

How Can Large Language Models Help Humans in Design And Manufacturing? Part 2: Synthesizing an End-to- End LLM-Enabled Design and Manufacturing Workflow

**Liane Makatura¹ Michael Foshey¹ Bohan Wang¹ Felix Hähnlein²
Pingchuan Ma¹ Bolei Deng¹ Megan Tjandrasuwita¹ Andrew Spielberg³
Crystal Owens¹ Peter Yichen Chen¹ Allan Zhao¹ Amy Zhu²
Wil Norton¹ Edward Gu¹ Joshua Jacob¹ Yifei Li¹ Adriana Schulz²
Wojciech Matusik¹**

¹Massachusetts Institute of Technology, Cambridge, Massachusetts, United States of America,

²University of Washington, Seattle, Washington, United States of America,

³Harvard University, Cambridge, Massachusetts, United States of America

The MIT Press

Published on: May 28, 2024

DOI: <https://doi.org/10.1162/99608f92.0705d8bd>

License: [Creative Commons Attribution 4.0 International License \(CC-BY 4.0\)](https://creativecommons.org/licenses/by/4.0/)

ABSTRACT

Generative AI models provide foundational knowledge and sufficient reasoning to aid in individual aspects of a computational design and modeling workflow (see [Part 1](#)). In this work, we analyze the ability of state-of-the-art LLMs (in particular GPT-4) to reason about the entire end-to-end workflow, from conceptualizing to realization, for two target domains: static physical objects (here, furniture), and dynamical cyberphysical systems (here, quadcopters). Our investigation serves as an exemplar of how to use LLMs to build out such workflows, as well as a means of identifying which steps are most brittle and especially warrant future study. We conclude with an analysis of the capabilities and limitations of LLMs in the context of design and manufacturing as a whole, as well as a brief discussion of the grander opportunities and ethical risks put forth by such systems.

“How Can Large Language Models Help Humans in Design And Manufacturing?” is a two-part article. Part 1, “Elements of The LLM-Enabled Computational Design and Manufacturing Pipeline,” can be read [here](#).

Keywords: large language models, GPT-4, computational design, computational fabrication, CAD, CAM, design for manufacturing

1. Introduction to Part 2

In [Part 1](#), we described how large language models (LLMs) can aid users in each step of a computational design and manufacturing (CDaM) process; now, in Part 2, we describe how each LLM-augmented step can be composed into end-to-end realizations of physical, functioning devices. Following the approach of [Part 1’s Section 10](#), this part focuses chiefly on the interplay between different aspects of the CDaM workflow, and demonstrates a strategy for uniting them. As before, we caution the reader not to expect flawless application of LLM technology without hiccups, but rather, a) to understand which aspects of the LLM-powered pipeline remain challenging for GPT-4, and b) to understand the realizability, adherence to specification, and the performance of real-world counterparts of virtual designs ideated and iterated on through natural language. As a capstone for this part, we summarize capabilities, limitations, and dualisms of GPT-4 as a whole.

In this part, we focus purely on our two running examples that served as a testbed for many of our investigations in Part 1: a static design problem (a cabinet) and a dynamic system design problem (a quadcopter).

Vision. Although not every part of a future LLM-augmented computational design workflow must incorporate the application of LLMs or other generative models, it is important to ensure that each LLM-powered piece of the pipeline can be integrated into typical workflows. Are the inputs and outputs of each step well-defined, and

if so, through what data format should they be broadcast? Is an LLM such as GPT-4 able to gracefully reason about longer design and manufacturing processes? Does the synthesis of these steps introduce any new challenges that do not emerge when dealing with a subset of the processes; for example, are optimized designs also designed for manufacturing? These represent a subset of the questions and potential challenges in building complex end-to-end systems. While here we focus on showing end-to-end demonstrations for two examples of modest design difficulty, we envision a future where complex objects, devices, and cyberphysical machines could be ideated, specified, and created through AI-powered CDaM workflows, across scales.

Not all steps must necessarily be powered by LLMs, nor by AI models at all; nor must a single model such as a generalist foundation model be used throughout the process. However, users must be able to reason persistently through the stages of the workflow and iterate as necessary. Knowledge and design automation provided by LLMs should minimize the amount of iteration needed, though some iteration—especially concerning formalizing objectives, requirements, and fixed specifications—is inevitable in any industrial workflow. Mistakes are made, humans change their minds, missions are altered, and LLM-enabled pipelines will eventually need to have the same flexibility as their traditional counterparts. Given their relative youth, any limitations seen in employing LLMs in these demonstrations should be seen as temporary roadblocks and technical opportunities for improvement, not as insurmountable barriers.

Although this analysis is limited to the text-based GPT-4 model, multimodal models are increasingly growing in interest and popularity. In the future, we imagine that large models will accommodate inputs in the form of text, detailed images, hand-written sketches, video, audio, code, formal mathematics, and — depending on the instrumentation of the input device being used and permissions enabled — emotional and contextual cues from interactive gestures, human expression, spoken tone, and more. Models will be able to more cleanly infer user intent and goals, especially as user data and feedback itself scales through deployment of these systems. Models will proactively work with users at every step of the pipeline and direct users' flow through that pipeline, including reasoning about how and when to decompose hard problems hierarchically into more tractable subproblems. Such models will aid each specific user in aspects they did not consider, with advice personalized to that user's level of expertise.

2. Part 2: Related Work

Integrating subcomponents of CDaM into a holistic end-to-end LLM-enabled design pipeline requires reasoning about each aspect of the pipeline and their interplay. Such LLM-enabled pipelines are a relatively new and rapidly growing area of research across domains; we note some notable recent examples of LLM and other here.

In concurrent work, LLMs have been employed in domain-specific pipelines. In robotics, [Stella et al. \(2023\)](#) demonstrated how GPT-4 could also be used throughout the conceptual technical design and fabrication process. LLMs have also been applied to end-to-end ideation and synthesis in chemistry ([Boiko et al., 2023](#)).

Beyond the application of LLMs, other generative methods have been used for end-to-end design in other domains, such as materials synthesis ([Hsu et al., 2022](#); [Merchant et al., 2023](#)) and soft robotics ([Wang et al., 2023](#)).

3. End-to-End Design Examples

In the preceding Part, we explored how GPT-4 can benefit different stages of the design and manufacturing processes. Now, we consider the comprehensive end-to-end design processes for two examples that have been consistently referenced throughout this manuscript: the cabinet and the quadcopter. From initial design and evaluation to manufacturing and testing, we showcase how GPT-4 can be combined with traditional (modified) workflows at each stage to realize working physical objects. For the cabinet, GPT-4 automates 3D design, part sourcing, manufacturing instructions, assembly guidance, and performance evaluations. Similarly, for the quadcopter, GPT-4 facilitates design, part selection, manufacturing, assembly, and testing. These examples highlight the potential of GPT-4 in streamlining the product development process, while simultaneously exposing areas in which pain points persist.

3.1. Cabinet

In this subsection, we consolidate and elaborate on the comprehensive LLM-assisted development process for a wood cabinet with minimal human intervention. Throughout this process, GPT-4 played major roles in generating the 3D design, facilitating part sourcing based on the design specifications, generating machine-readable manufacturing instructions, providing human-readable assembly guidance, and conducting final performance evaluations (Figure 1). This holistic approach highlights the extensive capabilities of GPT-4 in automating and optimizing various stages of this product development process.

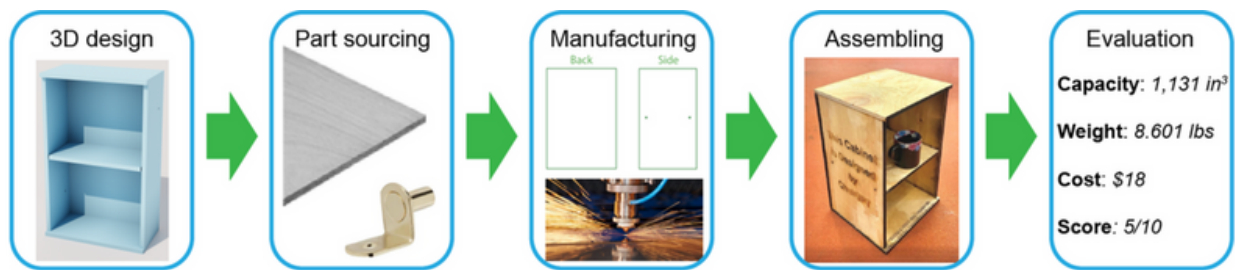


Figure 1. The cabinet design process using GPT-4. We showcase the comprehensive development process for a wood cabinet, highlighting GPT-4's roles in 3D design generation, part sourcing, manufacturing instructions, assembly guidance, and performance evaluations.

3.1.1. Design

We employed GPT-4 to generate computer-aided design (CAD) models using such as OpenJSCAD ([Part 1, Section 6.1.1](#)). After a few iterations, GPT-4 successfully generated accurate CAD design files (see [Part 1, Section 6](#) for more details). Additionally, we demonstrated utilizing GPT-4 to modify the cabinet design,

including tasks such as adding a door and a handle ([Part 1, Section 6.2.1](#)) or adjusting the number of shelves ([Part 1, Section 6.3](#)). GPT-4 even integrated prefabricated elements like brackets into the cabinet design ([Part 1, Section 6.2.2](#)). We ultimately decided to fabricate the version of the cabinet presented in Part 1, Figure 2 with one shelf supported by four brackets.

