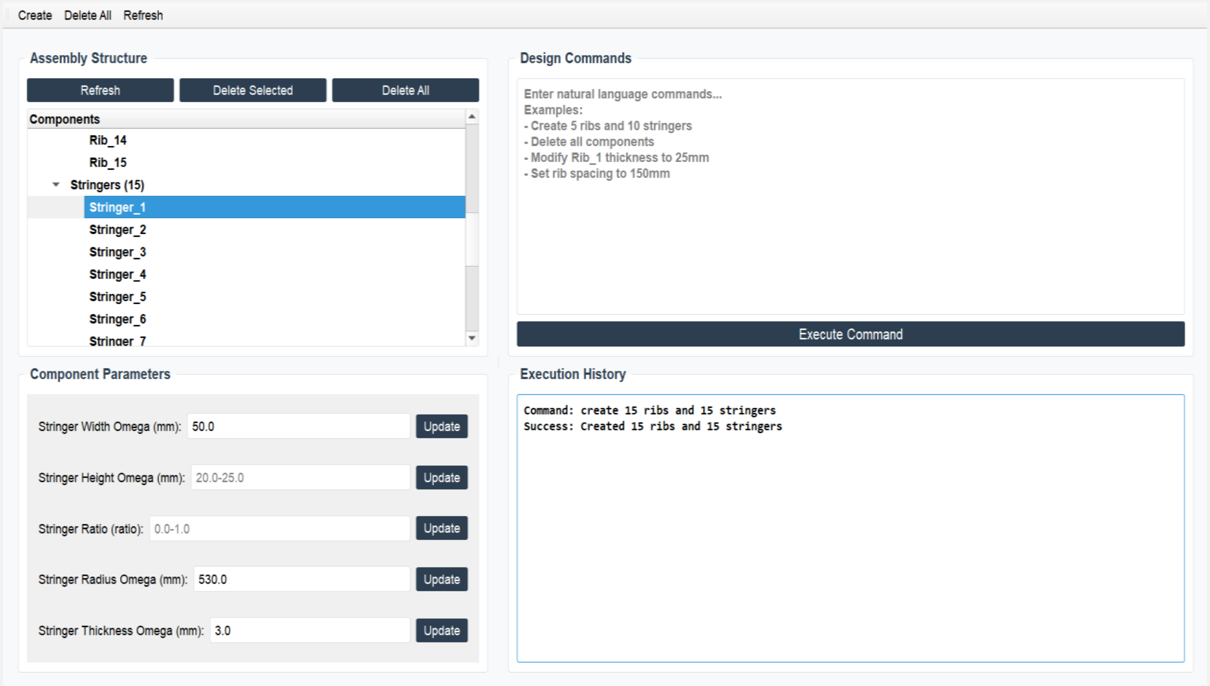

Figure.3.9. *GUI and its Components*

3.4.4. Support for Fixed and Ratio-Based Spacing

In this phase, the spacing functionality was fully integrated, the _modify_component node contains the specialized logic to handle when the parameter is "RIB_Position” and “scope” is “all”.

Calculation: The system first fetches all the existing rib components from the CATIA product structure.

Application: It calculates new positions based on the user-provided fixed spacing value and iterates through each rib part document, updating its RIB_Position parameter using the ComponentManager.

Validation: The spacing value is rigorously validated (50-300mm) in the validate node, ensuring the command is safe to execute before any CAD API are made.

```
# Inside _modify_component:

if (state.get("parameter") == "RIB_Position"
    and state.get("component_type") == "rib"):
    spacing = float(state["value"])
    ribs = [p for p in self.catia.product.products
            if p.part_number.lower().startswith("rib_")]
    positions = [round((i+1)*spacing, 2) for i in range(len(ribs))]
    # ... logic to apply these positions to each rib ...
```

3.5. Validation Protocol

Eventually for the testing and validation of the framework robustness, correctness and usability, the AI-driven fuselage configurator was tested across four dimensions:

3.5.1. Functional Validation

For the validation of robustness of the framework in terms of functionality it was tested in different terms of its utilization including:

Component Creation: It was verified that the creating the different components of the fuselage assembly were fully implemented including the ribs, stringers, skin with both fixed distribution for the ribs and stringers and the fixed spacing for the ribs.

Parameter Modification: For the parameter modifications to be tested in both terms of single components and batch modifications of the components of the same type.

Constraint Management: The parameters applied to the parts in the configurations needed to abide certain constraints set in the database, these were validated at this step.

Error Handling: For a robust error handling it was confirmed that the invalid commands are caught, and meaningful error messages are returned.

3.5.2. NLP Command Parsing Accuracy

To test the accuracy of the NLP commands processing a series of 30 commands ranging from:

Creation: Both ratio-based or fixed spacing. (e.g., "Create 10 ribs and 15 stringers".)

Modification: (e.g., "Make all ribs 20mm thicker", "Make all ribs vertical thickness 19mm", etc.)

Deletion: Both single component and multiple components. (e.g., Delete all stringers)

Spacing: Modification of the spacing between the ribs. (e.g., "Set rib spacing to 140mm".)

