Proceedings of the ASME 2025
International Design Engineering Technical Conferences and
Computers and Information in Engineering Conference
IDETC-CIE2025
August 17-20, 2025, Anaheim, CA

DETC2025-168856

AGENTIC LARGE LANGUAGE MODELS FOR CONCEPTUAL SYSTEMS ENGINEERING AND DESIGN

Soheyl Massoudi[1,*], Mark Fuge[1]

[1]IDEAL
Chair of Artificial Intelligence in Engineering Design
ETH Zurich
Zurich, Switzerland

ABSTRACT

This study investigates whether large language model (LLM)–powered agents can effectively manage long-context reasoning, ensure task continuity, and perform iterative refinements in early-stage engineering design. Focusing on a Solar-Powered Water Filtration System, we first engaged OpenAI's o1 model through a human-driven conversation to establish a high-quality baseline for functional decomposition, subsystem mapping, and numerical model scripts. We then developed a multi-agent framework, which leverages a graph-based approach to capture and refine the system's functions, subsystems, and numerical modeling elements. An ablation study compared this multi-agent setup to a simpler two-agent system. Both agentic frameworks are powered by the LLama 3.3 70B model.

Neither approach succeeded in fully recovering the required physics-based numerical models. However, the multi-agent system demonstrated deeper coverage by detailing subfunctions, requirements, and constraints. Despite these strengths, the LLM encountered a failure mode during extended context processing, preventing successful completion of the design workflow. These findings underscore the challenges of handling large, evolving design contexts within current LLM architectures. To address these limitations, improved inter-agent communication strategies and advanced reasoning LLMs—such as OpenAI's o1 or DeepSeek's R1—may help enhance design coherency. Further research on robust context management and structured messaging is needed to achieve an end-to-end automated engineering design capability.

Keywords: Large language models, AI in Engineering, autonomous agents, design automation, local large language models, human-AI collaboration in design

*Corresponding author: smassoudi@ehtz.ch

1. INTRODUCTION

Engineering design is an inherently complex, iterative, and multi-objective process, characterized by frequent back-and-forth between problem definition, conceptual exploration, and system integration [1]. It requires managing evolving requirements, balancing conflicting constraints, and coordinating subsystems to achieve emergent system-level behaviors [2, 3]. To navigate these challenges, designers rely on methods such as functional decomposition [4], tacit knowledge formalization [5], systems structure mining [6] and thinking [7]. Yet, despite these methodological advances, engineering design remains difficult to formalize and automate, with no universally optimal heuristics or models that generalize across domains [8].

Computational tools have helped mitigate parts of the inherent complexity in engineering design. Numerical models, for example, capture the governing physics and interactions between system components [9, 10], surrogate models reduce simulation costs for optimization [11], and multi-objective optimization frameworks effectively explore trade-offs between competing criteria [12, 13]. While these techniques enable efficient analysis of predefined design spaces, they often struggle to accommodate the dynamic, iterative, and ambiguous nature of early-stage design exploration and requirements elicitation [14].

Generative artificial intelligence (GenAI) has emerged as a promising alternative in engineering design. Approaches based on Variational Autoencoders (VAEs), Generative Adversarial Networks (GANs), and diffusion models have demonstrated potential in generating complex geometries [15, 16], synthesizing innovative topologies [17], and modeling high-dimensional design spaces [18]. These methods enhance the diversity and feasibility of generated designs by integrating performance constraints directly into the generation process. Yet, despite their success, such models remain highly specialized—they require significant domain-specific tuning, focus on narrow tasks, and lack the abil-

ity to orchestrate the entire design process from requirements elicitation to final realization [19].

Large Language Models (LLMs) have begun to expand these capabilities by facilitating natural language interaction, requirements extraction, and conceptual ideation [20–22]. Recent work already applies LLMs to tradespace exploration, requirements-engineering analysis, and diagnostics of system engineering-artifact failure modes [23–25]. However, in current practice, LLMs serve as static assistants—called upon to generate, summarize, or rephrase design artifacts—rather than as active agents capable of autonomously managing iterative design workflows, invoking simulations, and reasoning across design stages [26].

In contrast, LLM agents are beginning to transform adjacent fields through tool integration, memory, reflection [27–29], and multi-step planning [30, 31]. Agent frameworks have been successfully applied in scientific discovery [32, 33], chemistry [34], biomedical research [35], and software engineering [36], where they autonomously generate hypotheses, execute experiments, and iterate on solutions in complex, knowledge-intensive domains. These successes suggest that engineering design—an inherently iterative, tool-augmented, and knowledge-driven activity—could similarly benefit from agentic LLM architectures.

To the best of our knowledge, LLM agents specifically for conceptual engineering systems design have not yet been investigated [37, 38]. This absence is particularly notable given the alignment between the capabilities of LLM agents—such as subgoal planning [39], tool use [40], and memory-augmented reasoning [41]—and the core challenges of complex system design [42].

Our goal is to evaluate the effectiveness of an LLM-powered multi-agent framework for engineering design, specifically in managing long-context reasoning, structured task navigation, and iterative refinement. To validate our approach, we will conduct an ablation study comparing our multi-agent system to a pair of two LLM agents executing the same design tasks within a simple feedback iterative loop. This study aims to determine whether a structured, orchestrated agentic framework leads to superior performance in decomposing a high-level engineering goal into functional components, subsystems, and numerical models.

This research is guided by three key questions:

- Context length and retention: Can an LLM effectively maintain coherence across extensive design tasks that require reasoning over long contexts, or does a structured agentic approach offer advantages in information retention and recall?
- Task continuity and structured navigation: How well can an LLM transition between different engineering design phases—such as requirements extraction, functional decomposition, subsystem selection, and numerical modeling—without explicit orchestration? Does an agent-based framework improve continuity?
- Iterative refinement and structured workflows: Engineering design is inherently iterative. Does a structured agentic framework outperform the agents pair in progressively refining design elements over multiple iterations, leading to higher-quality outputs?

To investigate the role of structured multi-agent frameworks in engineering design, we conduct an ablation study comparing our agentic system with a baseline in which a duo of LLM agents, guided by similar contextual system prompts, attempts to extract functions, subfunctions, subsystems, and numerical models using an iterative while-loop structure. This comparison will evaluate whether a multi-agent approach provides advantages in reasoning, coherence, and structured decision-making over an agentic pair acting in isolation.

The study focuses on assessing the ability of each approach to correctly identify and structure key design elements, generate justified numerical models, and maintain logical consistency across design iterations. Design quality, completeness, and efficiency will be evaluated by measuring the correctness of functional decomposition, the coherence of subsystem selection, and the integration of numerical models. Additionally, we will assess whether the structured agentic framework improves information management through a graph-based design representation, potentially reducing redundant iterations and accelerating convergence to a structured design.

This investigation is limited to early-stage engineering design, focusing on requirements extraction, functional decomposition, subsystem identification, and initial numerical modeling. It does not cover later design stages such as detailed component-level optimization, multi-objective optimization, or CAD/generative AI-based shape synthesis. The findings are expected to provide empirical evidence on whether an orchestrated multi-agent workflow leads to better-structured and more efficient design processes compared to single-agent reasoning.

2. METHODOLOGY

2.1 LLM as an AI agent

LLMs are widely recognized for their conversational capabilities. However, their deployment as autonomous AI agents introduces a distinct paradigm. In this role, an LLM serves as the core reasoning engine, but its effectiveness hinges on the augmentation of four fundamental capabilities: planning, memory, tool use, and action execution. These four pillars, as presented by [43], are depicted in Fig. 1.

The integration of these capabilities is made feasible through advancements in function calling via structured JSON outputs, which enable LLMs to interact with external systems programmatically. Server-side libraries further facilitate this process by enforcing structured outputs during chat completions. These advancements align with OpenAI's introduction of Chat Markup Language (ChatML), which defines a structured sequence of messages:

- System message: Defines high-level constraints, including personality traits (e.g., `You are a helpful assistant.`), domain specialization (e.g., `You focus on Python, C++, and Java.`), and output format enforcement (e.g., `Respond using the specified JSON structure.`).
- User message: Represents the input from an external entity, typically textual but potentially including images, audio, or