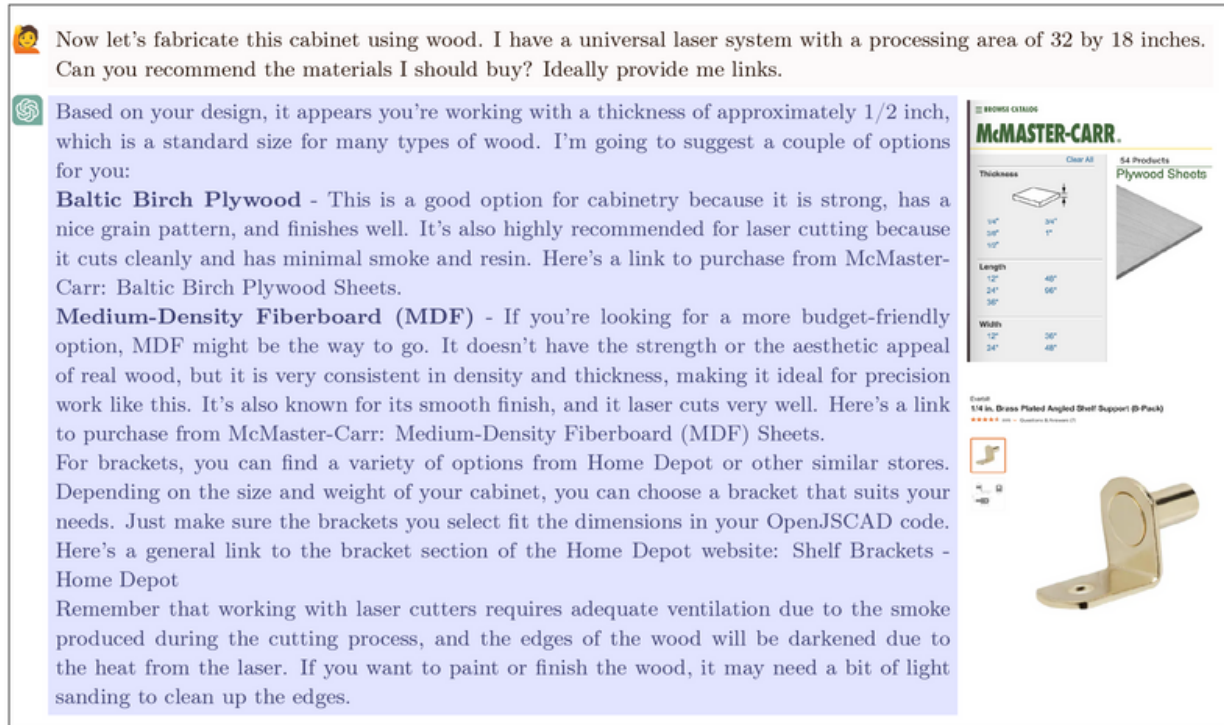


Figure 2. Part sourcing guided by GPT-4 based on the specific cabinet design and manufacturing method. In this example, we employ laser cutter and premanufactured hardware.

3.1.2. Manufacturing

We then tested GPT-4's capabilities in tackling real-world manufacturing challenges encountered during the fabrication of the cabinet. These challenges encompass various aspects of the general manufacturing process, such as part sourcing, generating machine-readable instructions, addressing material and fabrication constraints, facilitating design modifications, providing assembly guidance, and offering human-readable instructions.

For this particular example, our manufacturing plan involved utilizing a universal laser system to cut commercially available wood boards. First, by leveraging the detailed design specifications in OpenJSCAD, GPT-4 provided accurate suggestions for sourcing specific items, including brackets from Home Depot and wood pieces from McMaster. Each suggestion was accompanied by a concise description of the item ([Figure 2](#) and Part 1 [Section 8.3.1](#)). Furthermore, in order to realize the subtractive manufacturing approach, GPT-4

successfully generated precise 2D DXF files for laser cutting the wood pieces. These files were derived from the original 3D OpenJSCAD model, effectively guiding the laser cutter in executing the required cutting patterns ([Part 1, Section 8.4.2](#)).

We also showcased GPT-4's ability to adapt the design based on fabrication constraints. For example, the largest wood pieces we were able to obtain from McMaster that could fit into our laser cutter were 12 by 24 inches. Unfortunately, this size was too small to support the original cabinet design. As we presented this constraint to GPT-4, it successfully scaled down the entire design to ensure compatibility with the available wood pieces (Figure 3). Given that our laser cutter is capable of engraving patterns onto the cutting materials, we took advantage of this feature and requested GPT-4 to generate Python code for producing DXF files that could engrave words onto the side board of the cabinet. GPT-4 was able to fulfill this request, providing us with the necessary code to create the desired engraving.

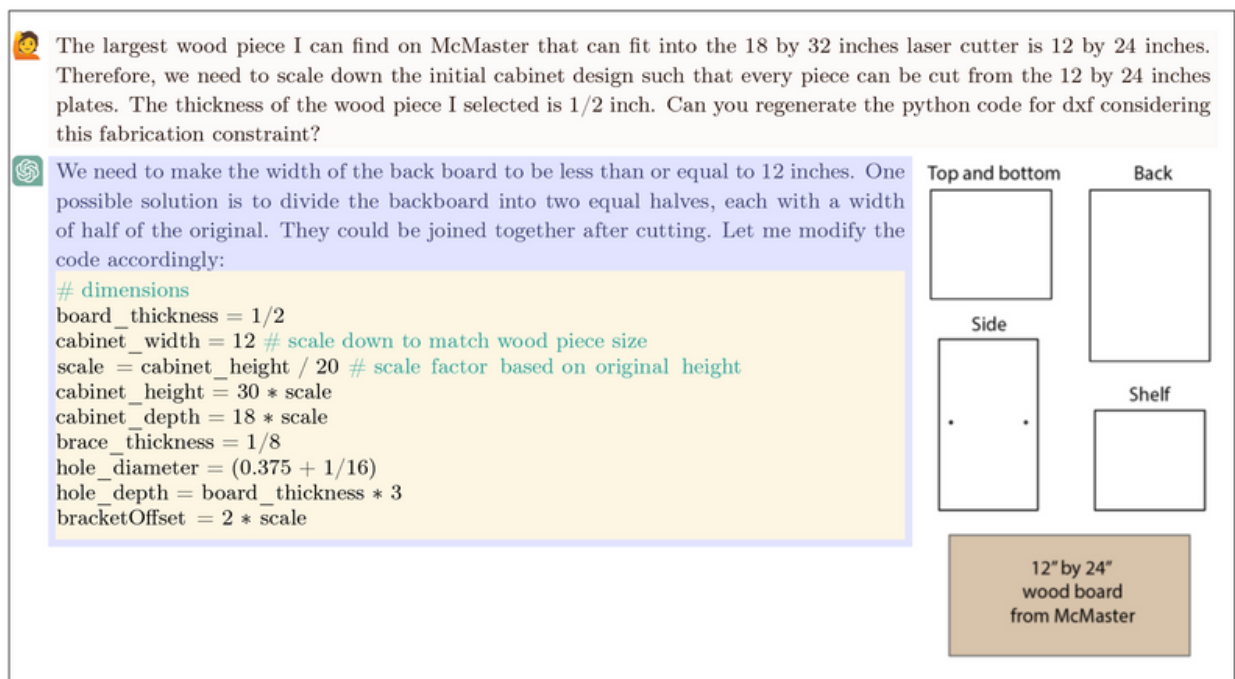


Figure 3. Handling manufacturing constraints with the assistance of GPT-4. Here, the available wood board size is insufficient for the original cabinet design. GPT-4 successfully scaled down the entire design, ensuring manufacturability.

Finally, once we had the laser-cut wood pieces, we proceeded to assemble them into a fully functional cabinet. During this process, we sought GPT-4's expertise in recommending the most suitable adhesive for wood. GPT-4 suggested Titebond Wood Glue as the optimal glue for our purposes (Figure 4). Additionally, GPT-4 provided detailed treatment guidance and step-by-step procedures for the assembly, ensuring a smooth and successful construction process (Figure 4). The final fabricated cabinet is showcased in Figure 5, offering three different view angles. To provide a clear perspective of its size, a mug is placed on the middle shelf, serving as a reference for scale.

 Great. I purchased Play Wood from McMaster with 1/2 inches. I want to use glue to assemble them together. Please provide a link for the glue I should buy and give me the details treatment procedure.