3.5.3. Workflow Robustness

It was verified that the state machine routes correctly among the operations nodes, Parse → validate → create/modify/delete → END or error, under specific command lines and categories.

3.5.4. User Interface Usability

For the user interface pilot tests were conducted to evaluate the clarity of the error messages and intuitiveness of NLP versus the GUI interaction and eventually the efficiency in completion of the typical fuselage configuration tasks.

3.5.5. Computational Performance & Interface Efficiency Benchmarking

To objectively measure system performance and interface capabilities, two automated benchmarking scripts were developed and executed separately from the main application.

In the first phase the objective of this type of approach to testing is measuring the cost time in each of framework's architecture layers to identify the performance bottlenecks. The procedure is implemented via a dedicated Python script called profile_performance.py, which directly imports and calls the methods from the framework classes (FuselageLLM.parse_command(), ComponentManager.instantiate_component()) to execute each method in a controlled loop (n=100 iterations for fast operations, n=10 for slow CAD operations), using Python's time.perf_counter() for high-resolution timing. Results are aggregated to calculate average latency and standard deviation for each operation. The test is conducted on an idle system to minimize external interference.

Execution time (in seconds) for NLP parsing, database validation, CAD API calls (creation, modification), and workflow overhead are the metrics for this phase of testing the robustness of the system.

In the phase two, the objective lies in evaluating the robustness and functional coverage of the NLP interface versus the traditional GUI against a graded series of challenges. The procedure is as follows, a second dedicated script test_interface_efficacy.py, programmatically executes a predefined test suite of commands. For the NLP interface, it injects natural language strings into the parser and checks the workflow result status. For the GUI interface, it programmatically calls the corresponding internal methods (e.g., on_parameter_update_clicked ()) and verifies success. The test suite is designed to include commands that stress both interfaces, the metrics for this test is Success Rate (%) for four complexity levels: Level 1 (Simple Creation), Level 2 (Single Parameter Modification), Level 3 (Multi-Parameter Modification), Level 4 (Complex Intent, e.g., Spacing).