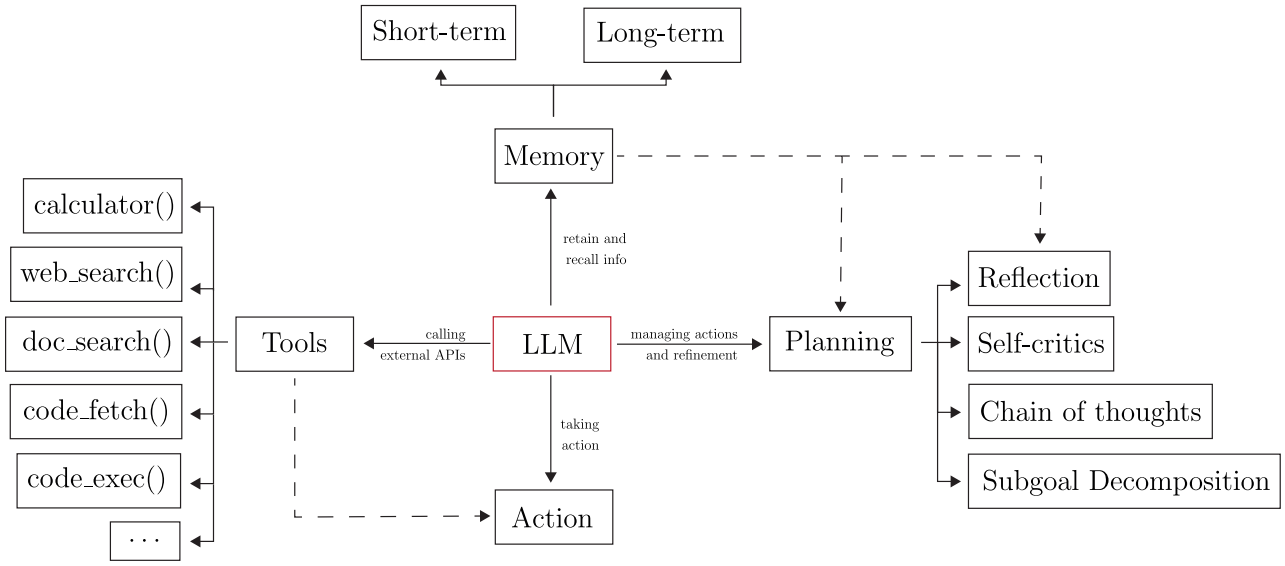

FIGURE 1: THE FOUR FOUNDATIONAL PILLARS OF AN LLM AGENT—PLANNING [43], MEMORY, TOOLS, AND ACTION. THE LLM SERVES AS THE CORE REASONING ENGINE, EXTENDED WITH CAPABILITIES FOR STRUCTURED DECISION-MAKING, EXTERNAL TOOL USAGE, MEMORY RETENTION, AND ITERATIVE REFINEMENT THROUGH PLANNING MECHANISMS SUCH AS REFLECTION AND SELF-CRITICISM.

video. In this study, we focus solely on text-based interactions.

- Assistant message: The LLM-generated response, which may include direct text replies or structured tool calls.
- Tool message: Used to transmit the results of a tool execution back to the LLM, allowing it to incorporate external data into its reasoning.

A typical ChatML payload is represented as follows:

```
[
  { "role": "system",
    "content": "You are a helpful assistant" },
  { "role": "user",
    "content": "Explain functional decomposition." }
]
```

Tool use is a critical enabler of LLM-based agents, transforming them from static information processors into dynamic problem-solvers. Structured JSON-based outputs allow LLMs to execute external API calls, retrieve real-time data from the internet, or interact with databases. This extends their functionality beyond knowledge retrieval to execution-driven problem-solving, including:

- Information retrieval: Querying search engines, Wikipedia, or ArXiv to augment responses with up-to-date knowledge.
- Code execution: Using environments such as Python REPL to validate hypotheses and generate executable algorithms.
- Retrieval-Augmented Generation (RAG): Leveraging vector databases for personalized or domain-specific knowledge retrieval, particularly useful for proprietary or confidential datasets.

By incorporating these capabilities, LLMs evolve from mere conversational models into structured cognitive agents capable of iterative problem-solving. Whenever a tool is invoked, the agent performs an action, can evaluates its outcome, and adjust its subsequent reasoning accordingly. This iterative feedback loop is foundational to intelligent, goal-directed AI systems, aligning with modern cognitive architectures for language agents [30].

2.2 Multi-Agent Architecture for Engineering Design

Engineering design requires complex, multi-step reasoning, making it challenging for a single LLM agent to effectively manage the process. Despite improvements in LLM reasoning capabilities with in-training chain-of-thought [44, 45], a single model still struggles with long-context retention, structured decomposition, and iterative refinement. Additionally, engineering design workflows demand smooth transitions between interdependent tasks—such as requirements extraction, functional decomposition, subsystem selection, and numerical modeling—where un-

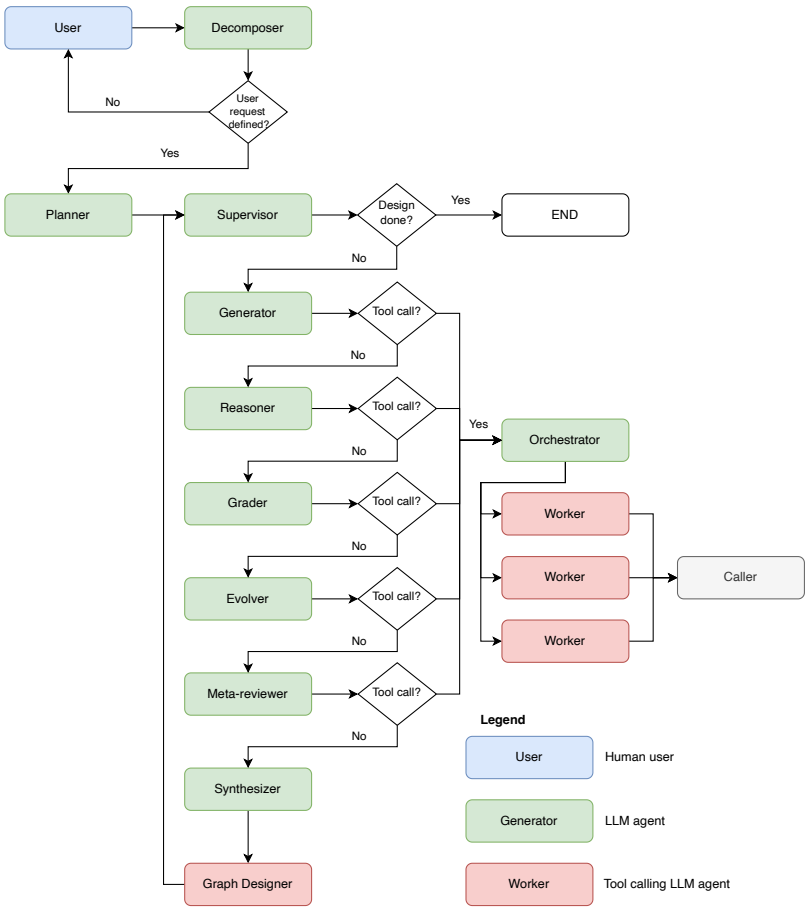

FIGURE 2: A MULTI AGENT SYSTEM (MAS) FRAMEWORK POWERED BY LLM FOR ENGINEERING DESIGN. THE DECOMPOSER ITERATES WITH THE USER TO REFINE SPECIFICATIONS, FOLLOWED BY STRUCTURED PLANNING AND SUPERVISION. SPECIALIZED AGENTS HANDLE GENERATION, EVALUATION, AND SYNTHESIS, WHILE THE ORCHESTRATOR MANAGES TOOL-BASED TASKS. THE GRAPH DESIGNER INTEGRATES FINAL UPDATES INTO THE DESIGN STATE GRAPH (DSG), ENSURING A STRUCTURED WORKFLOW.

structured execution often leads to inconsistencies or loss of information across iterations.

To overcome these limitations, we adopt a LangGraph-based multi-agent architecture patterned after Google's AI Co-Scientist "generate → critique → evolve" loop [33]. We chose this template for three reasons:

1. Built-in self-critique: Reflection agents immediately test each proposal against the *Cahier des Charges*, reducing hallucinations and premature commitment.

2. Parallel exploration: Ranking and evolution keep multiple competing hypotheses alive, giving a population-based search effect without the compute cost of a full evolutionary algorithm.

3. Domain agnosticism and scalability: Roles such as Planner, Supervisor, Generator, and Reflection are task-generic; no physics is hard-coded, so the same agent line-up can tackle other early-stage design problems (e.g., drone layout, HVAC sizing) by swapping in a new requirements file.

Implemented in LangGraph, each agent becomes a stateful node with explicit hand-off rules and shared access to the persistent Design-State Graph. This orchestration provides (i) deterministic task control, (ii) long-term memory, and (iii) traceable decomposition-capabilities a single LLM loop cannot reliably supply.

2.2.1 Agent Roles and Workflow. The framework consists of specialized LLM agents, each responsible for distinct phases of the engineering design process. They make a multi-agent system (MAS). The agents are not trained but conditioned by a detailed System message and powered by LLM only. Figure 2 illustrates the structured workflow, where agents operate iteratively and interact via a shared DSG. The main agents include:

- **Decomposer Agent (DE)** – Engages in multi-turn dialogue with the user to extract, refine, and formalize design requirements, producing a structured *cahier des charges* (see Appendix A). This is stored into the *state* variable accessible

by all agents. If the user requests `I want a water filtration system powered by solar energy`, it will start by offering a *cahier des charges* layout. It guides the user by asking questions such as: `What is the desired flow rate of the filtration system?`, `What type of impurities need to be removed from the water (e.g., bacteria, viruses, heavy metals)?`, `What is the budget for the project?`, `What materials will be used for the system (e.g., stainless steel, plastic, ceramic)?`.

- **Planner Agent (PL)** – Translates high-level design goals into a structured workflow, defining the sequence of design steps (e.g., functional decomposition, subsystem matching, numerical modeling). It reads the *cahier des charges* and decides a plan of action to perform the early conceptual design task at hand. It passes on the objectives (e.g., `Identify primary functions`) of the each design step as well as the expected outputs (e.g., `Provide a functional hierarchy diagram`) to the supervisor.