 For plywood assembly, a good choice would be Titebond III Ultimate Wood Glue. It offers superior strength, a long open-assembly time, and is safe to use. You can find it on the McMaster site [here](#). Here's a simple procedure to assemble your cabinet:

- 1. Dry Fit:** Start by fitting all the pieces together without any glue to ensure they all fit properly.
- 2. Sanding:** If necessary, sand the edges of your wood pieces for a better fit. Be careful not to alter the dimensions significantly.
- 3. Gluing Procedure:**
 - **Back Board to Side Boards:** Apply a bead of glue along the side edges of the back board. Align it with the side boards such that the back board is sandwiched between the two side boards. Ensure the edges are flush. Clamp them together and wipe off any excess glue that squeezes out with a damp cloth. Allow the glue to dry according to the manufacturer's instructions.
 - **Bottom Board:** Apply glue along the bottom edges of the side and back boards. Position the bottom board so that its edges are flush with the bottom edges of the side and back boards. Clamp and allow to dry.
 - **Top Board:** Similarly, apply glue to the top edges of the side and back boards, align the top board, clamp, and allow to dry.
 - **Shelf Board:** Apply glue along the edges of the shelf board where it will contact the side and back boards. Slide it into position and clamp. Allow to dry.
- 4. Finishing:** After the glue is completely dry, you can finish your cabinet with a finish of your choice (paint, varnish, etc) to increase its durability and improve appearance. Remember, it's important to clamp your pieces together after you've applied the glue to ensure a tight bond. Also, ensure your workspace is level so your cabinet comes out square. Please remember to follow all safety guidelines when working with tools and materials, including wearing appropriate personal protective equipment. Always work in a well-ventilated area when using glues or finishes.

Figure 4. Comprehensive assembly instructions provided by GPT-4. GPT-4 provides guidance on glue selection, pretreatment, assembly sequence, and finishing options.



Figure 5. The manufactured cabinet. Real-world cabinet manufactured from different view angles, showcasing its scale with a mug as a reference.

3.1.3. Performance Evaluation

Using the 3D OpenJSCAD design as a basis, GPT-4 was prompted to develop functions to evaluate performance criteria: storage capacity, load capacity before sagging substantially, plywood material cost, and accessibility for individuals in wheelchairs. See [Part 1, Section 9.1](#) for details on this type of design evaluation for numerical evaluation and [Part 1, Section 9.2](#) for subjective metrics. When comparing these values for the original and scaled cabinets, the storage capacity, load capacity before sagging, material cost, and accessibility were computed to be 9,205 in³, \$67, 14.336 lb, and a score of 5/10 for the original cabinet, and 1,131 in³, \$18, 8.601 lb, and a score of 5/10 for the scaled-down cabinet. It failed to compute an appropriate strength and could not verify the equation, but other evaluations are correct. Based on these numbers, both cabinets were qualitatively classified as ‘strong’ on a range of ‘strong,’ ‘moderate,’ or ‘weak,’ being far stronger than a typical cabinet used in an office. However, while the larger cabinet was categorized as ‘highly useful’ on a scale of ‘useful,’ ‘moderate,’ or ‘not useful’ due to its high strength and storage capacity, the smaller cabinet was only ‘moderately useful’ due to its small size. Further aesthetic evaluation in text form is included in Figure 6, classifying without computation the original design as ‘Contemporary’ and the scaled design as ‘Minimalist.’

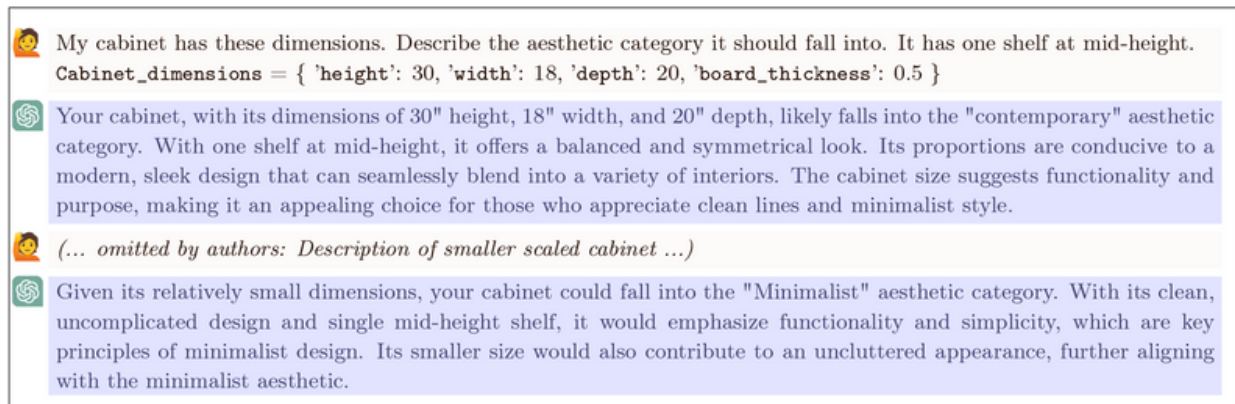


Figure 6. Aesthetic description. GPT-4 comments on the aesthetic category of the designed and fabricated cabinets based on its geometry.

3.2. Quadcopter



Figure 7. The quadcopter design process using GPT-4. The process involves sourcing parts, creating a 3D design, manufacturing, assembling, and testing the quadcopter. We successfully manufactured a working quadcopter.

In this subsection, our objective is to design a functional indoor quadcopter capable of flight, hovering, and landing, using GPT-4 with minimal human intervention. The process involves sourcing parts, creating a 3D design incorporating the chosen parts, and ultimately, manufacturing, assembling, and testing the quadcopter, as depicted in Figure 7.

3.2.1. Parts Sourcing

Many real-world systems, like drones, are built from preexisting components with various kinds of specifications. Therefore, our first task is to utilize GPT-4 to select appropriate parts for our specific use case. This was successfully accomplished using GPT-4. The detailed process is elaborated in [Part 1, Section 8.3](#), and the selected parts are shown in Figure 8 (left).

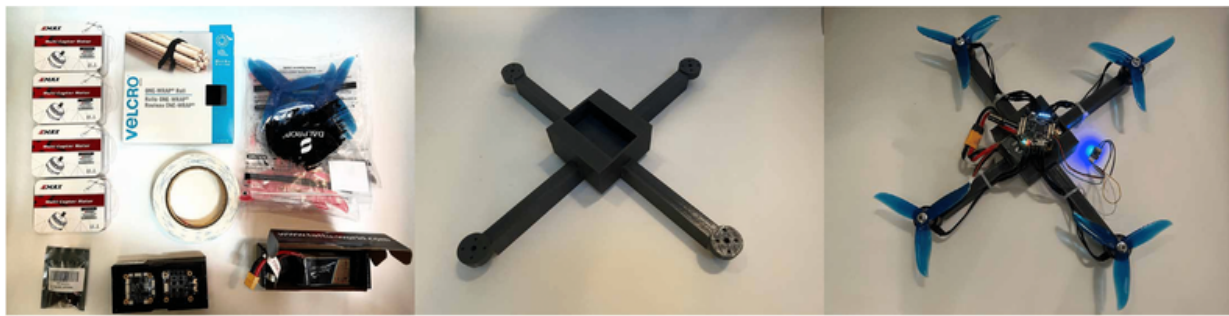


Figure 8. The parts and the printed frame of the copter. Left: Selected parts. Middle: Printed frame. Right: Assembled copter.

3.2.2. Generating Designs From Natural Language Specifications

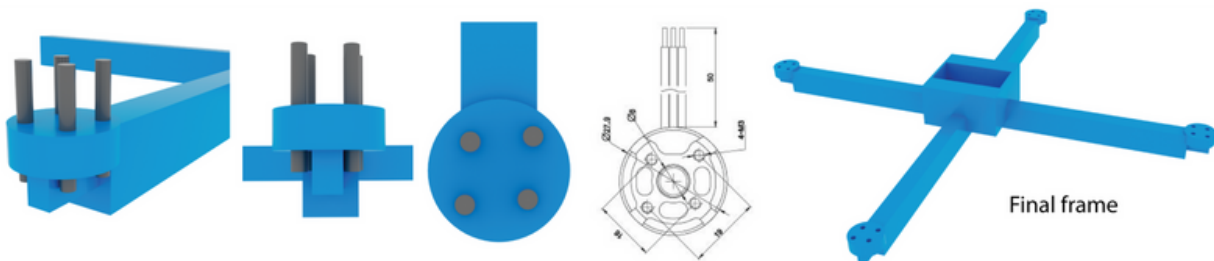


Figure 9. The quadcopter frame. Here we show that GPT-4 can create cylinders for the motor mounting holes. Using Boolean operations, we successfully created a valid frame.

Upon identifying the components, we employ GPT-4 once more to generate a viable quadcopter design incorporating those parts. With minimal human intervention, we successfully crafted a geometric design for the quadcopter, as detailed in [Part 1, Section 6.2.2](#). Now, we shift our focus to practical issues: 1) how parts are mounted onto the designed frame, and 2) whether the frame is manufacturable. Given that describing each part's geometric details is challenging and that GPT-4 does not fully comprehend how to design the frame for

optimal physical balance, we provide GPT-4 with low-level instructions to guide adjustments to the current frame design rather than expecting it to independently modify the design.

First, we adjust the frame bar's cross-sectional size using GPT-4 to ensure it is 3D-printable and adequately robust. Then, we combine all boxes and cylinders that form the frame in the geometric design. To stabilize the battery placement, we semi-integrate it into the frame, and subtract it from the frame so the frame securely holds the battery. For the controller and signal receiver, which are much lighter and smaller than the battery, we simply glue them onto the battery, eliminating the need for additional accommodations.