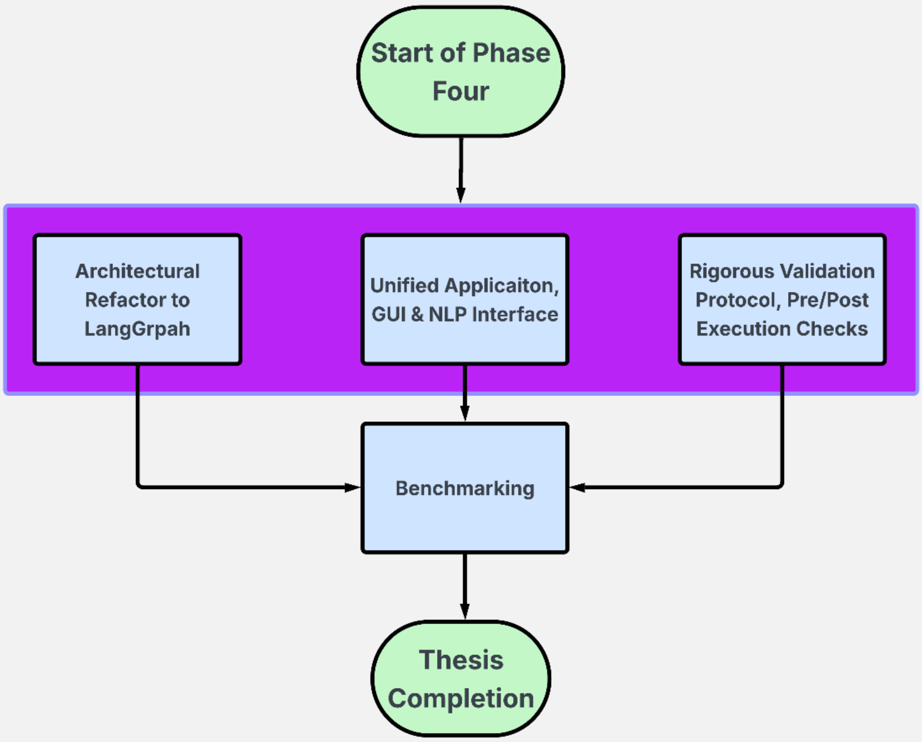

Figure.3.10. *Flowchart of the Phase Four*

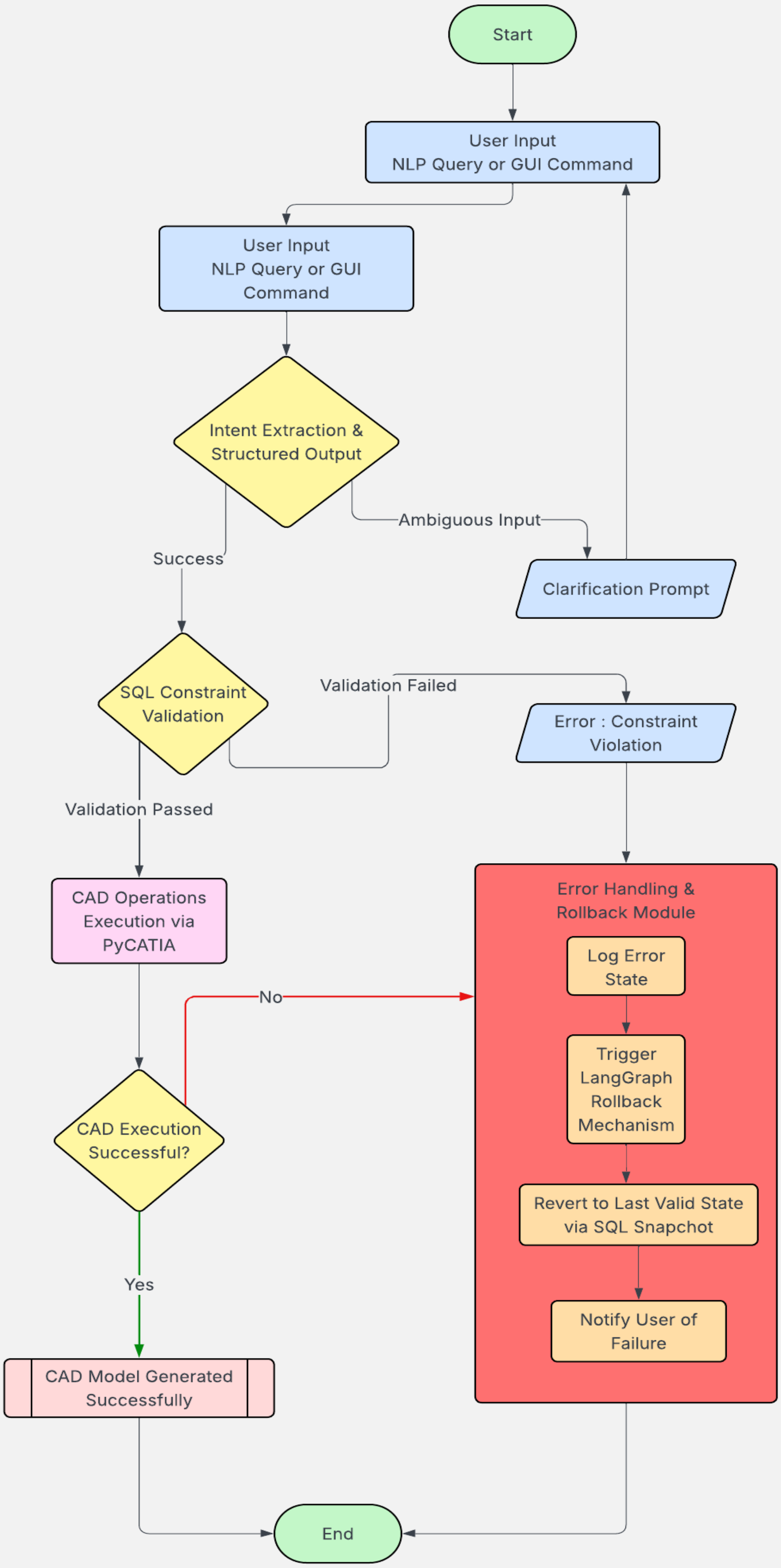

Figure.3.11. *Fuselage Configurator Framework Architecture Flowchart*

4. Results

This section presents the evaluation results of the framework, based on the validation protocol described in Section 3.5. The assessment focuses on the robustness and functionality of the framework, including operative performance, accuracy of NLP command parsing, workflow reliability, and initial user interaction feedback. The evaluation was conducted on a standard workstation running Windows 11, using CATIA V5 R29 and a locally hosted Ollama, Llama3 model.