- **Supervisor Agent (SU)** – Oversees execution, ensuring agents operate within predefined constraints, validating progress at each stage before allowing transitions to subsequent steps. It passes on instructions to all agents, such as `List all main functions of the system, including but not limited to: water intake, pre-filtration, ...`. As it controls the advancement to the next design step it can return `True` or `False`. In the second case, it states the reason (e.g., `The storage subsystem was not implemented`) and asks the agents to repeat the task by giving instructions reflecting the current state of the DSG.

- **Generator Agent (GE)** – Proposes design elements at each stage, from functional decomposition to subsystem selection, iterating upon previous outputs to refine solutions. Because of this it generates multiple competing proposals (*diversity*) to satisfy the supervisor's instructions. During the functional decomposition design round, the generation agent suggests different proposals (ways) to map the subfunctions. The subfunctions may vary as well as their edge connectivity. One proposal may consider `Monitor` and `Automation` alongside `Solar Power Generation` and `Storage` while the other does not.

- **Reflection Agent (RE)** – Evaluates feasibility, completeness, and consistency of generated proposals, flagging missing elements, constraints violations, or inconsistencies. The Reflection agent gives a feedback on each proposal against the supervisor's instructions. It describes how well it aligns with the (*cahier des charges*) and the Supervisor's instructions.

- **Ranking Agent (RA)** – Assigns numerical (*scores*) to proposals based on criteria such as engineering feasibility, alignment with requirements, and structural coherence. It gives a grade out of 10, 10 being the maximum.

- **Evolution Agent (EV)** – Refines and improves top-ranked proposals, merging compatible solutions and addressing flagged weaknesses. It does not use any form of optimization algorithm but uses the current context to create a new, improved one. It could be argued that it does some form of *semantic crossover* between the original content proposal and the feedback given by the reflection agent. The Evolution agent returns a new content proposal for each proposal.

- **Meta-Reviewer Agent (MR)** – Conducts a final assessment of refined proposals, selecting the most viable design to advance to structured implementation. It creates a meta review of the design process that is used to inform the supervisor of what went well and what went wrong. It selects the proposal it deems most suited for the design step at hand (*elitist selection*).

- **Synthesizer Agent (SY)** – Translates the selected evolved proposal into structured modifications for the DSG, bridging conceptual synthesis with graph-based execution. So it defines nodes (e.g., `ID: F9, Name: Reverse Osmosis`) and edges as tuples (e.g., `(F5,F1)`) to define a directed connection from first node to second node.

- **Graph Designer Agent (GD)** – Executes modifications to the DSG, ensuring updates remain logically consistent and traceable. The GD agent is powered by Python functions to implement the output of the synthesizer and only calls the LLM to reflect on its implementation and give it as feedback as it returns to the supervisor.

In addition, an **Orchestrator Agent (OR)** manages auxiliary tasks such as external data retrieval, while **Worker Agents (WO)** perform specific tool-based operations (e.g., web searches, simulations, literature retrieval). This modular structure allows for parallelized execution and scalable adaptation to different engineering problems. The OR agent would send tasks based on an agent personalized request. Such tasks are described as:
`Conduct a web search to find the latest advancements in solar panel technology, including bifacial panels and tracking systems, to optimize energy generation.`
The WO agent then runs a web search by calling the Tavily tool to retrieve information and send it back to the specialized agent, the Caller, to enrich its self-reflexion process.

In essence, the multi-agent workflow is a form of soft mono-*objective* optimization process to obtain the best proposal under the constraints of satisfying the *cahier des charges* within each design step. However, the *semantic crossover* operation remains limited to a proposal and its reflection feedback, not between proposals. The Supervisor acts as a *constraint* satisfaction check, and a starter of the next optimization process (e.g., finding the suitable subsystems to embody the discovered functions).

2.3 Graph-Based Design Representation

At the core of our framework lies the Design State Graph (DSG), a structured representation that encodes the evolving engineering design as a directed acyclic graph. Each node

corresponds to a design element—such as a main function, subfunction, subsystem, or numerical model—while edges define hierarchical dependencies, ensuring logical flow and traceability. Nodes store essential metadata, including type (e.g., `main_function`, `subsystem`), descriptive labels for interpretability, performance parameters, and structured references to parent-child relationships. This structured approach allows for systematic decomposition, where high-level functional requirements progressively translate into subsystems and numerical models, maintaining consistency across iterations.

The graph is constructed iteratively but only materialized at the final stage by the **Synthesizer Agent**, which consolidates validated design proposals into a structured graph representation. Once synthesized, the **Graph Designer Agent** executes explicit tool calls (`add_node()`, `delete_node()`, `update_node()`) to instantiate and refine the graph. Throughout this process, the **Supervisor Agent** ensures consistency, enforcing logical dependencies and validating modifications before integration. By structuring design decisions within a persistent, graph-based representation, the framework enables modularity, traceability, and iterative optimization across the engineering workflow.

2.4 Comparison with an LLM Agent Pair Approach

To assess the necessity of this multi-agent orchestration, we compare our framework with a baseline approach using a generator-reflection agent pair LLM in a feedback loop. In this alternative setup, the two agents system (2AS) model iteratively attempts functional decomposition, subsystem selection, and numerical modeling within a while-not-successful-loop structure, using similar system prompts to guide reasoning. We adopt this approach because *self-reflection* has been shown to reduce hallucination and improve factual accuracy in long-context tasks [29].

Key differences between the two approaches include:

- **Context Retention:** The multi-agent framework maintains structured design memory through a persistent graph representation, whereas the LLM agent pair relies on last proposal and last feedback to further the design process.
- **Task Specialization:** Agentic orchestration assigns dedicated roles (e.g., decomposition, evaluation, synthesis), optimizing problem-solving efficiency, whereas the LLM pair only differentiate roles alongside generation and reflection.
- **Iterative Refinement:** The agentic workflow allows feedback loops between generation, evaluation, and refinement, ensuring progressive improvement. The LLM pair also works with a feedback mechanism, but it remains limited in comparison.

By structuring engineering design as a multi-agent workflow, we hypothesize that the proposed framework will lead to higher-quality, more coherent, and more reusable design outputs compared to a paired-agent approach.

2.5 Experimental Methodology

The MAS and 2AS experiments run on *Llama 3.3-70B*, an open-weights model distilled from Llama 3.1-405B. We chose it because it (i) retains sufficient capacity for long-context reasoning, (ii) natively supports up to 128 k tokens with stable JSON function-calling, and (iii) can be self-hosted, ensuring reproducibility. We set the temperature to zero to limit the stochasticity of the completion tokens as much as possible. Prototyping is conducted on a Mac Studio M2 Ultra with an integer 8-bit quantized model, while full-scale experiments are executed on an RTX 4090 GPU cluster with a float 16 precision model. Inference is handled using Ollama for local execution and SGLang for distributed multi-GPU processing. The later setup is used for the results presented in this paper.

To evaluate the necessity of structured multi-agent orchestration, we compare our framework against a duo of agents operating independently in a self-reasoning loop. Both approaches attempt the same engineering design tasks: functional decomposition, subsystem selection, and numerical modeling.

The multi-agent framework iteratively updates the DSG, with agents performing specialized roles under explicit orchestration. To allow for a fair comparison, the two-agents ablation also uses the *Llama 3.3 70B* model. To simulate multi-step design reasoning, the agents pair is provided with the system prompts representative of the process undertaken by the multi-agent system.

While the two agents lacks external tools, it is expected to structure its outputs in free text or JSON, capturing relationships between functions, subsystems, and numerical models.

Both approaches are assessed using the following metrics 1) correctness and completeness of functional decomposition, 2) logical consistency of subsystem selection and 3) coherence and runnability of numerical models.

To establish a ground truth for comparison, we derive baseline functional decompositions, subsystem mappings, and numerical models through a guided multi-turn conversation between a human expert and OpenAI's o1 model. These baselines, provided in the appendix (see Appendix B), serve as references for evaluating both the two-agents and the multi-agent framework. Both approaches receive the *cahier des charges* as input.

Additionally, we use Langchain's Langsmith to track and log token exchanges during both the two-agents' iterative conversation and the multi-agent framework's structured design process.

3. RESULTS

3.1 Multi-Agent Framework Design State Graph

The MAS produces a DSG that captures core system elements—functions, subsystems, and numerical models—and their dependencies (Fig. 3). We assess three main criteria in this section: 1) how accurately the MAS recognizes the required *functions*, 2) whether it aligns subsystems with the *baseline* physical architecture, and 3) how thoroughly it implements *numerical modeling*.

As summarized in Table 1, the MAS identifies 6 out of the 7 functions specified in the *cahier des charges*. It notably merges Reverse Osmosis (RO) and UV under a single `Advanced Purification` category, and omits a dedicated `Monitoring & Automation` function. The 2AS, by contrast, recognizes all 7 functions but treats advanced purification as UV only.