Lastly, we mount the motors using screws for stability, requiring screw holes in the frame. To minimize human effort, we utilize GPT-4 to create holes in the frame. For each motor, we instruct GPT-4 to generate four cylinders representing the required holes, detailing the hole specifications via text according to the motor's specifications. Even though crafting mounting holes is not trivial, we successfully produced the correct cylinders after a few prompts with GPT-4. The cylinders are shown in dark gray in Figure 9 (Left). Once the hole cylinders for one motor are ready, we let GPT-4 group and duplicate them using our `place` function. GPT-4 managed to position the hole cylinders correctly but had issues with proper rotation, which we later manually corrected.

As seen, the hole cylinders overlap with the frame bar. Since we cannot change the frame bar's thickness due to the manufacturing concern, we manually adjusted the frame bar's tip thickness to prevent it from obstructing the holes. This was done manually, as it proved challenging to adequately convey the problem and solution to GPT-4.

Our experiments also revealed a limitation: adjusting designs we completed earlier proved difficult. After extensive interaction with GPT-4, referring back to previously discussed design elements becomes a challenge. Consequently, if any issues arise with earlier addressed parts, it becomes arduous to revisit them with GPT-4 and prompt modifications. Ideally, we should finalize each design without the need for future revisions, as adjustments later prove difficult. The final frame result is displayed in Figure 9 (Right).

3.2.3. Design-to-Manufacturing

The only part that needs manufacturing is the frame. Once the frame is determined, we fabricate it using a 3D printer. Because the representation of the frame results from Boolean operations of boxes and cylinders, it is simple to directly convert them to .stl format, which is widely recognized by the 3D printers. We used OpenJSCAD to do this conversion. Once we have the .stl file, we manufacture the frame using Stratasys Fortus 400 since it has a sufficiently large build volume and it realizes precise, robust, and durable parts. We instruct the printer to use the least amount of infill possible in the print settings. By choosing a lower infill percentage, the printer will create a sparse or hollow internal structure rather than a solid one. This decision helps conserve material and reduce print time without significantly compromising the strength or weight of the copter's frame, given that it is not a load-bearing component. The resulting fabricated copter frame not only meets the required

dimensions but also balances the strength and weight necessary for optimum flight performance. We visualize the printed frame in Figure 8 (middle).

3.2.4. Assembling and Real-World Verification

With the 3D printed frame ready, we proceed to the assembly stage, integrating the preprepared components. Given that assembly considerations were incorporated into our GPT-4-guided design process, the assembly of the quadcopter is straightforward. The battery is secured in the central frame slot using double-sided tape and wrappers. Similarly, the controller and receiver are placed atop the battery and secured with double-sided tape and wrappers. The four motors are attached using screws. All elements are affixed firmly and stably, resulting in a sturdy copter ready for flight, as shown in Figure 8 (right).

Once assembled, we conduct a series of tests. First, we administer an ascending test, directing the copter to lift off the ground and ascend to a specific altitude. This test gauges the combined thrust of the motors and the propellers' efficacy in converting the motors' rotary motion into lift. It also allows us to evaluate the copter's responsiveness to radio transmitter commands, the flight controller's interpretive capacity, and the copter's overall ascent stability. The motion is depicted in Figure 10 (left).

Following ascent, we undertake a hovering test. During this phase, the copter is directed to maintain its altitude and position for a set period. Hovering demands continuous, simultaneous operation of all four motors to counter gravity. This test significantly illuminates the copter's capacity to achieve and maintain stable flight, a vital characteristic of any functioning copter. The hovering motion is demonstrated in Figure 10 (middle).

Finally, we execute a descending test, instructing the copter to safely and gradually descend to the ground. This evaluates the copter's ability to control thrust reduction and the resulting downward motion, as well as the flight controller's capacity to interpret and carry out the descent command. It is also a crucial examination of the copter's landing abilities; a smooth, safe landing is essential to preserve the copter and its components. The descending motion is exhibited in Figure 10 (right).

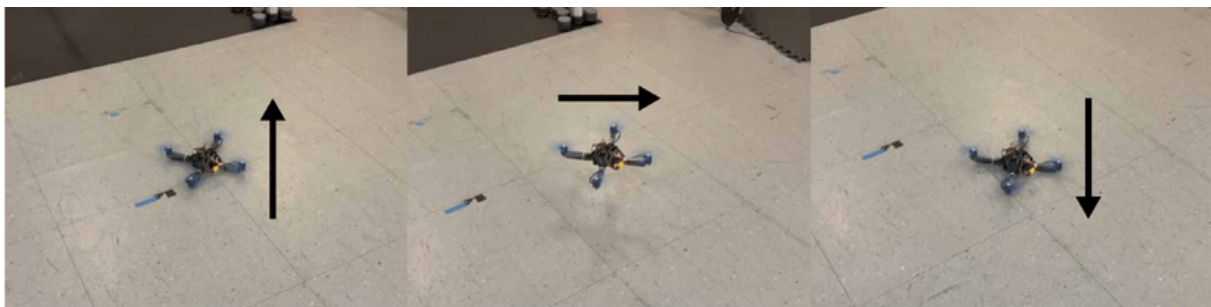


Figure 10. The flight test. Left: The ascending test. **Middle:** The hovering test. **Right:** The descending test.

3.2.5. Computer-Aided Engineering

We also investigate how GPT-4 can help with measuring the performance of a given quadcopter design. Given a current design iteration of the copter from [Part 1, Section 9.1.2](#), GPT-4 is able to identify important trade-offs to optimize and subsequently implement optimization strategies to improve performance. One such trade-off GPT-4 identified is between weight and size, where smaller copters are generally able to stay afloat longer due to reduced weight and aerodynamic drag, while larger copters have more space to accommodate larger batteries, which can provide more energy for longer flight times. Out of all the possible optimization methods to find the best combination of parameters that maximize flight time, speed, and distance, while meeting constraints on weight and size, GPT-4 chose a very suitable numerical method of particle swarm optimization (PSO) from the PySwarm library. Aside from being very efficient and simple to implement, PSO has a strong global search capability, which is beneficial when the optimal solution might be located in a large and complex space, and allows for real-time adjusting of the copter's weight and size based on performance data. GPT-4 has a strong grasp on the inherent trade-offs of such systems, and is capable of generating tailored ideas and feasible solutions to optimize performance.

We now turn to the details of using simulation to evaluate the quadcopter's performance. In the workflow of fabricating a functional robot, simulation is often used for both control and collecting performance metric statistics that can be used for optimization. Since our fabricated robot includes its own controller, we focus on using the robot's performance in simulation for design optimization. Our design space involves both a parameterized quadcopter whose frame bar lengths can vary but is otherwise constrained by the design created in [Section 3.2.2](#) and the controller design. While it is possible to ask GPT-4 to provide suggestions on the type of controller to apply, we choose to have GPT-4 generate a linear-quadratic regulator (LQR) controller, which is widely used for UAVs. We break down multicopter optimization into three steps: 1) Given the OpenJSCAD design of the quadcopter, convert the design into a format specific to modeling multibody systems, such as Universal Robot Description Format (URDF), and the means of computing relevant physical properties that inform the controller design, such as the robot's mass. 2) Given the robot's physical properties, generate an LQR controller for simulation. 3) Given a robot design in URDF, functions for extracting the design's relevant physical properties, and a controller, synthesize an algorithm to optimize the robot's design.

3.2.6. Converting OpenJSCAD to URDF

We start with the OpenJSCAD quadcopter design developed in [Section 3.2.2](#). Because there is no straightforward equivalent of subtraction and union from OpenJSCAD in URDF, we omit the creation of the holes in the motor base and replace the union of motor base parts with placement of individual links while retaining the essence of the original design. We then take an object-oriented approach to having GPT-4 synthesize the equivalent URDF code. As seen in Figure 11, we prompt GPT-4 to create a Component class whose instances store the geometry, mass, position, and orientation attributes of the corresponding OpenJSCAD primitives. Component instances also have distinct names to represent URDF links. We

additionally prompt GPT-4 to generate helper functions for placing instances with different geometries in Figure 11. This framework allows GPT-4 to generate a function that places components in terms of absolute coordinate positions and orientations and to replicate the Python equivalent of the OpenJSCAD, as shown in Figure 12.



Below are helper functions written in OpenJSCAD.

```
function createBox(w, h, d) {
  return cuboid({size:[w, h, d], center:[0, 0, 0]});
}
function createCylinder(r, h) {
  return cylinder({radius: r, height: h, center: [0, 0, 0], segments: 64});
}
```

Implement the above functions in Python. Both functions should create an instance of a Component, which is a class that keeps track of the component's name, mass, geometry, as well as absolute translation and rotation with respect to the origin. In addition, consider the below place() function.

```
// A function to add a component to the drone at specific coordinates
function place(component, x, y, z, a = 0) {
  let cc = translate([x, y, z], rotate([0, 0, a / 180.0 * Math.PI], component));
  return cc;
}
```

Implement the place function in Python, which should return a copy of the Component with the given translation and rotation. We want to give this Component a unique name, so also take in an additional suffix argument. If not None, the suffix should be added to the copy's name. Do not use any hypothetical libraries.