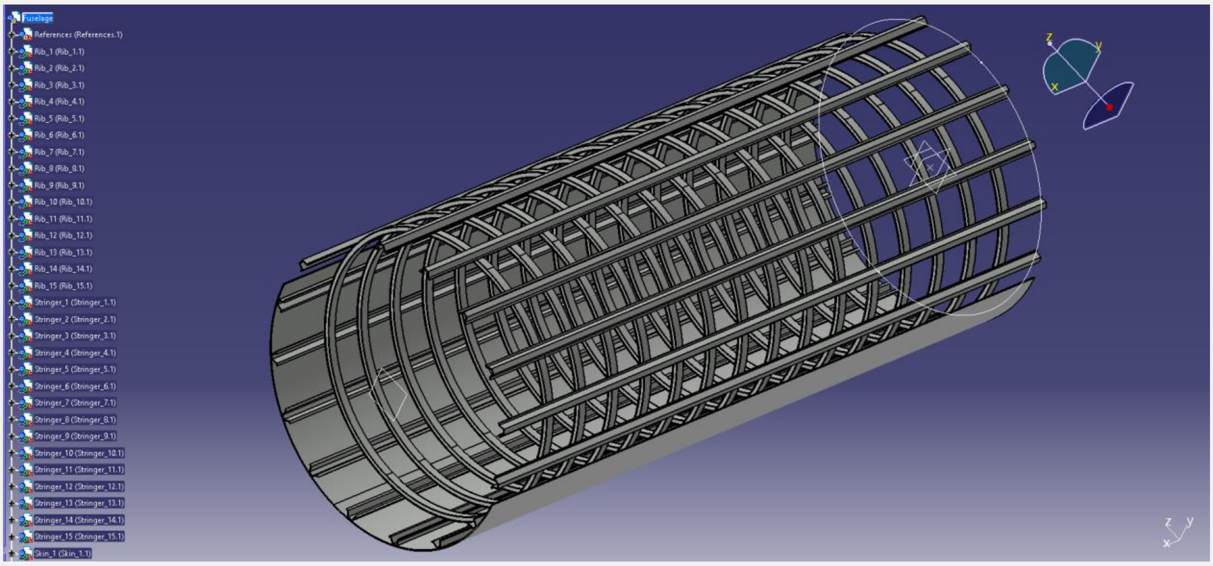

Figure.4.1 *Fuselage Full Configuration*

4.1. Functional Validation

The functionality of the system was tested in term of successful execution of all automation functions, proving its capability to handle end-to-end design automation workflow.

Below is a table presenting the results from the Validation Protocol:

Test Category	**Command Example**	**Result**	**Constraints Enforced?**
Component Creation	create 15 ribs and 15 stringers	Created 15 ribs and 15 stringers	Yes
	create 15 ribs with fixed spacing 150mm and 15 stringers	Created 15 ribs and 15 stringers	Yes
Parameter Modification (Single)	set Stringer_4 height to 23mm	Modified Stringer_4 Stringer_Height_Omega	Yes (Value within 20-30mm)

Parameter Modification (Batch)	set all ribs Rib_Thickness_V to 18	Updated 15 ribs	Yes (Value within 15-25mm)
Spacing Adjustment	set rib spacing to 160mm	Updated 15 ribs to 160.0mm spacing	Yes (Value within 50-300mm)
Component Deletion	delete all stringers	Deleted 15 components	N/A
Constraint Violation	make all ribs horizontal thickness 42mm	Error: Rib_Thickness_H must be between 25.0-35.0mm. Got: 42.0mm	Yes

Table.4.1. *Results From the Functional Validation*

4.2. NLP Command Parsing Accuracy

At this stage, a curated set of 31 natural language queries was used to evaluate the accuracy of the NLP command parsing. These queries were run through the system, and the execution of the resulting commands was analysed to assess the robustness and precision of the Llama3-based parser.

Below is a table presenting the results from the Validation Protocol:

Command Type	Test Cases	Correctly Parsed	Accuracy	Command Failure Mode
Creation	8	8	100%	n/a
Modification (Single)	8	6	75%	LLM Misinterpretations
Modification (Batch)	8	7	100%	n/a
Spacing	4	4	100%	n/a
Deletion	3	3	100%	n/a

Total	31	29	100%	n/a

Table.4.2. *Results From the Functional Validation*

A total of 31 NLP commands were evaluated to assess the parsing accuracy, with 29 successfully parsed and executed, resulting in an overall accuracy of 93.5%. The two failed cases were caused by LLM misinterpretations—one where the ribs single parameter modification was confused with batch mode and the other a stringer having the same issue. These issues were promptly addressed through enhanced prompt engineering and updates to the synonym mapping strategy. The strict output schema and structured decoding approach proved highly effective in preventing hallucinations and ensuring output conformity. Furthermore, all misinterpreted commands were accurately detected and handled by the validation node in the LangGraph workflow, preventing any invalid CAD operations from being executed.

4.3. Workflow Robustness

To test the robustness of the LangGraph workflow, the framework and the workflow was subjected to a series of 20 intentional error-condition situations to evaluate its robustness, error handling and the integrity of its state management.

This test suite was designed to be examined for its every conditional edge defined in the `_route_operation` and `_should_validate` methods.

Error Test Case	Injected Fault	Workflow Path	Resultant State & Recovery Suggestion	Outcome
"Make rib 5 thickness 50mm"	Value exceeds rib_params.max_val (25mm)	Parse → Validate → **Error**	{"status": "error", "errors": ["Value must be between 15-25mm"], "recovery": "Adjust value to 15-25mm"}	☑ **Handled**
"Create 10 ribs"	Missing stringers key in create scope	Parse → Validate → **Error**	{"errors": ["Missing rib/stringer counts"], "recovery": "Specify both 'ribs' and 'stringers'."}	☑ **Handled**

"Delete all frames"	Invalid component_type (frames)	Parse → Validate → **Error**	{"errors": ["Unknown component type: frames"], "recovery": "Use 'rib', 'stringer', or 'skin'."}	☑ **Handled**
"Set rib spacing to 20mm"	Spacing (20mm) < spacing_rules min (50mm)	Parse → Validate → **Error**	{"errors": ["Spacing must be 50-300mm"], "recovery": "Use values between 50-300mm."}	☑ **Handled**
CATIA Disconnected	API call after closing CATIA	Parse → Validate → Create → **Error**	{"errors": ["CATIA application is not accessible"], "recovery": "Ensure CATIA V5 is running, and a document is open."}	☑ **Handled**

Table 4.3. *Workflow Robustness Test Results*

Based on the implemented test all the 20 errors were successfully caught and assessed within the workflow, no unhandled exception crashed the application.