Referring to Table 2, the MAS lumps certain physical elements—for instance, combining the mechanical pump and mesh

TABLE 1: FUNCTIONAL DECOMPOSITION COMPARISON

Function (per Cahier des Charges)	Baseline (Human+o1)	MAS (LLama 3.3 70B)	2AS (LLama 3.3 70B)
Main Function (Purify Water)	1/1	1/1	1/1
Water Intake & Pre-Filtration	Identified (1/1)	Identified (1/1)	Identified (1/1)
Primary Filtration	Identified (1/1)	Identified (1/1)	Identified (1/1)
Advanced Purification	Separated RO and UV	Merged; partial separation	Labeled as UV only
Solar Power Generation & Storage	Includes battery modeling	Excludes battery subfunction	Partial; no battery
Water Storage & Distribution	Distinct storage and regulator	Combined node with subfunction detail	Flattened into single unit
Monitoring & Automation	Identified (1/1)	Omitted (0/1)	Identified (1/1)
Total Functions Detected	7/7	6/7	7/7

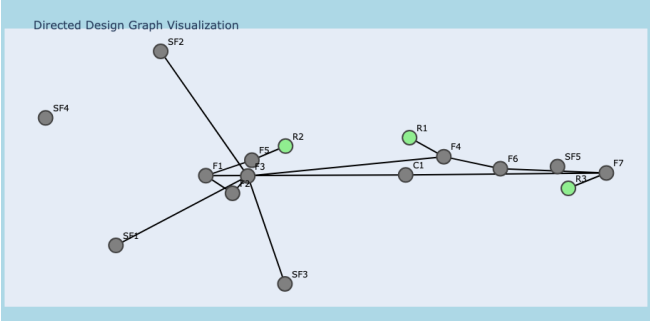

FIGURE 3: DESIGN STATE GRAPH (DSG) REPRESENTATION OF THE SOLAR-POWERED WATER FILTRATION SYSTEM. NODES REPRESENT FUNCTIONS, SUBFUNCTIONS, REQUIREMENTS, AND CONSTRAINTS, WHILE EDGES DENOTE FUNCTIONAL DEPENDENCIES.

filter into a single `Water Intake` node—and omits the battery from the solar power subsystem. This is consistent with the DSG's representation, where nodes primarily denote major functions rather than distinct physical parts. Similarly, Reverse Osmosis and UV are merged as one `Advanced Purification` node.

Compared to the physics-based baseline, Table 3 shows the MAS only partially implements core equations. For instance, the solar power node has a temperature-based efficiency correction but no battery state-of-charge logic, and advanced purification uses a non-physical combination of UV and chemical treatment. The water intake model is likewise hardcoded (e.g., 1000 L/h) with no pump head or pressure computations.

In the DSG, each function depends on other nodes (e.g., `Water Intake` feeding `Distribution`, `Advanced Purification` referencing UV/RO steps). While this arrangement matches most of the baseline's functional flow, the lack of explicit battery dynamics (for the solar subsystem) and the merged approach to advanced purification indicate that the MAS trades off detail for simplicity. Key findings include:

- All major functions are recognized except for a dedicated monitoring subsystem (though partial references to sensor constraints appear).
- Subsystem definitions are less granular—battery usage is omitted and advanced purification lumps RO and UV.
- Numerical rigor is limited to basic or empirical formulas, reducing the DSG's ability to generate physically predictive results.

Despite these limitations, the DSG does capture a coherent flow from Water Intake (F1) to Distribution (F7). It also indicates potential expansions for future design updates—e.g., adding feedback loops (Solar Power → Water Intake) to reflect energy constraints or refining filtration models (F2, F3, F4) to align with full fluid dynamics.

Lastly, the message exchanges among MAS agents can be retrieved through Langsmith (https://smith.langchain.com/public/45196006-8a13-4fc5-a2fd-13f63d726296/r), where 553 753 tokens were processed overall: 486 223 were prompt tokens, while 67 530 were completion tokens. This trace highlights how the system attempts to reconcile each design step with the *cahier des charges* and numeric modeling directives. The processed results are presented in Appendix C.

3.2 Two-Agent Framework in a Reflection Feedback Loop

In parallel, we evaluated a simpler two-agent system (2AS) that relies on a generator-reflection loop rather than a fully orchestrated multi-agent workflow. This subsection examines how the 2AS addresses functional decomposition, subsystem mapping, and numerical modeling when compared to the baseline and MAS.

From Table 1, the 2AS actually identifies all 7 functions, including `Monitoring & Automation`. However, it treats Advanced Purification as UV only, lacking explicit RO. Subfunction detail is also limited: for instance, it does not distinguish activated carbon from other filters.

As shown in Table 2, the 2AS lumps major subsystems at a coarse level—e.g., `Water Collection System` for pumping

TABLE 2: SUBSYSTEM MAPPING COMPARISON: BASELINE VS. MAS VS. 2AS

Subsystem	Baseline (Human+o1)	MAS (LLama 3.3 70B)	2AS (LLama 3.3 70B)
Pump + Mesh Filter	Explicitly defined: mechanical pump modeled using Bernoulli's principles and a mesh filter designed to capture large debris.	Integrated into the `Water Intake` node; no separate pump modeling.	Labeled as `Water Collection System` with static flow estimates, lacking explicit pump physics.
Activated Carbon Filter	Presented as a dedicated adsorption stage with pressure-drop calculations (via Ergun's equation) to assess efficiency.	Referred to indirectly (often merged with membrane filtration); lacks a standalone adsorption model.	Described generically as a `Filtration Unit` using a simplistic efficiency factor without detailed adsorption kinetics.
Reverse Osmosis (RO) + UV	Constructed as two distinct subsystems: RO (modeling osmotic pressure and permeate flux) and UV (calculating microbial inactivation via UV dose).	Merged under a single `Advanced Purification` node that combines UV and chemical dosing, omitting explicit RO modeling.	Treated as UV-only; RO aspects are not modeled.
Solar Panels + Battery	Includes a solar panel model with temperature-adjusted efficiency and a battery unit tracking state-of-charge via energy balance.	Focuses solely on the solar panel component with a simple temperature correction; battery subfunction is omitted.	Identified as `Solar Power System` with minimal panel modeling; no battery or energy storage component.
Water Tank + Flow Regulator	Divided into two parts: a water storage tank dynamically tracking inflow/outflow and a separate flow regulator based on fluid mechanics.	Combined into a single `Water Storage` node with limited detail on inflow/outflow balance.	Lumped as `Storage Tank + Distribution` with a simplified static volume assumption and no detailed flow regulation.
Sensors + IoT Module	Explicitly includes calibration routines for parameters (e.g., TDS, pH, turbidity) and fault detection.	Not represented as a distinct subsystem; only indirectly referenced in node payloads.	Mentioned solely in cost/maintenance aspects; lacks functional sensor calibration logic.
Distinct Subsystems Detected	6/6	4–5/6 (battery and sensor nodes missing)	4/6 (battery and sensor missing; advanced filtration is lumped)

and mesh filtration—and excludes battery storage within its `Solar Power System`. Similarly, sensor calibration is mentioned only as part of cost or maintenance placeholders rather than a functional subsystem.

According to Table 3, most 2AS equations are simplistic or purely empirical (e.g., static flow rates, no osmotic pressure). Despite identifying the `Monitoring & Automation` function, the 2AS does not implement any sensor calibrations or advanced automation logic—only placeholders that refer to cost. This limited physics-based rigor contrasts with the baseline's more complete hydraulic and thermodynamic equations, and even lags behind the MAS in certain areas (e.g., partial net inflow logic for water storage or partial filtering detail).

While the 2AS successfully generates a functional system architecture, it falls short in subsystem integration and numeric depth:

- One-to-one mapping from each function to a generic `unit` means less insight into interdependencies (e.g., how solar energy constrains pumping).
- No battery or sensor subsystem modeling, limiting advanced design scenarios like nighttime operation or fault monitoring.
- Static or cost-based equations for many processes, instead of first-principles (pressure, flow) or advanced environmental factors.

That said, the 2AS does provide a valid high-level design, including references to solar power, storage tanks, and distribution. Its reflection loop overcame some mistakes but could not maintain the same iterative depth as the MAS. We tracked 241 404 tokens exchanged (170 708 prompts, 70 696

TABLE 3: NUMERICAL MODELING COMPARISON: BASELINE VS. MAS VS. 2AS

Model Aspect	Baseline (Human+o1)	MAS (LLama 3.3 70B)	2AS (LLama 3.3 70B)
Pump Flow Rate	Uses Bernoulli's principle or a polynomial head curve; adjusts for fluid density and pump head.	Returns a hardcoded value (e.g., 1000 L/h) without dynamic adjustment for head or pressure.	Applies a simplistic static equation with no explicit head/pressure parameter.
Primary Filtration	Employs an Ergun-like equation to compute pressure drop coupled with flow rate for efficiency estimation.	Uses a linear or empirical efficiency model that lacks proper pressure-drop logic.	Relies on a single efficiency factor without any pressure dependency.
Advanced Purification	Separates reverse osmosis (modeling osmotic pressure and permeate flow) and UV disinfection (UV dose calculation).	Combines UV and chemical dosing into one formula; omits detailed RO modeling and osmotic pressure effects.	Implements only a UV-based model with no RO or osmotic pressure considerations.
Solar Power + Battery	Models solar power generation with temperature-adjusted efficiency and includes battery state-of-charge tracking.	Uses a simple temperature-based solar model; omits any battery modeling.	Provides a minimal solar efficiency function without battery or energy storage tracking.
Storage & Distribution	Updates tank level dynamically by balancing inflow and outflow; regulates flow based on tank conditions.	Offers a basic water quality simulation with partial net inflow logic; lacks robust flow regulation.	Computes storage level based on static volume assumptions without dynamic flow or pressure control.
Monitoring & Automation	Implements sensor calibration and fault detection routines based on established standards.	Lacks explicit sensor calibration functions; only includes descriptive references.	Contains only cost-related placeholders without any functional sensor processing.
Overall Physics Fidelity	**High**: Integrates Bernoulli, Ergun, osmotic pressure, battery, and sensor calibration models.	**Moderate–Weak**: Partial solar and filtration models; omits battery and full physics implementation.	**Weak**: Predominantly empirical or cost-driven with minimal physics-based modeling.

completions) via Langsmith (https://smith.langchain.com/public/d2b40a0d-fa0b-4066-a309-922bf6b2b81f/r), revealing numerous attempts to refine certain functions, yet ultimately failing to incorporate genuine physics-based detail.