(... content omitted by authors ...)

```
import math
import copy

class Component:
    def __init__(self, name, mass, geometry, translation=[0,0,0], rotation=[0,0,0]):
        self.name = name
        self.mass = mass
        self.geometry = geometry
        self.translation = translation
        self.rotation = rotation

class Cuboid:
    def __init__(self, w, h, d):
        self.width = w
        self.height = h
        self.depth = d

class Cylinder:
    def __init__(self, r, h):
        self.radius = r
        self.height = h

def create_box(name, mass, w, h, d):
    return Component(name, mass, Cuboid(w, h, d))

def create_cylinder(name, mass, r, h):
    return Component(name, mass, Cylinder(r, h))

def place(component, x, y, z, a=0, suffix=None):
    cc = copy.deepcopy(component)
    cc.translation = [x, y, z]
    cc.rotation = [0, 0, a / 180.0 * math.pi]
    if suffix is not None:
        cc.name = component.name + suffix
    return cc
```

(... content omitted by authors ...)

Figure 11. Creating object-oriented representations of robot components. Using the Component class for representing a robot link facilitates the placement of links with the place() function. Later in the example, the Component class is modified to additionally keep track of an instance's parent link.

 (... omitted by authors: GPT-4 synthesized functions ...)

Using createBox, createCylinder, and place defined above, write a Python script that recreates the box and cylinder components created by the below OpenSCAD script.

(... omitted by authors: OpenJSCAD quadcopter design ...)

Store the results of place() in a dictionary mapping component name to component because we will need to access the components later. Use the “suffix” argument to ensure all components have unique names. Use the following masses of quadcopter parts and convert them to kg:

(... omitted by authors: masses of each component ...)

 (... content omitted by authors ...)

```
def create_drone():
    # Create the quadcopter parts with respective masses in kg
    motor = create_cylinder("motor", 0.030, 14, 32)
    motor_base_part1 = create_cylinder("motorBasePart1", 0.01, 14, 10)
    propeller = create_cylinder("propeller", 0.0135, 65, 8)
    controller = create_box("controller", 0.0107, 41, 38, 8.1)
    battery = create_box("battery", 0.015, 70, 35, 33)
    receiver = create_box("receiver", 0.0015, 16, 11, 5.4)
    motor_base_part2 = create_box("motorBasePart2", 0.01, 20, 7, 10)
    frame_bar1 = create_box("frameBar1", 0.01, 2 * 130 * 1.5 + 31, 15, 25)
    frame_bar2 = create_box("frameBar2", 0.01, 2 * 130 * 1.5 + 31, 15, 25)
    body = create_box("body", 0.05, 75, 75, (33 + 8.1 + 5.4) / 2 * 1.2 * 1.5)

    # Dictionaries to store components
    drone_parts = {}

    # Place the frame parts
    drone_parts["frameBar1"] = place(frame_bar1, 0, 0, 0, 0, "_place1")
    drone_parts["frameBar2"] = place(frame_bar2, 0, 0, 0, 90, "_place2")

    # Place the central body, moved down by 4
    body_height = (33 + 8.1 + 5.4) / 4 * 1.2 * 1.5 - 4
    drone_parts["body"] = place(body, 0, 0, body_height, 0, "_place1")

    # Place the quadcopter parts
    dist = 130 * 1.5 + 25 # half of frameBar length

    # Place the motor base parts
    for i in range(4):
        angle = i * 90
        x = dist * math.cos(angle * math.pi / 180)
        y = dist * math.sin(angle * math.pi / 180)
        drone_parts[f"motorBasePart1_place{i+1}"] = place(motor_base_part1, x, y, 10, angle)
        drone_parts[f"motorBasePart2_place{i+1}"] = place(motor_base_part2, x, y, 0, angle)

    (... omitted by authors: 2 additional for loops for placing motors and propellers in a similar manner as the motor bases. ...)

    # Place battery, controller, and receiver on the body, one on top of the other
    drone_parts["battery"] = place(battery, 0, 0, 33 / 2 + 2 + 13, 0)
    drone_parts["controller"] = place(controller, 0, 0, 33 + 8.1 / 2 + 2 + 13, 0)
    drone_parts["receiver"] = place(receiver, 0, 0, 33 + 8.1 + 5.4 / 2 + 2 + 13, 0)
    return drone_parts

(... content omitted by authors ...)
```

Figure 12. Replicating quadcopter component placement. Disregarding the union and subtract operators in OpenJSCAD, the above Python code accurately reproduces the locations of the components as the ones in the OpenJSCAD design. The only error is that However, one difficulty is that unlike CAD-like designs, formats for representing robots also require relations between components to not have unique names, which OpenAI GPT-4 has corrects on additional prompting. We prepare

for this when synthesizing the component placement script by prompting GPT to store the components as a dictionary, which allows easy access to the components. We tackle the challenge of generating the quadcopter's joints by relying on GPT-4's knowledge of the spatial relation between components in a quadcopter. After equipping the Component class with a function that sets the parent link, we use this interface to have GPT-4 synthesize a sequence of robot joints, as shown in Figure 13. We find that although GPT-4 understands certain substructures, such as the fact that the motor is placed on top of the motor base and the propeller is connected to the top of the motor, its initial definition results in an invalid URDF format, as both of the frame bars are root links. We thus explicitly prompt GPT-4 to choose one of the frame bars as the root link.

Finally, GPT-4 is tasked with creating a full URDF file. Because the robot is represented with modular Component instances that contain all relevant information on individual links' mass and geometry as well as relations to parent links, it is relatively straightforward for GPT-4 to create helper functions that synthesize URDF links and joints. We note that creating a joint is a more involved task, since the link's absolute position and orientation must be converted to relative position and orientation to the parent link. It is necessary to explicitly prompt GPT-4 to use the appropriate rotation matrices in its calculation; otherwise, it does not appropriately account for how the parent link's rotation affects the child link's relative translation. Because we choose LQR as the robot's controller in simulation, we ask GPT-4 to compute the full assembly's mass and moment of inertia given the Python code it has generated thus far. It outputs reasonable Python code that computes the assembly's center of mass and uses the parallel axis theorem to combine the moment of inertia of the individual links.



Figure 13. Constructing relations between quadcopter links. After creating and placing links in the appropriate positions, GPT-4 must infer parent-child relations that are not available in the OpenJSCAD design to complete the Universal Robot Description Format (URDF) representation. In its first attempt, it produces an invalid file because only one of the frame bars can be the root joint. However, GPT-4 demonstrates an understanding of how the motor, motor base, and propeller should be connected and that the main body contains the battery, controller, and receiver.

In summary, we find that explicitly prompting GPT-4 to provide suitable object-oriented representations, such as the Component class, and modularizing the code generation as much as possible, for example, asking GPT-4

to synthesize helper functions, placing components, and defining parent–child relations separately are key techniques in successfully converting from OpenJSCAD to URDF.

3.2.7. Deriving an LQR Controller for the Quadcopter

Multicopter control is an extensively studied problem, for which various algorithms have been proposed such as proportional-integral-derivative (PID) controllers, LQR controllers, and more complex alternatives. We aim to synthesize an LQR controller, as it is not only a popular choice in the literature but also guarantees optimal control when the multicopter dynamics and stable fixed point are known. In particular, we focus on controlling a quadcopter, which is an underactuated system with six degrees of freedom but only four independent actuators, resulting in nonlinear dynamics even without aerodynamical effects. However, LQR provides optimal control by linearizing the system around a stable fixed point, which then has a closed form solution.

As LQR is widely used for control, the algorithm is easily accessible through the control Python library that performs LQR with a single function call. As such, the main challenge in using LQR for quadcopter control is deriving its state space representation, the A and B matrices, used in the linear model of the system as shown in Equation 3.1. The state variable x is a 12-dof state vector, including coordinate position, linear velocity, roll, yaw, pitch, and angular position. The control variable u is the 4-dof actuator input, typically consisting of the upward thrust and the external torque applied in the xyz directions. Note that x in Equation 3.1 is the difference between the current state and a user-specified target state. Similarly, u is with respect to the target control input, where the torques are 0 and the thrust balances out the gravitational force acting on the quadcopter. GPT-4 is then asked to provide a suitable fixed point at which the quadcopter is stable, and it does provide a correct response.

$$\dot{x} = Ax + Bu \quad (3.1)$$

In order to verify GPT-4’s results, we ask GPT-4 to first synthesize the full kinematics model before using a symbolic manipulation library, SymPy, to compute the state space matrices. This forces GPT-4 to produce an interpretable dynamical model that a domain expert can verify, rather than directly outputting the A and B matrices. This proves to be essential as even though GPT-4 captures the high-level idea of assuming the quadcopter to be a rigid body and applying Newton-Euler equations to describe its acceleration and angular acceleration, it is unable to zero-shot provide the exact model of the system without user feedback. For instance, as shown in Figure 14, GPT-4 formulates the correct rotation matrix but does not apply it correctly to convert from inertial frame (control inputs) to the body frame (linear acceleration). Although GPT-4 can correct its error when given feedback, this type of error is difficult to catch without rigorously checking the correctness of GPT-4’s calculations and highlights the limitation that in some inverse design domains, a user cannot generate precise outputs from GPT-4 without the expert knowledge to perform verification. After this fix, the resulting code for deriving the A and B matrices is still incorrect as the simulated quadcopter sinks downward due to a sign error in the equations for acceleration. Similarly, generating the simulation loop in

PyBullet requires several rounds of iteration on the initial code. Some of the mistakes are more obvious, such as incorrectly indexing into the control input vector when applying external force and torques, whereas others are more specific to PyBullet’s application programming interface (API). For the former, we directly pointed out GPT-4’s mistake. In the latter case, we find that simply giving GPT-4 the error message and asking it to fix its code suffices for our problem.