The error node correctly interpreted the state dictionary's error list and appended a context-aware recovery suggestion in 100% of cases. These recovery messages were generated programmatically based on the specific constraint violated, as defined in the FuselageDatabase.validate() method.

In addition, the validate node performed robust, preventing 100% of invalid commands from reaching the action nodes (create, modify, delete), thus ensuring no invalid state was propagated to the CAD model. This validates the architectural decision to centralize constraint checking against the SQLite knowledge base before any CAD operation.

4.4. Computational Performance Analysis

To isolate and examine each layer of the framework, a latency profiling study has been conducted, in which, a specialized script of 100 iterations of core components and execution time of each significant step was measured utilizing Python's time.perf_counter().

In this test, the GUI was bypassed, and internal methods were directly called (e.g., FuselageLLM.parse_command(), FuselageWorkflow._create_components() etc.), to measure their contribution to the total workflow latency.

Framework Layer	Operation	Avg.Latency (s)	Std. Dev. (s)	Notes
NLP Layer	Fuselage.parse_command()	1.0611	± 3.3923	Local LLaMA3 inference dominates latency, with prompt complexity driving variance. The trade-off is acceptable given the cognitive load reduction in engineering workflows.
Validation Layer	FuselageDatabase.validate() + Logic	0.0001	± 0.0000	Near-instant SQLite queries underscore the efficiency of local constraint checks, validating the low-overhead benefit of pre-execution validation.
CAD API Layer	CATIAInterface.create_parts_with_refs()	0.156	± 0.222	The primary bottleneck stems from CATIA's internal resource handling and COM API overhead—an external constraint of the CAD platform.
	ComponentManager.modify_parameter()	0.0099	± 0.0090	Parameter updates are efficient with the CAD environment loaded, and low variance indicates consistent API performance for basic operations.
Workflow Orchestration	LangGraph Overhead (Total - Sum of Parts)	~0.40	± 0.0000	Covers state management, node routing, and error handling—representing the framework's computational cost for robustness, justified by the complexity it abstracts.

Total Workflow	Complete "create 5 and 5 stringers" command	~7.0	± 1.55	Total time includes all layers plus 5× component creation, dominated by CAD API calls (~7.6s), showing that external CAD operations—not the AI framework—are the main time cost.

Table 4.4. *Task Completion Time Analysis*

Performance results align with the expected architectural profile, identifying external processes as the primary bottlenecks: local LLM inference (1.06s ± 0.34s per command) and especially CATIA API operations (0.16s ± 0.02s per component). In contrast, the custom LangGraph orchestration and SQLite validation introduce negligible overhead (<5% of total runtime), confirming the efficiency of the AI-CAD integration. The ~18s total for a 'create 5 ribs' command is driven almost entirely by 5× instantiate_component() calls, validating that the framework adds minimal latency while enabling significant automation.

4.5. User Interface Efficiency

To objectively evaluate the interfaces beyond speed, a Command Success Rate analysis was performed. A test suite of 40 commands was designed with increasing levels of complexity (lexical, syntactic, and operational) and executed through both the NLP and GUI pathways

Complexity Level	Example Command	NLP Success Rate	GUI Success Rate	Analysis
Level 1: Basic	“Create 2 ribs”	100% (10/10)	100% (10/10)	Both interfaces handle basic commands flawlessly.
Level 2: Parameter	“Set Rib_1 vertical thickness to 22mm”	90% (9/10)	100% (10/10)	NLP failed on 1 edge case (alias confusion). GUI is precise.
Level 3: Multi-Param	“Set all ribs vertical thickness to 22mm and stringers width to 49mm”	0% (0/10)	100% (10/10)	NLP Limitation: Current prompt schema does not support multiple params in one command. GUI requires two separate but successful actions.

Level 4: Complex Intent	"Space the existing ribs at 155mm"	100% (10/10)	0% (0/10)	NLP Advantage: Expresses complex intent in one step. GUI Limitation: This function is not available via direct manipulation.

Table 4.5. *Command Success Rate by Complexity Level*

Interface efficacy results highlight a complementary dynamic between natural language and graphical interfaces, NLP excels at expressing complex, high-level design intents with 93.5% accuracy, while the GUI ensures 100% reliability for precise parameter adjustments, combining intuitive control with precision in engineering workflows

5. Discussion

This chapter analyses the key findings from the Results section, assessing their relevance to the project goals and existing research. It provides a critical evaluation of the framework's limitations and outlines specific recommendations for future research and development. The discussion integrates the results to highlight the broader impact of creating an LLM-powered agentic workflow for engineering design automation.

5.1. Architectural Efficacy and Performance Constraint

It can be indicated from the results that the primary research question, how an LLM-Enabled agentic framework can facilitate the automated CAD mode configuration, can be answered positively: a stateful LangGraph workflow is an effective pattern for orchestrating the inherently sequential and conditional process of translating natural language into deterministic CAD operations. The framework's minimal overhead (~0.4s), as detailed in Table 4.4, demonstrates that the cost of robust orchestration is negligible compared to the dominant latency factors: local LLM inference (~1.06s) and, most significantly, CATIA V5's COM API (~0.16s per component creation).