4. DISCUSSION

Both the MAS and 2AS correctly identified most of the functions for the Solar-Powered Water Filtration system, with the exception of a dedicated monitoring function. However, because the *cahier des charges* had already outlined these functions, there was little novelty in their discovery. A greater challenge arose in translating subfunctions into distinct, mechanically embodied subsystems. For instance, while the 2AS mostly appended generic labels (e.g., `Pre-filtration units`, `Distribution system`), it did at least specify a `tank` for water storage and `pumps and piping` for distribution. By contrast, the MAS refrained from creating new subsystem nodes in the DSG, opting instead to embed subsystem information within existing nodes to facilitate numerical modeling. Although we prompted the MAS to define main functions, subfunctions, subsystems, and numerical models separately, it prioritized mapping functional dependencies and iterating on the same graph. One could argue this approach echoes a human engineer's workflow: using subfunctions primarily to identify subsystem needs, and then discarding subsystems once a numerical model takes over - at least until moving on to detailed design. From this perspective, the MAS' performance in subsystem delineation—though not strictly aligned with our instructions—was still a reasonable interpretation of how conceptual design can unfold in practice.

Despite partial successes in identifying system functions, neither the MAS nor the 2AS produced high-quality code. Analyzing token exchanges revealed that the 2AS prematurely ended its improvement loop by triggering termination tokens due to a negative logic—`The task is not yet complete, so I should not write STOP`. Consequently, the reflection agent disregarded its system prompt, defying explicit instructions. Meanwhile, the MAS encountered a hard crash (on 3 trials) when the LLM attempted to generate more than 37 000 tokens in a sin-

gle response, exceeding the 50 000-token context limit. Although no record of this oversized output exists—since tokens were not streamed—evidence of repeated phrases (e.g., `the the the...`) on early models that experimented with context length extension, such as *Vicuna 13B v1.5 16k*, strongly suggests the LLM fell into a looping failure mode. Notably, the total tokens emitted were over half those successfully processed (67 530), underscoring the severity of the breakdown.

These failures highlight that LLM agents do not consistently adhere to system prompts, especially if their underlying training biases them toward reinforcing certain responses. Structured JSON outputs mitigate some of these issues by constraining the model to strict formatting rules, but they are not universally foolproof. Additionally, *Llama 3.3 70B*'s infinite-loop crash suggests limited robustness in context handling. According to Meta's Llama team [46], the model inherits a Rotary Position Embedding (RoPE) base frequency of 500 000 from *Llama 3.1 405B*, as recommended by [47], which reduces attention decay but also risks increased hallucination, potentially triggering indefinite token repetition.

Even if the above limitations are resolved, effective context and memory management remains a significant challenge. Human engineers excel at processing, categorizing, organizing, summarizing, prioritizing, and retrieving information, whereas our current MAS approach offers only a rudimentary implementation of context retrieval (e.g., the *cahier des charges*, supervisor instructions, and the DSG). This shortfall becomes critical when guiding the engineering design process with LLM-powered agents. Transformers remain the underlying architecture for most contemporary LLMs, relying on attention mechanisms to interpret lengthy contexts. However, even reflection loops like those in the 2AS may not sufficiently "shift" the context to break out of local minima—essentially producing repetitive or shallow outputs. Improving the MAS would likely require more sophisticated processing, categorization, and retrieval of agent-generated messages, thereby providing richer semantic variance and facilitating genuinely iterative design workflows. Indeed, human-directed prompts in OpenAI's o1 have demonstrated an ability to circumvent such modes, suggesting that carefully managed context updates could yield higher-quality outcomes. While setting the LLM temperature to zero successfully enforced deterministic JSON outputs—an impressive feat—it also constrained exploration of alternative design pathways. Future investigations might integrate reasoning-centric LLMs, such as OpenAI's o1 (and its successors) or DeepSeek's R1, to foster deeper exploration. Yet this strategy entails a tradeoff: increased "thinking" tokens consume more context space, demanding ever more robust memory-management techniques.

Finally, we also acknowledge the limitations of our evaluation baseline. It is subject to the bias of the human user who prompted OpenAI's o1 model. It also overlooks the non-uniqueness of design solutions. For the same functions, different subsystems can be mapped, and different numerical scripts can be developed with non-similar physics modeling. The level of depth to the functional decomposition is also somewhat arbitrary. Our baselines did not allow to fully acknowledge the MAS for its mapping of SF1 to SF5 subfunctions such as sedimentation, membrane filtration, activated carbon filtration, UV treatment, and chemical disinfection. The process of functional decomposition itself is subjective and artificial and is not necessarily the best suited method for all types of systems (e.g., a nail clipper).

5. CONCLUSION

This study introduced a multi-agent framework using the *Llama 3.3 70B* LLM to support early-stage engineering design tasks, from initial user requests to functional decomposition, subsystem mapping, and rudimentary numerical modeling. Applied to a Solar-Powered Water Filtration system, the approach successfully generated a coherent design state graph, capturing key dependencies and outperforming a simpler two-agent generation-reflection loop in terms of structured design representation. However, the LLM's instability over extended contexts hindered the development of physically robust numerical models. Future work should focus on mitigating this failure mode through improved context and memory management, as well as more sophisticated messaging strategies among agents. Integrating advanced reasoning models (e.g., OpenAI's o1 or DeepSeek's R1, and their successors) could further enhance performance, but must account for the expanded "thinking" tokens they generate. Ultimately, refining these techniques is critical for enabling fully automated, end-to-end engineering design that converges on practical, physics-based solutions. We will also look into extending the metrics to better quantify the LLM agents engineering design process (final design quality, coverage and respect of the specifications, among others).

NOMENCLATURE

Acronyms & Technical Terms

LLM	**L**arge **L**anguage **M**odel – transformer-based model trained on vast text corpora
MAS	**M**ulti-**A**gent **S**ystem – orchestrated set of LLM agents with specialized roles
2AS	**T**wo-**A**gent **S**ystem – baseline architecture with generator & reflection agents
DSG	**D**esign **S**tate **G**raph – persistent graph of functions, subsystems, and models
Cahier des Charges	French term for a scoped requirements document setting system-level objectives
ChatML	Chat Mark-up Language – ordered list of `{role,content}` messages introduced for OpenAI models
Tool Call	Structured JSON emitted by an LLM to invoke external code, search, or simulation
RAG	**R**etrieval-**A**ugmented **G**eneration – technique to ground LLM output with documents
RoPE	**Ro**tary **P**osition **E**mbedding – positional encoding allowing long context windows

Agent Role Abbreviations

DE	Decomposer agent
PL	Planner agent

SU	Supervisor agent
GE	Generator agent
RE	Reflection agent
RA	Ranking agent
EV	Evolution agent
MR	Meta-Reviewer agent
SY	Synthesizer agent
GD	Graph-Designer agent
OR	Orchestrator agent
WO	Worker agent

REFERENCES

[1] Wynn, David C. and Eckert, Claudia M. "Perspectives on iteration in design and development." *Research in Engineering Design* Vol. 28 No. 2 (2017): pp. 153–184. DOI 10.1007/s00163-016-0226-3.

[2] Salado, Alejandro, Nilchiani, Roshanak and Verma, Dinesh. "A contribution to the scientific foundations of systems engineering: Solution spaces and requirements." *Journal of Systems Science and Systems Engineering* Vol. 26 No. 5 (2017): pp. 549–589. DOI 10.1007/s11518-016-5315-3.

[3] Meluso, John et al. "A Review and Framework for Modeling Complex Engineered System Development Processes." *IEEE Transactions on Systems, Man, and Cybernetics: Systems* Vol. 52 No. 12 (2022): pp. 7679–7691. DOI 10.1109/TSMC.2022.3163019.

[4] She, Jinjuan, Belanger, Elise and Bartels, Caroline. "Evaluating the effectiveness of functional decomposition in early-stage design: development and application of problem space exploration metrics." *Research in Engineering Design* Vol. 35 No. 3 (2024): pp. 311–327. DOI 10.1007/s00163-024-00434-w.

[5] Benfell, Adrian. "Modeling functional requirements using tacit knowledge: a design science research methodology informed approach." *Requirements Engineering* Vol. 26 No. 1 (2021): pp. 25–42. DOI 10.1007/s00766-020-00330-4.

[6] Sexton, Thurston and Fuge, Mark. "Organizing Tagged Knowledge: Similarity Measures and Semantic Fluency in Structure Mining." *Journal of Mechanical Design* Vol. 142 No. 031111 (2020). DOI 10.1115/1.4045686.