 Derive the A and B matrices for the LQR controller by linearizing the dynamics of the quadcopter around the fixed point. Assume that the system has 12 states and 6 dof. The control inputs should be total thrust and x,y,z torques. You should first write the full kinematics model and then call sympy to compute the jacobians. Account for the yaw angle in the equations for linear velocity and angular velocity.

 (... omitted by authors: full description of quadcopter dynamics and high-level approach ...)

```
import sympy as sp

# Define the state variables
px, py, pz, phi, theta, psi, vx, vy, vz, wx, wy, wz = sp.symbols('px py pz phi theta psi vx vy vz wx wy wz')

# Define the control inputs
u1, u2, u3, u4 = sp.symbols('u1 u2 u3 u4')

# Define the constants
Ix, Iy, Iz, m = sp.symbols('Ix Iy Iz m')

# Rotation Matrix
R = sp.Matrix([[sp.cos(theta)*sp.cos(psi), sp.sin(phi)*sp.sin(theta)*sp.cos(psi) - sp.cos(phi)*sp.sin(psi), sp.cos(phi)*sp.sin(theta)*sp.cos(psi) + sp.sin(phi)*sp.sin(psi)],
               [sp.cos(theta)*sp.sin(psi), sp.sin(phi)*sp.sin(theta)*sp.sin(psi) + sp.cos(phi)*sp.cos(psi), sp.cos(phi)*sp.sin(theta)*sp.sin(psi) - sp.sin(phi)*sp.cos(psi)],
               [-sp.sin(theta), sp.sin(phi)*sp.cos(theta), sp.cos(phi)*sp.cos(theta)]])

# System dynamics
f = sp.Matrix([vx, vy, vz, wx, wy, wz,
               R[2, 0]*u1/m, R[2, 1]*u1/m, R[2, 2]*u1/m - 9.81,
               ((Iy - Iz)*wy*wz + u2) / Ix, ((Iz - Ix)*wx*wz + u3) / Iy, ((Ix - Iy)*wx*wy + u4) / Iz])

state = sp.Matrix([px, py, pz, phi, theta, psi, vx, vy, vz, wx, wy, wz])
controls = sp.Matrix([u1, u2, u3, u4])

# Compute Jacobians
A = f.jacobian(state)
B = f.jacobian(controls)
```

(... content omitted by authors ...)

 Can you double check the way the rotation matrix is applied when computing the system dynamics?

 Yes, you're correct, and I apologize for the mistake in my previous explanation. The total force in the body frame is given by $[0, 0, -u1]$. To find the force in the inertial frame, we multiply by the rotation matrix. The rotational dynamics are correct as given, but the translational dynamics need to be updated to account for the orientation of the quadcopter. Let's write them correctly.

The linear acceleration (ignoring air resistance) is just the force divided by the mass, so:

```
acc_x = R[0, 2] * -u1 / m
acc_y = R[1, 2] * -u1 / m
acc_z = R[2, 2] * -u1 / m + g
```

where g is the gravitational acceleration.

Let's correct this in our sympy code:

(... omitted by authors: previous code with correction made ...)

Figure 14. Deriving quadcopter state-space matrices for LQR control. Although GPT-4 formulates the problem correctly and uses the correct kinematics equations and rotation matrix, the initial derivation of the state-space matrices has a couple of errors, one of which is that GPT-4 indexes the wrong elements of the rotation matrix.

3.2.8. Co-Design of Quadcopter's Shape and Control

Now that the quadcopter design can be outputted to URDF and the LQR controller can be synthesized from a given design, we ask GPT-4 to optimize the quadcopter design. Much of the design space has been fixed by the fact that most components have unchangeable dimensions. However, as the framebars are not predefined components but rather constructed from carbon-fiber tubes, we focus on optimizing their lengths. The control then follows directly from a given design, as we only need to compute the design's total mass and moment of inertia to apply the LQR controller derived by GPT-4. With the objective of minimizing the number of simulation steps required to reach a goal height of 1 m, we prompt GPT-4 with the full Python script that converts OpenJSCAD to URDF and performs simulation, and we find that it is able to provide an outline of the optimization, consisting of an objective function that involves creating a quadcopter with specified framebar lengths and performing the simulation loop, as shown in Figure 15. GPT-4 also provides reasonable bounds on the frame bar lengths of (100 mm, 500 mm) when prompted further. GPT-4 then provides an outer optimization script that uses the helper objective function. We also explicitly prompt GPT-4 to complete its outline by updating component creation code to generate frame bars with the correct masses and to change the placement of components dependent on the length of the frame bars, namely the motors, motor bases, and propellers.

 I have the following code to generate the geometry of a Quadcopter, create an LQR controller, and simulate it. I would like to optimize the lengths of frameBar1 and frameBar2 in order to minimize the time it takes the robot to reach its target set point 1 meter off the ground, constrained to a reasonable parameter range. How might I go about modifying this code to do that?

(... omitted by authors: script for creating and simulating quadcopter ...)

 *(... content omitted by authors ...)*

First, define a function that takes the lengths of frameBar1 and frameBar2 as parameters and returns the time it takes for the robot to reach its target set point 1 meter off the ground. This function should include the code for setting up and simulating the robot, as well as measuring the time it takes to reach the target set point.

```
from scipy.optimize import minimize

def simulation_time(length1, length2):
    # Use lengths to set up and simulate the robot.
    # Measure and return the time it takes to reach the target set point.

    # Modify the frameBar1 and frameBar2 lengths
    frame_bar1 = create_box("frameBar1", 0.2526, 2 * length1 * 1.5 + 31, 15, 25)
    frame_bar2 = create_box("frameBar2", 0.2526, 2 * length2 * 1.5 + 31, 15, 25)

    # Create drone and simulate
    drone_parts = create_drone()

    set_parent_relations(drone_parts)
    # Generate URDF file
    write_urdf_file('quadcopter_scad.urdf', drone_parts)

    # Place your simulation and control code here.
    # Ensure that your control objective is to reach 1 meter above ground.
    # time_to_reach_target = ...

    return time_to_reach_target
```

Next, use the minimize function from the scipy.optimize module to find the lengths that minimize the function:

```
# Define constraints and bounds
bounds = [(MIN_LENGTH1, MAX_LENGTH1), (MIN_LENGTH2, MAX_LENGTH2)] # Replace with actual
min and max lengths

# Initial guess
x0 = [(MIN_LENGTH1 + MAX_LENGTH1)/2, (MIN_LENGTH2 + MAX_LENGTH2)/2] # initial guess can be
the midpoint of bounds
result = minimize(simulation_time, x0, bounds=bounds, method='SLSQP') # SLSQP method supports bounds

# The optimal lengths are stored in result.x
optimal_length1, optimal_length2 = result.x

(... omitted by authors: printing the results ...)

(... content omitted by authors ...)
```

Figure 15. Code skeleton for optimizing quadcopter design. To optimize a quadcopter's design, GPT-4 identifies and proposes to wrap the code that produces the quadcopter's Universal Robot Description Format (URDF) file and that performs linear-quadratic regulator control in an objective function, which enables it to perform optimization in the outer scope. However, its initial suggestion to use SLSQP is unsuitable, as there are no gradients provided from simulation.

As seen in Figure 15, GPT-4 initially proposes SLSQP, an optimization method that requires gradients from the objective function, which is not trivial to apply as we would require gradients computed from simulation.

When prompted to provide an alternative, GPT-4 suggests differential evolution, which meets the specification of not requiring a differentiable optimization problem, but can be computationally expensive. We thus explicitly prompt GPT-4 to provide code to perform grid search over the two frame bar lengths. This terminates in a reasonable amount of time with the result of making the frame bars as short as possible, which is the expected solution as the smaller the quadcopter, the less inertia it faces when taking off and decelerating toward the goal.

The resulting quadcopter with optimized frame bar lengths is visualized in Figure 16.