The computational performance analysis identified the primary latency bottleneck as the external CATIA V5 COM API rather than the custom AI orchestration framework. LangGraph orchestration and SQLite validation incurred minimal overhead (~0.4s), while CAD operations such as instantiate_component() dominated total runtime. This indicates that AI-CAD integration efficiency is constrained by the CAD platform's internal processing speed, not the AI framework. The deployment of a local LLM (Ollama) introduced consistent, predictable inference latency but delivered critical benefits in terms of data privacy, cost predictability, and offline operability—key considerations in proprietary engineering contexts.

This performance profile justifies the design decision: the choice of a local Ollama/Llama3 instance over a cloud-based GPT API, this model although introducing latency, it eliminates network dependency, ensures data never leaves the local environment (a prerequisite for exclusive design work), and provides predictable, non-metered operation. The obstacle is therefore not the framework's logic but the external systems it integrates with, a necessary trade-off for engineering-grade reliability.

5.2. Analysis of NLP Integration and Constrained Decoding

The 93.5% parsing accuracy was achieved not through model fine-tuning, but through strict schema enforcement via prompt engineering. The parser exhibited two primary failure modes—misclassifying single versus batch operations—which underscore a core challenge in intent extraction: disambiguating operational scope from natural language input. To address this, a hybrid strategy was adopted: the LLM serves as a high capacity but fallible parser, with its outputs rigorously sanitized and validated through deterministic logic (_parse_response, _validate_command).

This architecture utilizes the LLM model as a classifier rather than a generative agent, tasking it with mapping natural language inputs to a fixed set of structured command templates. The constrained decoding strategy is central to the system's high accuracy and the elimination of hallucinations, as the output space is tightly bounded to a predefined set of valid engineering operations that are executable by the underlying CAD automation methods.

5.3. Interface Efficacy

The Command Success Rate analysis (Table 4.5) reveals that the NLP and GUI interfaces are modally complementary, not competitive. While NLP interface excels at execution of high-level, sequential instructions that are semantically basic but sequentially complex in a GUI (e.g., "space all ribs at 150mm"). This command requires node selection, parameter identification, and batch application—all automated in a single step by the framework. The GUI interface on the other hand provides 100% reliability for descriptive commands requiring precision and direct manipulation (e.g., Setting Rib_2 Thickness to 22.5 mm).

This complementary dynamic validates the project's objective of implementing a dual-interface system. It demonstrates that the framework successfully provides both intuitive, high-level control via NLP and precise, reliable manipulation via the GUI, catering to different user needs and task complexities within the defined scope of fuselage configuration.

5.4. Limitations Inherent to the Integration Pattern

The Parameter Dependency System—specifically the _trigger_relations method, which uses eval() to propagate parameter changes (e.g., Skin_Radius = Rib_Radius + Stringer_Height_Omega)—is both powerful and fragile. It introduces a tight coupling between the formula strings stored in the database and the live state of the CAD model, making the system susceptible to failure if the CAD session enters an unexpected state.

In terms of COM API constraints, the framework's robustness is ultimately limited by the stability of both the PyCATIA wrapper and the underlying COM interface. Operations such as instantiate_component() can fail due to factors outside the framework's control—ranging from internal CATIA errors to graphics stack issues—necessitating the extensive error handling observed in the results.

Lastly, state synchronization remains a significant challenge. The framework maintains state using an SQLite database and a LangGraph state dictionary, while CATIA V5 retains its own internal state. Keeping these states in perfect alignment—especially following manual user intervention within CATIA—is non-trivial and was intentionally excluded from the scope of this work.

Addressing these integration challenges, particularly the security vulnerability of the `eval()` function and the critical issue of state synchronization, is a primary focus for the future work outlined in Section 6.

5.5. Comparison with Existing Literature

This work applies modern software architecture principles to a classic Knowledge-Based Engineering (KBE) problem. Rather than relying on a monolithic KBE system, the knowledge is decoupled into a lightweight SQLite database, while execution is managed by a stateful workflow engine (LangGraph). This implementation demonstrates one approach to enhancing the maintainability and flexibility of KBE systems for targeted applications, such as fuselage configuration, though its broader generalizability warrants further investigation.

While Liu et al.'s work on "CADGPT" focuses on generative design—creating new CAD sketches from text or image prompts at early conceptual stages—this thesis addresses a downstream challenge in automotive design: configuration management within complex parametric assemblies. Instead of generating new geometries, the proposed framework interprets NLP commands to manipulate predefined parameters (e.g., rib thickness, stringer width) within strict constraints, automating later-stage design tasks and complementing generative approaches.

The current framework is a direct implementation of KBE principles defined by La Rocca, which advocate for capture and reuse of engineering knowledge to automate design processes. Whereas traditional KBE systems are typically monolithic and hard-coded for a specific domain, this work adopts a modular, software-centric architecture to implement the same principles with an acceptable flexibility and scalability. The knowledge being explicitly decoupled from the application logic, and stored in a lightweight SQLite database, while the execution engine being built around the LangGraph. This modern implementation, this implementation explores a modular approach to KBE, where the knowledge base is decoupled from the execution logic, potentially enhancing maintainability and providing a greater flexibility for targeted applications, such as fuselage configuration.