[7] Cook, Stephen C. and Pratt, Jaci M. "Advances in systems of systems engineering foundations and methodologies." *Australian Journal of Multi-Disciplinary Engineering* Vol. 17 No. 1 (2021): pp. 9–22. DOI 10.1080/14488388.2020.1809845.

[8] Watson, Michael D., Mesmer, Bryan and Farrington, Phillip. "Engineering Elegant Systems: Postulates, Principles, and Hypotheses of Systems Engineering." Adams, Stephen, Beling, Peter A., Lambert, James H., Scherer, William T. and Fleming, Cody H. (eds.). *Systems Engineering in Context*: pp. 495–513. 2019. Springer International Publishing, Cham. DOI 10.1007/978-3-030-00114-8_40.

[9] Matray, Victor et al. "A hybrid numerical methodology coupling reduced order modeling and Graph Neural Networks for non-parametric geometries: Applications to structural dynamics problems." *Computer Methods in Applied Mechanics and Engineering* Vol. 430 (2024): p. 117243. DOI 10.1016/j.cma.2024.117243.

[10] Lemu, Hirpa G. "Advances in numerical computation based mechanical system design and simulation." *Advances in Manufacturing* Vol. 3 No. 2 (2015): pp. 130–138. DOI 10.1007/s40436-015-0110-9.

[11] Alizadeh, Reza, Allen, Janet K. and Mistree, Farrokh. "Managing computational complexity using surrogate models: a critical review." *Research in Engineering Design* Vol. 31 No. 3 (2020): pp. 275–298. DOI 10.1007/s00163-020-00336-7.

[12] Sharma, Shubhkirti and Kumar, Vijay. "A Comprehensive Review on Multi-objective Optimization Techniques: Past, Present and Future." *Archives of Computational Methods in Engineering* Vol. 29 No. 7 (2022): pp. 5605–5633. DOI 10.1007/s11831-022-09778-9.

[13] Pereira, João Luiz Junho et al. "A Review of Multi-objective Optimization: Methods and Algorithms in Mechanical Engineering Problems." *Archives of Computational Methods in Engineering* Vol. 29 No. 4 (2022): pp. 2285–2308. DOI 10.1007/s11831-021-09663-x.

[14] Camburn, Bradley et al. "Design prototyping methods: state of the art in strategies, techniques, and guidelines." *Design Science* Vol. 3 (2017): p. e13. DOI 10.1017/dsj.2017.10.

[15] Chen, Wei and Fuge, Mark. "BézierGAN: Automatic Generation of Smooth Curves from Interpretable Low-Dimensional Parameters." (2021). DOI 10.48550/arXiv.1808.08871.

[16] Chen, Qiuyi et al. "Inverse Design of Two-Dimensional Airfoils Using Conditional Generative Models and Surrogate Log-Likelihoods." *Journal of Mechanical Design* Vol. 144 No. 021712 (2021). DOI 10.1115/1.4052846.

[17] Mazé, François and Ahmed, Faez. "Diffusion Models Beat GANs on Topology Optimization." *Proceedings of the AAAI Conference on Artificial Intelligence* Vol. 37 No. 8 (2023): pp. 9108–9116. DOI 10.1609/aaai.v37i8.26093.

[18] Habibi, Milad and Fuge, Mark. "Inverse Design With Conditional Cascaded Diffusion Models." 2024. American Society of Mechanical Engineers Digital Collection. DOI 10.1115/DETC2024-143607.

[19] Regenwetter, Lyle, Nobari, Amin Heyrani and Ahmed, Faez. "Deep Generative Models in Engineering Design: A Review." *Journal of Mechanical Design* Vol. 144 No. 071704 (2022). DOI 10.1115/1.4053859.

[20] Doris, Anna C. et al. "DesignQA: A Multimodal Benchmark for Evaluating Large Language Models' Understanding of Engineering Documentation." *Journal of Computing and Information Science in Engineering* (2024): pp. 1–16DOI 10.1115/1.4067333.

[21] Duan, Runlin et al. "ConceptVis: Generating and Exploring Design Concepts for Early-Stage Ideation Using Large Language Model." 2024. American Society of Mechanical Engineers Digital Collection. DOI 10.1115/DETC2024-146409.

[22] Ataei, Mohammadmehdi et al. "Elicitron: A Framework for Simulating Design Requirements Elicitation Us-

ing Large Language Model Agents." 2024. American Society of Mechanical Engineers Digital Collection. DOI 10.1115/DETC2024-143598.

[23] Apaza, Gabriel and Selva, Daniel. "Leveraging Large Language Models for Tradespace Exploration." *Journal of Spacecraft and Rockets* Vol. 61 No. 5 (2024): pp. 1165–1183. DOI 10.2514/1.A35834.

[24] Norheim, Johannes J., Rebentisch, Eric, Xiao, Dekai, Draeger, Lorenz, Kerbrat, Alain and Weck, Olivier L. de. "Challenges in applying large language models to requirements engineering tasks." *Design Science* Vol. 10 (2024): p. e16. DOI 10.1017/dsj.2024.8.

[25] Topcu, Taylan G., Husain, Mohammed, Ofsa, Max and Wach, Paul. "Trust at Your Own Peril: A Mixed Methods Exploration of the Ability of Large Language Models to Generate Expert-Like Systems Engineering Artifacts and a Characterization of Failure Modes." *Systems Engineering* Vol. n/a (2025). DOI 10.1002/sys.21810. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/sys.21810.

[26] Chiarello, Filippo et al. "Generative large language models in engineering design: opportunities and challenges." *Proceedings of the Design Society* Vol. 4 (2024): pp. 1959–1968. DOI 10.1017/pds.2024.198.

[27] Wei, Jason et al. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." (2023). DOI 10.48550/arXiv.2201.11903.

[28] Yao, Shunyu et al. "ReAct: Synergizing Reasoning and Acting in Language Models." (2023). DOI 10.48550/arXiv.2210.03629.

[29] Shinn, Noah et al. "Reflexion: Language Agents with Verbal Reinforcement Learning." (2023). DOI 10.48550/arXiv.2303.11366.

[30] Sumers, Theodore R. et al. "Cognitive Architectures for Language Agents." (2024). DOI 10.48550/arXiv.2309.02427.

[31] Xie, Chengxing and Zou, Difan. "A Human-Like Reasoning Framework for Multi-Phases Planning Task with Large Language Models." (2024). DOI 10.48550/arXiv.2405.18208.

[32] Lu, Chris et al. "The AI Scientist: Towards Fully Automated Open-Ended Scientific Discovery." (2024). DOI 10.48550/arXiv.2408.06292.

[33] Gottweis, Juraj et al. "Towards an AI co-scientist." (2025). DOI 10.48550/arXiv.2502.18864.

[34] M. Bran, Andres et al. "Augmenting large language models with chemistry tools." *Nature Machine Intelligence* Vol. 6 No. 5 (2024): pp. 525–535. DOI 10.1038/s42256-024-00832-8.

[35] Gao, Shanghua et al. "Empowering biomedical discovery with AI agents." *Cell* Vol. 187 No. 22 (2024): pp. 6125–6151. DOI 10.1016/j.cell.2024.09.022.

[36] Qian, Chen et al. "Communicative Agents for Software Development." (2023). DOI 10.48550/arXiv.2307.07924.

[37] Guo, Taicheng et al. "Large Language Model based Multi-Agents: A Survey of Progress and Challenges." (2024). DOI 10.48550/arXiv.2402.01680.

[38] Wang, Lei et al. "A survey on large language model based autonomous agents." *Frontiers of Computer Science* Vol. 18 No. 6 (2024): p. 186345. DOI 10.1007/s11704-024-40231-1.

[39] Huang, Xu et al. "Understanding the planning of LLM agents: A survey." (2024). DOI 10.48550/arXiv.2402.02716.

[40] Qin, Yujia et al. "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs." (2023). DOI 10.48550/arXiv.2307.16789.

[41] Zhang, Zeyu et al. "A Survey on the Memory Mechanism of Large Language Model based Agents." (2024). DOI 10.48550/arXiv.2404.13501.

[42] Vogel, Samuel and Rudolph, Stephan. "Complex System Design with Design Languages: Method, Applications and Design Principles." (2018). DOI 10.48550/arXiv.1805.09111.

[43] Weng, Lilian. "LLM-powered Autonomous Agents." (2023). URL https://lilianweng.github.io/posts/2023-06-23-agent/.

[44] et al., OpenAI. "OpenAI o1 System Card." (2024). DOI 10.48550/arXiv.2412.16720.

[45] et al., DeepSeek-AI. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." (2025). DOI 10.48550/arXiv.2501.12948.

[46] Grattafiori, Aaron et al. "The Llama 3 Herd of Models." (2024). DOI 10.48550/arXiv.2407.21783.

[47] Xiong, Wenhan et al. "Effective Long-Context Scaling of Foundation Models." (2023). DOI 10.48550/arXiv.2309.16039.

Cahier des Charges
Solar-Powered Water Filtration System

1. Project Overview

Title: Design of a Solar-Powered Water Filtration System

Client Objective: Develop a self-sustaining water filtration system powered by solar energy, capable of purifying water from natural sources (e.g., lakes, rivers, or rainwater).