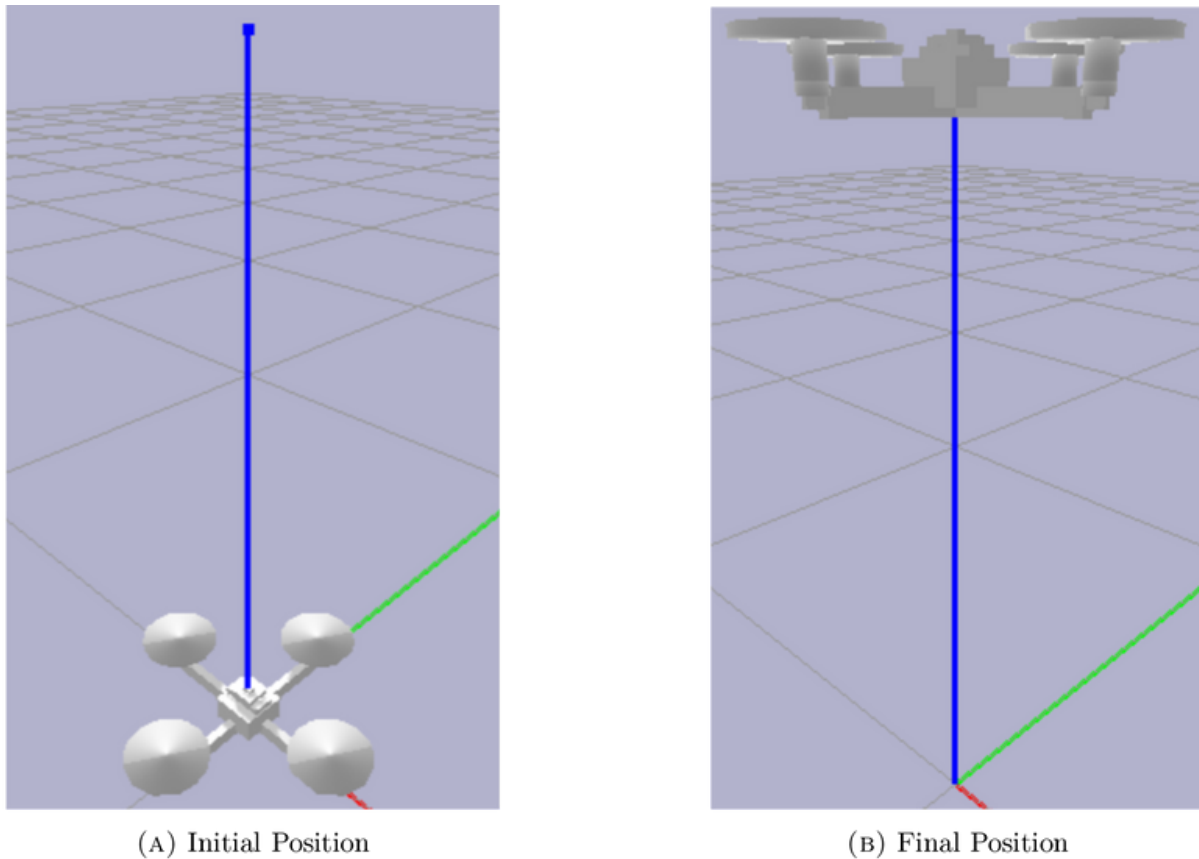


Figure 16. Simulating quadcopter performance. Left: Following the OpenJSCAD to Universal Robot Description Format (URDF) conversion script provided by GPT-4, we show the quadcopter rendered in PyBullet's simulator. The quadcopter has optimized frame bar lengths discovered with the script provided by GPT-4 and is resting at its initial position.

Right: Using the LQR controller designed by GPT-4, the quadcopter is able to lift off and hover at the goal position 1 m off the ground.

4. Part 2: Discussion

In this discussion, we elaborate on the key capabilities (C), and limitations (L) previously outlined in Part 1, as they relate to end-to-end workflows. It should be noted that this discussion necessarily adds atop the previous

discussion of each aspect of the CDaM pipeline in Part 1.

Capabilities:

C.1. Extensive Knowledge Base in Design and Manufacturing. GPT-4 clearly has broad knowledge and understanding of the stages in the CDaM pipeline, as well as their composition. It is, as in Part 1, able to correctly choose correct optimizers for each inverse design step, can identify and compute appropriate controllers when necessary, and it can choose appropriate formats for transmitting information between each step.

Modularity Support. Despite being end-to-end processes, any submodule of the CDaM pipeline can be swapped out for any other instance, assuming it produces the same type of input and output, and satisfies all specifications. However, any downstream tasks that are affected by that output may need to be modified.

Limitations:

L.1. Reasoning Challenges. As seen with the quadcopter, GPT-4 is unable to reason about intricate geometry; when dealing with precision assembly tasks such as mounting components on a thin quadcopter frame, explicit hints must be given to GPT-4. Even then, further intricate changes to geometry (such as creating mounting holes on a thin beam) may require iteration before reaching a working configuration. Further, if additional mass that was not accounted for must be added post hoc (such as is the case with dense batteries), user intervention may be required to account for this. In general, when constrained by geometry from commercial-off-the-shelf parts, GPT-4 lacks design freedom, and then can struggle to find the working solution.

Potential Solutions: Allowing GPT-4 to choose the most constrained aspects (commercial-off-the-shelf parts) first could help inform all other aspects of the design pipeline and reduce likelihood of future error and downstream iteration.

L.2. Correctness and Verification. Even though GPT-4's reasoning capabilities may be correct for simulated phenomena, they are not necessarily correct for the physical world. In particular, we saw with the cabinet that GPT-4 is able to scale numerical values reasonably, but both could not provide a certificate of why the scaled values were accurate, nor could it appropriately reason about the scaled cabinet's strength. In the quadcopter example, GPT-4 further struggles with reasoning about orientations and shifting frames of reference when computing system dynamics. These echo familiar themes from Part 1 of GPT-4 struggling with domain translation and reference frame geometry.

Potential Solutions: Explicitly providing many context cues about tasks, including changes to domain or specifications, can provide more information about which the LLM can reason.

L.3. Scalability. End-to-end workflows represent the longest processes, and as such, it is no surprise that GPT-4 struggles with memory. Adjusting completed designs at the manufacturing and testing stage remains difficult.

Potential Solutions: Carefully organizing conversations into broad but shallow trees, and re-prompting with context when necessary can help extend GPT-4's scalability.

Specific to long workflows is also the fact that at certain points, file conversion (for different processes) often becomes necessary. For the quadcopter translation from OpenJSCAD to URDF, this ultimately becomes a geometry-intensive language translation problem. In these regimes, structure is required from the user to help guide the format translation, and we must remind GPT-4 to use appropriate rotation matrices for its calculation.

5. Discussion

In this chapter, we summarize capabilities, limitations, and dualisms in applying LLMs across all aspects of the CDaM pipeline. If nothing else, we hope you, the reader, carefully consider and examine these points.

5.1. Observed Capabilities and Limitations

Our examination of GPT-4's performance in the fields of design and manufacturing has underscored several crucial capabilities instrumental in supporting the design processes, in addition to pinpointing significant limitations. Next, we summarize these observations and discuss various methods to mitigate the identified limitations.

Capabilities:

C.1. Extensive Knowledge Base in Design and Manufacturing: GPT-4 boasts a comprehensive knowledge base in the realms of design and manufacturing. Its capabilities extend to solving a wide array of problems and automatically completing specifications, making it a versatile tool across various domains.

C.2. Iteration Support: GPT-4 incorporates an iterative approach to problem-solving. When feedback is provided on errors, it attempts to rectify them. This ability to adapt, although not always successful, is a valuable facet of a GPT-4's performance.

C.3. Modularity Support: GPT-4 supports modular design, demonstrating the ability to reuse or adapt previous designs or solutions when explicitly instructed. While it does not inherently retain memory of past interactions, explicit instructions can help leverage its modular capabilities effectively.

Limitations:

L.1. Reasoning Challenges: GPT-4 encounters difficulties with certain types of reasoning, particularly those involving analytical reasoning and computations. These limitations can manifest as notable challenges in the design and manufacturing domain, for instance, a general lack of spatial reasoning capabilities.

Potential Solutions: Implementing well-crafted domain-specific languages (DSLs) can help address these challenges. DSLs, widely used in computer science, encapsulate recurring knowledge, rules, and valuable

abstractions, thereby filling knowledge gaps. Alternatively, APIs that can perform the complex computations can be integrated. GPT-4's proficiency in creating high-level abstractions can be utilized to generate inputs that be processed through APIs by computational solvers.

L.2. Correctness and Verification: GPT-4 often produces inaccurate results or justifications for its solutions and lacks the ability for self-verification. Text output by GPT-4 should be verified before being trusted, rather than the converse.

Potential Solutions: Apart from relying on human verification, automated verification can be accomplished by utilizing APIs that conduct checks and validations. By leveraging GPT-4's iterative capabilities (C.2), we can create a feedback loop that continues until a satisfactory solution is obtained.

L.3. Scalability: As tasks become larger or more complex, GPT-4's performance can deteriorate, often struggling to manage multiple tasks concurrently.

Potential Solutions: One strategy is to partition larger tasks into multiple subtasks. For instance, rather than requesting it to evaluate multiple performance metrics simultaneously, it may be more effective to request them individually. When constructing more complex models, employing an incremental design process can prove beneficial. Components can be designed and verified separately before being assembled into the final model. GPT-4's modularity support (C.3) can be used to facilitate the creation of complex models from a series of instructions.

L.4. Iterative Editing: When a design needs modifications, specifying those changes as a prompt will often lead to unsatisfactory behaviors. This is because the GPT-4, upon receiving a change prompt, will regenerate the design, often overlooking elements specified in previous prompts. This situation poses challenges to design editing.

Potential Solutions: One of our solutions involved feeding the prompts back in to create a full specification in a single prompt, though this also caused challenges due to the scalability limitations discussed above. An alternative was explicit modularization to enable parts to be designed and reused, though the latter was not possible in some cases.