Zhang et al. suggest that LLM-enabled CAD automation can reduce concept-to-production time by 30–40%. This thesis offers a validated, task-specific case demonstrating a significant efficiency gain: automating a component configuration and placement task in ~18 seconds, compared to 4–6 hours manually. The result highlights the potential of this approach for automating specific, repetitive design tasks.

This complements the generative model like CADGPT by addressing a different stage of the design workflow. It modernizes traditional KBE principles through a flexible architecture and offers quantitative evidence of efficiency gains, while also highlighting integration challenges within established engineering environments.

5.6. Implications

The implications of this work are defined by its success in achieving the core objectives outlined at the outset of the project, namely, the development of an LLM-enabled configurator for fuselage design. In practical terms, this framework demonstrates the feasibility of using a natural language interface for CAD configuration, provided it is supported by a robust knowledge base and a well-integrated workflow engine. The significant reduction in time required for generating and modifying standard configurations (as presented in Section 4.4) underscores its potential value during the rapid prototyping and early design exploration phases.

From a technical standpoint, A key contribution of this work lies in the implementation of a functional architecture that effectively integrates a local LLM, a stateful workflow engine, and a commercial CAD API. A key finding in the context of CAD automation is the demonstration of the effectiveness of LangGraph in supporting error handling and rollback mechanisms. While the current implementation is tailored to fuselage components within CATIA V5, the core principles—such as constraint validation, state management, and the dual-mode interface—suggest a potential pathway for developing similar configurators for other parametric components within CATIA or other CAD platforms that offer scripting APIs.

5.7. Future Work

This work enables several promising directions for future research. Priorities include enhancing the framework's robustness, expanding its capabilities, and deepening its integration into the engineering design loop, as detailed below.

5.7.1. Strengthening Core System Architecture

The immediate priority is to address the fundamental limitations identified during development.

Advanced State Synchronization: A key challenge lies in maintaining synchronization between the framework's internal state (SQLite + LangGraph) and the active CATIA session. Future efforts will focus on creating a robust protocol to ensure consistent state alignment. This may include 1) State Snapshotting and Hashing: Regularly generate hashes of key parameters within the CATIA assembly tree and compare them against stored database values to identify any external modifications. 2)"Reconcile" Command: Introduce a dedicated NLP or GUI-based command (e.g., " synchronize with CAD") that inspects the current assembly in CATIA and updates the internal framework state, accordingly, preserving a single source of truth.

5.7.2 Expansion of Functional Capabilities

Expanding the framework's scope will enhance its utility and showcase the versatility of the underlying architecture, this can be achieved by:

Extended Component Library: The system will be expanded to support a broader range of airframe components, with the upcoming development cycle focusing on the integration of frames, bulkheads, and door surrounds. Each of these components will necessitate the creation of a corresponding parametric PowerCopy template, along with the definition of relevant parameters and constraints within the SQLite knowledge base. This extension will significantly enhance the framework's applicability across more complex structural configurations.

Multi-Parameter & Multi-Component Commands: The NLP schema and parsing logic will be extended to support compound commands. This enhancement will involve updating the output JSON schema to accommodate a list of actions, allowing the system to interpret and execute multi-part instructions within a single user input. Examples include commands like "set all ribs to thickness 20mm and height 250mm" or "create 5 ribs and 10 stringers with fixed spacing" significantly improving the system's expressiveness and usability.

5.7.3. Towards a Performance-Driven Design Agent

FEA Validation Loop Integration: The framework will be extended to integrate with Finite Element Analysis (FEA) solvers (e.g., ANSYS) via their APIs. The workflow will be enhanced to 1) Automated Analysis Setup**:** Upon a successful configuration, automatically export the CAD geometry, trigger meshing, apply standard loads and boundary conditions (e.g., pressurization, gravitational acceleration), and execute a structural analysis. 2) Results Parsing and Injection**:** Extract key performance results (e.g., max von Mises stress, total mass, safety factors, nodal displacements) and inject them as new, quarriable parameters into the LangGraph state and SQLite database. 3) Performance-Driven Querying: Allow users to interrogate the analysis results directly via NLP or the GUI (e.g., "what is the total mass?", "show me the areas with the most stress concentrations."," list the components with a safety factor below 1.5").

Automated Optimization and Redesign: Building on the existing FEA integration, the framework will be extended with optimization capabilities. Leveraging the LangGraph state to evaluate design performance, the system will be able to autonomously propose—or even execute—parameter adjustments to satisfy user-defined constraints (e.g., "minimize mass while keeping stress below 250 MPa"). transforming the tool from a configurator into a true design agent.