2. Functional Requirements

- ✓ **Main Function:** Purify contaminated water into safe, potable drinking water.
- ✓ **Subfunctions:**
  - **Water Intake & Pre-Filtration:** Collect and pre-filter water from various sources.
  - **Primary Filtration:** Remove large sediments and debris.
  - **Advanced Purification:** Eliminate bacteria, viruses, and chemical contaminants.
  - **Solar Power Generation & Storage:** Power the system using solar panels and store energy.
  - **Water Storage & Distribution:** Store purified water and distribute it for usage.
  - **Monitoring & Automation:** Detect water quality, system health, and automate functions.

3. Non-Functional Requirements

✓ **Performance:**

- **Filtration Capacity:** At least 10 liters per hour.
- **Purity Level:** Must remove 99.99% of contaminants, including bacteria, heavy metals, and microplastics.
- **Solar Efficiency:** Must function with minimal sunlight (50% efficiency in low light conditions).

✓ **Sustainability & Materials:**

- **Eco-Friendly Materials:** Use biodegradable or recyclable materials.
- **Energy Efficiency:** Optimize power consumption for continuous operation with minimal storage.
- **Waste Management:** Implement a mechanism for handling and disposing of filtered waste properly.

✓ **Usability & Maintenance:**

- **User-Friendly Interface:** Easy-to-use control panel with basic automation & alerts.
- **Self-Cleaning Mechanism:** Prevent clogging and reduce manual maintenance.
- **Modularity:** Components should be replaceable without requiring expert intervention.

✓ **Safety & Compliance:**

- Must comply with WHO & EPA drinking water standards.
- Should include fail-safe mechanisms to prevent unclean water distribution.

4. Constraints & Design Considerations

✓ **Environmental Conditions:**

- Must operate in remote locations with limited access to electricity.
- Must function in temperatures ranging from -10°C to 50°C.
- Should withstand high humidity and exposure to dust & dirt.

✓ **Power & Storage:**

- Must be 100% solar-powered with at least 6-hour battery backup.
- The system should consume less than 50W for continuous operation.

✓ **Size & Portability:**

- Must be compact & lightweight for easy transport (<20 kg).
- Should be scalable for household and community use.

✓ **Cost Constraints:**

- **Target Budget:** Less than $500 for a household unit and $5000 for a community-scale system.

5. Expected Deliverables

- ✓ **System Architecture:** Definition of main components and subsystems.
- ✓ **Functional Decomposition:** Breakdown of filtration, power, and automation functions.
- ✓ **Conceptual Design:** Propose three design variants for evaluation.
- ✓ **Numerical Modeling:** Simulation of power consumption, filtration efficiency, and sustainability metrics.
- ✓ **Final Report:** A technical document summarizing findings, proposed solutions, and performance estimates.

📌 Final Note

The design process must follow a structured engineering workflow, ensuring that every step aligns with the functional objectives, technical constraints, and performance goals outlined above.

Implement this cahier des charges and write ‘FINALIZED’ at the end of it.

APPENDIX B. BASELINES FOR THE SOLAR-POWERED WATER FILTRATION SYSTEM

B.1 System Structure and Numerical Models

The engineering design process follows a structured workflow, where each function is mapped to a specific subsystem, and each subsystem is governed by numerical models. The relationships between these elements are outlined below:

TABLE 4: FUNCTIONAL DECOMPOSITION, SUBSYSTEMS, AND NUMERICAL MODELS

Function	**Subsystem**	**Numerical Model(s)**
Water Intake & Pre-Filtration	Pump + Mesh Filter	`calculate_pump_flow_rate()`
Primary Filtration	Activated Carbon Filter	`calculate_pressure_drop_and_efficiency()`
Advanced Purification	Reverse Osmosis + UV	`calculate_ro_filtration_rate()`, `calculate_concentrate_flow_rate()`, `calculate_uv_dose()`
Solar Power Generation	Solar Panels + Battery	`calculate_solar_power()`, `update_battery_state_of_charge()`
Water Storage & Distribution	Water Tank + Flow Regulator	`update_tank_level()`, `regulate_flow_rate()`
Monitoring & Automation	Sensors + IoT Module	`calibrate_tds_sensor()`, `calibrate_ph_sensor()`, `calibrate_turbidity_sensor()`, `detect_system_faults()`

B.2 Python Numerical Models

Numerical models developed as baselines via a multi-turn conversation between OpenAI's o1 model and a human expert.

B.2.1 Water Properties.

```python
def water_density_kg_m3(Tc: float) -> float:
    """
    More accurate polynomial fit for water density at 1 atm.
    Valid from ~0°C to ~100°C.

    rho(T) = 999.83952 + 16.952577*T - 7.9905127e-3*T^2
             - 4.6241757e-5*T^3  (truncated higher-order terms)
    """
    return (999.83952
            + 16.952577 * Tc
            - 7.9905127e-3 * (Tc**2)
            - 4.6241757e-5 * (Tc**3))


def water_viscosity_pa_s(Tc: float) -> float:
    """
    Approximate dynamic viscosity (Pa·s) of water based on temperature (°C).
    """
    T_k = 273.15 + Tc
    A = 2.414e-5
    B = 247.8
    C = (T_k - 140.0)
    if abs(C) < 1e-6:
        return 1.0e-3
    return A * (10 ** (B / C))
```

B.2.2 Water Intake & Pre-Filtration.

```python
def calculate_pump_flow_rate(
    input_power_watts: float,
    pump_efficiency: float,
    head_m: float,
    temperature_c: float = 20.0,
    use_pump_curve: bool = False
) -> float:
    """
    Computes the volumetric flow rate (L/h) for a pump. If use_pump_curve is True,
    it uses a simplified polynomial-based pump curve; otherwise, it uses an
    ideal hydraulic power equation.
    """
    if head_m <= 0.01:
        # Avoid singularity
        return 0.0

    rho = water_density_kg_m3(temperature_c)
    g = 9.81

    if not use_pump_curve:
        hydraulic_power_w = input_power_watts * pump_efficiency
        flow_m3_s = hydraulic_power_w / (rho * g * head_m)
        return max(0.0, flow_m3_s * 1000.0 * 3600.0)
    else:
        # Example polynomial-based approach
        a = 1.2e-5
        b = 1.05
        c = 5e-5
        effective_power_w = input_power_watts * pump_efficiency
        flow_m3_s = a * (effective_power_w**b) - c * head_m
```



```
        if flow_m3_s < 0:
            flow_m3_s = 0.0
        return flow_m3_s * 1000.0 * 3600.0
```

B.2.3 Primary Filtration.

```
def calculate_pressure_drop_and_efficiency(
    flow_rate_l_h: float,
    filter_area_m2: float,
    bed_depth_m: float,
    temperature_c: float = 20.0,
    particle_diameter_m: float = 1e-4,
    void_fraction: float = 0.4,
    ergun_const_lam: float = 150.0,
    ergun_const_inert: float = 1.75,
    contact_eff_factor: float = 100.0
) -> tuple:
    """
    Full Ergun equation for packed-bed pressure drop + simplified efficiency model.
    """
    flow_m3_s = (flow_rate_l_h / 1000.0) / 3600.0
    if flow_m3_s < 1e-12:
        return 0.0, 0.95

    mu = water_viscosity_pa_s(temperature_c)
    rho = water_density_kg_m3(temperature_c)
    v = flow_m3_s / filter_area_m2

    lam_term = ergun_const_lam * ((1 - void_fraction)**2 / (void_fraction**3)) \
               * ((mu * v) / (particle_diameter_m**2))
    in_term = ergun_const_inert * ((1 - void_fraction)/ (void_fraction**3)) \
              * ((rho * v**2) / particle_diameter_m)
    pressure_drop_pa = (lam_term + in_term) * bed_depth_m

    eff = 1.0 - math.exp(-contact_eff_factor * (bed_depth_m / v))
    if v < 1e-6:
        eff = 0.95
    return pressure_drop_pa, min(max(eff, 0.0), 1.0)
```

B.2.4 Advanced Purification (RO & UV).

```
def calculate_ro_filtration_rate(
    feed_flow_l_h: float,
    applied_pressure_pa: float,
    osmotic_pressure_pa: float,
    membrane_permeability_m_s_pa: float,
    membrane_area_m2: float,
    temperature_c: float = 25.0,
    membrane_rejection: float = 0.99
) -> float:
    """
    Reverse Osmosis flux with a simplified rejection factor.
    Clamps result to not exceed feed_flow_l_h.
    """
    alpha = 0.03
    T_ref = 25.0
    A_T = membrane_permeability_m_s_pa * math.exp(alpha * (temperature_c - T_ref))

    effective_pressure_pa = (applied_pressure_pa - osmotic_pressure_pa) \
                            * (1.0 - (1.0 - membrane_rejection))
    if effective_pressure_pa <= 0:
        return 0.0