Dualism:

D.1. Context Information: GPT-4's performance improves significantly with the provision of context information. The more detailed the domain description, the better it performs. Furthermore, GPT-4 is adept at providing context for its actions, making it an asset in sequential workflows. This characteristic proves particularly beneficial when using GPT-4's generated content in subsequent tasks, as these tasks can utilize the context included in the output from the initial task. However, it may also be harmful if the user does not

provide enough context, or if the user would like to create an unusual design, but GPT-4 is unable to overcome the biases it associates with a particular domain.

D.2. Unprompted Responses: GPT-4 often infers aspects that are not specified in the prompt, either auto-completing specifications or finding ways to make decisions without enough information. While this is interesting in the design context in terms of allowing for partial specifications that can be auto-completed, it can sometimes be overly proactive, guiding the design in some aspects, which may limit creativity.

6. Ethical Considerations

While powerful LLM-augmented CDaM technology has the potential to improve design, manufacturing, and livelihoods, it also raises risks. Here, we briefly outline some of these. We emphasize that this list is not exhaustive.

Job Displacement: At its best, the integration of generative AI technology into CDaM workflows has the potential to make design at the industrial and hobbyist levels accessible to everyone. This could expand the possibility of what designers and engineers can create, and accelerate their workflows, in turn enabling individuals and companies to take bigger creative risks in products and open new markets. Even in this best-case scenario, without efforts to train a workforce in how to use and/or develop AI technology, this could further inequality between skilled and unskilled workers. If, however, generative AI does not enable enhanced worker creativity (or if it is not applied for that purpose), and instead it allows companies to churn out products with less labor or less skilled labor, then generative AI not only runs a serious risk of damping industries, but also will displace workers, especially at the skilled level.

Design Data Privacy and Security and Intellectual Property (IP): Interrogative LLMs generate output in response to human query. This language and these responses encode ‘foundational knowledge’ across domains; however, where does this knowledge come from? Current LLMs are trained from public data, privately procured exclusive data, and in some cases, private user data. Who owns that data? If it is the user, and that data is used to train LLMs, then how can the user be sure that their data is not being leaked through the models to other users? This is especially important when dealing with sensitive user data, or sensitive company data, in the case of IP. If LLM-augmented CAD programs are used to invent new technologies, it must be determined if that IP belongs to the user, and if so, how LLM service providers will ensure that data (or the knowledge it encodes) is not improperly ever transmitted to unprivileged parties.

Dual-Use Nature and Potential Military Applications: Already we have seen demonstrations that LLMs can be used for creating dangerous malware, which could have immediate serious ramifications and applications. It is not hard to imagine LLM-augmented CDaM technology being used for other weaponized purposes, such as creating new military vehicles, payloads, or other equipment. While some of this technology can be used for strategic defensive and life-saving operations, it can also be used for offensive capabilities, and often-times, the classification of defensive and offensive capabilities are contextual and/or subjective. To what level such

technology should (and even can) be regulated and deployed must be a subject of policy debate at both governmental and industrial consortia levels. We require international collaboration and active and open debate on ethical use in military applications.

Design Reliability and Accountability: As we have seen, LLMs get things wrong, and in certain domains, not infrequently. LLMs can rarely be trusted blindly right now, but in the future, they may become robust enough to operate autonomously on difficult queries. In both these advanced situations and even today as AI proliferates through society and users learn more about it, we must ensure that downstream consumers are broadcast accurate measurements of task uncertainty and appropriately vet those answers. Further, when generative AI models do make mistakes (perhaps without any human-in-the-loop), we must apply existing or develop new models of accountability and, when harm is committed, appropriate compensation for damages.

Limited Access to Tools: AI-powered CDaM technology can empower everyone, but only if everyone has access to them. If running such technology requires access to advanced hardware or embargoed chipsets, then only privileged classes will have access to this technology. Rather than empower and reduce inequities, AI-empowered LLM technology can then become a vehicle for inequality, especially by amplifying existing gaps.

7. Conclusions

In conclusion, we find that GPT-4 possesses numerous capabilities that can be leveraged within the domain of design and manufacturing. This area, where creativity converges with practicality, presents exciting opportunities for advancements that can potentially bring about a significant shift in the way we ideate, prototype, and manufacture a broad range of products.

However, it is essential to recognize that substantial work remains to be done to fully support the integration of these tools within this field. A fundamental issue is that design for manufacturing involves a delicate balance between creativity and formal verification. Engineering design presents a paradox; it requires precision and exactness, yet thrives on an iterative and exploratory spirit. While our experiments have managed to circumvent limitations for formal reasoning through user guidance, DSL crafting, and APIs that call computations, there is still much to understand about how best to implement these strategies.

For instance, we hope that our analysis can stimulate new insights about DSL design. Historically, DSLs have been developed with human users in mind. However, when we shift our perspective to creating DSLs for an AI coder, new questions and possibilities emerge. What should these DSLs look like? We believe our analysis provides valuable insights into this concept, particularly within the design and manufacturing domain.

Similarly, in terms of API usage and framework development, we have observed a myriad of possibilities. Approaches range from dividing problems into parts that can be tackled by GPT-4 and others that are best solved by traditional methods, to iterative solutions, and even to a complete reframing of the problem by asking GPT-4 to generate problem-solving code. Each of these approaches carries potential advantages and

disadvantages. So, what should an optimal framework look like? We hope our research will aid others in formulating an answer to this question.

In summary, we believe our analysis offers valuable insights into how LLMs like GPT-4 can be harnessed in the domain of design and manufacturing. While we have made substantial strides, the path to fully exploiting the potential of these tools in this domain remains open, rich with opportunities for further exploration and innovation.

Acknowledgments

This material is based upon work supported in part by Defense Advanced Research Projects Agency (DARPA) Grant No. FA8750-20-C-0075, DARPA Fellowship Grant No. HR00112110007, and the National Science Foundation (NSF) under Grant No. 2141064. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the authors(s) and do not necessarily reflect the views of the National Science Foundation.

References

- Boiko, D. A., MacKnight, R., Kline, B., & Gomes, G. (2023). Autonomous chemical research with large language models. *Nature*, 624(7992), 570–578. <https://doi.org/10.1038/s41586-023-06792-0>
- Hsu, Y.-C., Yang, Z., & Buehler, M. J. (2022). Generative design, manufacturing, and molecular modeling of 3D architected materials based on natural language input. *APL Materials*, 10(4), Article 041107. <https://doi.org/10.1063/5.0082338>
- Merchant, A., Batzner, S., Schoenholz, S. S., Aykol, M., Cheon, G., & Cubuk, E. D. (2023). Scaling deep learning for materials discovery. *Nature*, 624(7990), 80–85. <https://doi.org/10.1038/s41586-023-06735-9>
- Stella, F., Della Santina, C., & Hughes, J. (2023). How can llms transform the robotic design process? *Nature Machine Intelligence*, 5(6), 561–564. <https://doi.org/10.1038/s42256-023-00669-7>
- Wang, T.-H., Zheng, J., Ma, P., Du, Y., Kim, B., Spielberg, A., Tenenbaum, J. B., Gan, C., & Rus, D. (2023). DiffuseBot: Breeding soft robots with physics-augmented generative diffusion models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, & S. Levine (Eds.), *Advances in Neural Information Processing Systems* (Vol. 36, pp. 44398–44423). Curran Associates.

©2024 Liane Makatura, Michael Foshey, Bohan Wang, Felix Hähnlein, Pingchuan Ma, Bolei Deng, Megan Tjandrasuwita, Andrew Spielberg, Crystal Owens, Peter Yichen Chen, Allan Zhao, Amy Zhu, Wil Norton, Edward Gu, Joshua Jacob, Yifei Li, Adriana Schulz, and Wojciech Matusik. This article is licensed under a

[Creative Commons Attribution \(CC BY 4.0\) International license](#), except where otherwise indicated with respect to particular material included in the article.

References

- Boiko, D. A., MacKnight, R., Kline, B., & Gomes, G. (2023). Autonomous chemical research with large language models. *Nature*, 624(7992), 570–578. <https://doi.org/10.1038/s41586-023-06792-0>
[↩](#)
- Hsu, Y.-C., Yang, Z., & Buehler, M. J. (2022). Generative design, manufacturing, and molecular modeling of 3D architected materials based on natural language input. *APL Materials*, 10(4), Article 041107. <https://doi.org/10.1063/5.0082338>
[↩](#)
- Merchant, A., Batzner, S., Schoenholz, S. S., Aykol, M., Cheon, G., & Cubuk, E. D. (2023). Scaling deep learning for materials discovery. *Nature*, 624(7990), 80–85. <https://doi.org/10.1038/s41586-023-06735-9>
[↩](#)
- Stella, F., Della Santina, C., & Hughes, J. (2023). How can llms transform the robotic design process? *Nature Machine Intelligence*, 5(6), 561–564. <https://doi.org/10.1038/s42256-023-00669-7>
[↩](#)
- Wang, T.-H., Zheng, J., Ma, P., Du, Y., Kim, B., Spielberg, A., Tenenbaum, J. B., Gan, C., & Rus, D. (2023). DiffuseBot: Breeding soft robots with physics-augmented generative diffusion models. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, & S. Levine (Eds.), *Advances in Neural Information Processing Systems* (Vol. 36, pp. 44398–44423). Curran Associates.
[↩](#)