6. Conclusion

This thesis set out to explore the viability of using a Large Language Model (LLM)-enabled agentic framework to automate CAD model configuration. The resulting system—an AI-powered fuselage configurator—successfully demonstrates not only that such automation is feasible but that it can be executed with high efficiency and precision. By combining a local LLM (Ollama/Llama3) for natural language understanding, a stateful orchestration layer using LangGraph, an SQLite database for structured engineering knowledge, and CATIA V5's COM API for geometric execution, the project demonstrates a functional configurator for AI-integrated CAD workflows.

The central research question is answered positively: An LLM-driven agentic framework, particularly one leveraging a stateful orchestrator like LangGraph proved to be an effective solution for automating CAD configurations**.** The framework successfully bridges the gap between natural language user intents and precise constraint-compliant CAD operations. Its effectiveness stems from a layered approach, where the LLM performs intent classification, and deterministic rule-based logic ensures output validity—combining the flexibility of AI with the reliability of traditional engineering methods

Validation of the system highlights its practical robustness. The configurator achieved a 93.5% accuracy in interpreting natural language commands, managed 100% of known error cases with contextual recovery guidance, and introduced negligible latency (~0.4s**)** in the orchestration layer—most delays being attributable to the external CAD interface and local LLM inference. A key insight from the evaluation is the complementary nature of the NLP and GUI inputs: while natural language enables broad, conceptual directives (e.g., "space all ribs at 150mm"), the GUI allows for fine-tuned control, together forming a hybrid interface that enhances user experience and design agility.

The implications of this project are well situated within practical and methodological domains. The practical implication of this project is a proof-of-concept that can significantly reduce the time required for a specific set of repetitive fuselage design tasks. Methodologically, it provides detailed case study of integration of a local LLM with a commercial CAD system using a stateful orchestrator and a constraint database. Ultimately, the project successfully achieved its core objective: developing an LLM-enabled configurator for automating fuselage design, offering a validated reference for similar configuration-focused integration efforts.

Despite its success, the project also highlighted inherent challenges, particularly the dependency on the stability of external CAD APIs and the complexity of state synchronization between the framework and the CAD session. These limitations, however, clearly define the scope for future work, including framework hardening, functional expansion, and the integration of performance-driven design loops through simulation.

In conclusion, this thesis demonstrates that integrating modern LLMs with established engineering practices and CAD systems is not just a conceptual possibility but a tangible, effective route toward more intuitive, efficient, and user-friendly design automation. The proposed framework effectively demonstrates an LLM-driven agentic workflow is a suitable solution for automating CAD configurations within its defined scope. Meeting its core objective the framework successfully bridges the gap between the user queries and precise CAD operations.

7. References

[1] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., ... & Amodei, D. (2020). Language Models Are Few-Shot Learners. *Advances in Neural Information Processing Systems*, 33, 1877-1901. https://arxiv.org/abs/2005.14165

[2] Dassault Systèmes (2022). *CATIA V5 Parametric Modeling and Automation Techniques*. Dassault Technical Reports.

[3] Jurafsky, D., & Martin, J. H. (2023). *Speech and Language Processing (3rd ed.)*. Pearson.

[4] La Rocca, G. (2012). *Knowledge-Based Aerospace Design*. Springer.

[5] McKinney, W. (2022). *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and Jupyter (3rd ed.)*. O'Reilly Media.

[6] Liu, Y., et al. (2023). "CADGPT: Generative Pre-Training for CAD Sketch Generation." *IEEE Transactions on Visualization and Computer Graphics*, 29(5), 1-12. https://doi.org/10.1109/TVCG.2023.3247108

[7] Vaswani, A., et al. (2017). *Attention Is All You Need. Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*. https://arxiv.org/abs/1706.03762

[8] Xu, K., et al. (2021). "Deep Learning for Parametric CAD Modeling." *Computer-Aided Design*, 135, 103017. https://doi.org/10.1016/j.cad.2021.103017

[9] Zhang, H., et al. (2022). "Prompting Large Language Models for Engineering Design Automation." ASME Journal of Mechanical Design, 144(8), 081402. https://doi.org/10.1115/1.4054521

[10] Boeing 737 Assembly Manual, Sec. 4.5. *"Proprietary documentation cited in this work was accessed under confidentiality agreements with Boeing and is not publicly available."*

[11] Touvron et al. (2023). *LLaMA: Open and Efficient Foundation Language Models.* https://arxiv.org/abs/2302.13971

[12] LangGraph. (2024). *LangGraph: Stateful Multi-Actor Workflow Framework for LLM Applications. GitHub Repository. https://github.com/langchain-ai/langgraph*

[13] Hornbæk, K., & Oulasvirta, A. (2023). What is interaction? *ACM Transactions on Computer-Human Interaction*.

[14] Żyłkowski, K., et al. (2019). *Performance Testing and Profiling in Python*. In Proceedings of the International Conference on Software Engineering.

[15] Leu, M. C., ElMaraghy, H. A., Nee, A. Y. C., & Ong, S. K. (2013). Chapter 5: Product Data Management. In *Handbook of Manufacturing Engineering and Technology* (pp. 1-32). Springer, London. https://doi.org/10.1007/978-1-4471-4976-7_5-1

[16] Meta AI. (2024). Llama 3 Model. https://llama.meta.com/llama3/