```



```
    flux_m_s = A_T * effective_pressure_pa
    flow_m3_s = flux_m_s * membrane_area_m2
    permeate_l_h = flow_m3_s * 1000.0 * 3600.0
    return min(permeate_l_h, feed_flow_l_h)


def calculate_concentrate_flow_rate(
    feed_flow_l_h: float,
    permeate_flow_l_h: float,
    partial_recycle_factor: float = 1.0
) -> float:
    """
    Concentrate flow = (feed - permeate)* partial_recycle_factor
    """
    c_flow = (feed_flow_l_h - permeate_flow_l_h) * partial_recycle_factor
    return max(c_flow, 0.0)


def calculate_uv_dose(
    flow_rate_l_h: float,
    uv_power_w: float,
    uv_exposure_time_s: float,
    turbidity_ntu: float = 0.0,
    temperature_c: float = 25.0
) -> float:
    """
    Estimates UV dose (mJ/cm^2) with turbidity-based transmission
    and lamp temperature derating.
    """
    flow_ml_s = (flow_rate_l_h * 1000.0)/3600.0
    if flow_ml_s < 1e-12:
        return 0.0

    lamp_opt_temp = 25.0
    lamp_temp_coeff = -0.005
    diff_temp = temperature_c - lamp_opt_temp
    power_factor = max(0.0, 1.0 + lamp_temp_coeff*diff_temp)

    transmission_eff = max(0.0, 1.0 - min(1.0, turbidity_ntu/100.0))
    power_mw = uv_power_w * 1000.0 * power_factor
    dose_mj_cm2 = (power_mw * uv_exposure_time_s) / flow_ml_s
    return dose_mj_cm2 * transmission_eff
```

B.2.5 Solar Power Generation.

```
def calculate_solar_power(
    irradiance_w_m2: float,
    panel_area_m2: float,
    nominal_panel_eff: float = 0.18,
    panel_temp_c: float = 35.0,
    nominal_temp_c: float = 25.0,
    power_temp_coeff: float = -0.004
) -> float:
    """
    Basic solar panel power with temperature derating.
    """
    temp_diff = panel_temp_c - nominal_temp_c
    eff_factor = max(0.0, 1.0 + power_temp_coeff*temp_diff)
    eff = nominal_panel_eff*eff_factor
    return irradiance_w_m2*panel_area_m2*eff


def update_battery_state_of_charge(
```



```
    current_soc: float,
    battery_capacity_wh: float,
    solar_power_w: float,
    load_power_w: float,
    dt_h: float,
    round_trip_eff: float = 0.9
) -> float:
    """
    net_power = solar_power_w - load_power_w
    If net_power>0 => charging with round_trip_eff,
    else discharging.
    """
    old_energy_wh = current_soc*battery_capacity_wh
    net_power = solar_power_w - load_power_w

    if net_power>0:
        delta_energy = net_power*dt_h*round_trip_eff
    else:
        delta_energy = net_power*dt_h

    new_energy_wh = old_energy_wh + delta_energy
    new_energy_wh = max(0.0, min(new_energy_wh, battery_capacity_wh))
    return new_energy_wh/battery_capacity_wh
```

B.2.6 Water Storage & Distribution.

```
def update_tank_level(
    current_volume_l: float,
    inflow_l_h: float,
    outflow_l_h: float,
    dt_h: float
) -> float:
    """
    new_volume = current_volume + (inflow - outflow)*dt
    clamp >= 0
    """
    delta_l = (inflow_l_h - outflow_l_h)*dt_h
    new_volume = current_volume_l + delta_l
    return max(0.0, new_volume)


def regulate_flow_rate(
    tank_volume_l: float,
    desired_outflow_l_h: float,
    max_outflow_l_h: float,
    dt_h: float
) -> float:
    """
    Caps outflow at min(desired, max_outflow, tank_volume_l/dt_h)
    to avoid depleting the tank in dt_h hours.
    """
    if tank_volume_l <= 0:
        return 0.0
    feasible_outflow = tank_volume_l/dt_h
    return min(desired_outflow_l_h, max_outflow_l_h, feasible_outflow)
```

B.2.7 Monitoring & Automation.

```
def calibrate_tds_sensor(
    raw_conductivity: float,
    offset: float = 0.0,
    scale: float = 1.0,
    temperature_c: float = 25.0
```



```
) -> float:
    """
    Applies conduction-based TDS calibration with ~2%/°C correction.
    """
    temp_diff = temperature_c - 25.0
    temp_factor = 1.0 - 0.02*temp_diff
    if temp_factor<0.0:
        temp_factor=0.0
    adjusted_raw = raw_conductivity*temp_factor
    return max(0.0, (adjusted_raw-offset)*scale)


def calibrate_ph_sensor(
    raw_voltage: float,
    reference_voltage_at_ph7: float = 2.5,
    slope_mV_per_pH: float = 59.16,
    temperature_c: float = 25.0
) -> float:
    """
    Nernst-based pH calibration with temperature compensation of slope.
    pH = 7 + (measured_mV - neutral_mV)/slope(T)
    """
    T_k = 273.15 + temperature_c
    slope_correction = T_k/298.15
    actual_slope = slope_mV_per_pH*slope_correction
    measured_mV = raw_voltage*1000.0
    neutral_mV = reference_voltage_at_ph7*1000.0
    ph = 7.0 + (measured_mV-neutral_mV)/actual_slope
    return max(0.0, min(ph,14.0))


def calibrate_turbidity_sensor(
    raw_value: float,
    offset: float = 0.0,
    scale: float = 1.0
) -> float:
    """
    turbidity (NTU) = (raw_value - offset)*scale
    """
    ntu = (raw_value - offset)*scale
    return max(0.0, ntu)


def detect_system_faults(
    flow_rate_l_h: float,
    battery_soc: float,
    turbidity_ntu: float,
    tds_ppm: float = None,
    ph_value: float = None,
    min_flow_warning_l_h: float = 10.0,
    min_flow_critical_l_h: float = 5.0,
    min_battery_soc: float = 0.2,
    max_turbidity_ntu: float = 5.0,
    max_tds_ppm: float = 500.0,
    ph_range: Tuple[float, float] = (6.5, 8.5)
) -> Dict[str, str]:
    """
    Returns severity for flow ('OK','Warning','Critical'),
    plus OK/Fault for battery, turbidity, TDS, pH.
    """
    faults: Dict[str, str] = {}

```

```
    # Flow severity
    if flow_rate_l_h<min_flow_critical_l_h:
        faults["flow"] = "Critical"
    elif flow_rate_l_h<min_flow_warning_l_h:
        faults["flow"] = "Warning"
    else:
        faults["flow"] = "OK"

    # Battery
    faults["battery"] = "Fault" if battery_soc<min_battery_soc else "OK"

    # Turbidity
    faults["turbidity"] = "Fault" if turbidity_ntu>max_turbidity_ntu else "OK"

    # TDS
    if tds_ppm is not None:
        faults["tds"] = "Fault" if tds_ppm>max_tds_ppm else "OK"

    # pH
    if ph_value is not None:
        if not (ph_range[0]<=ph_value<=ph_range[1]):
            faults["ph"] = "Fault"
        else:
            faults["ph"] = "OK"

    return faults
```

APPENDIX C. MULTI-AGENT FRAMEWORK RESULTS

This appendix presents the design graph generated by the Multi-Agent Framework, showing how it refined and organized the functions, subfunctions, requirements, and constraints of the Solar-Powered Water Filtration System. Each node in the design graph corresponds to a discrete functionality or requirement (e.g., Water Intake, Pre-Filtration, UV Treatment), while edges represent dependencies or flows between these nodes.

C.1 Design State Graph Overview

The multi-agent system generated a directed graph, containing nodes for functions, subfunctions, requirements, and constraints. Table 5 summarizes each node and its corresponding payload, highlighting how the agents refined the content.

TABLE 5: NODES EXTRACTED BY THE MULTI-AGENT FRAMEWORK

Node Name	Type	Summary of Refined Payload
F1: Water Intake	Function	Responsible for collecting water from natural sources (rivers, lakes, reservoirs), with parameters like intake method, flow rate, and initial water quality. See Code C.2.1.
F2: Pre-Filtration	Function	Removes large debris and contaminants before primary filtration. Depends on water intake output. See Code C.2.2.
F3: Primary Filtration	Function	Uses membrane-based methods (MF/UF/NF) to reduce turbidity and protect advanced purification. See Code C.2.3.
F4: Advanced Purification	Function	UV disinfection, chemical disinfection, and advanced oxidation. Ensures compliance with drinking water standards. See Code C.2.4.
F5: Solar Power Generation	Function	Generates power from solar panels to supply the filtration system. Considers irradiance, panel efficiency, temperature, etc. See Code C.2.5.
F6: Water Storage	Function	Stores purified water for distribution, with parameters like storage capacity, tank material, water quality monitoring. See Code C.2.6.
F7: Distribution	Function	Delivers purified water to end-users. Involves pipe design, flow rate, and water pressure management. See Code C.2.7.
R1: Water Quality Standards	Requirement	Summarizes WHO/EPA standards, thresholds for contaminants (pH, turbidity, coliforms, metals, etc.). See Code C.4.1.
R2: Energy Efficiency Targets	Requirement	Minimizes overall power usage, sets key performance parameters (kWh/m^3 filtration). See Code C.4.2.
R3: Maintenance Access	Requirement	Ensures system design is maintainable, addressing accessibility, spare parts, and maintenance schedules. See Code C.4.3.
C1: System Boundaries	Constraint	Defines the boundaries for water intake and distribution points, external interfaces, and major subsystem interfaces. See Code C.5.
SF1–SF5	SubFunctions	Detailed expansions for sub-processes like Sedimentation, Membrane Filtration, Activated Carbon Filtration, UV Treatment, and Chemical Disinfection.

C.2 Multi-Agent Refined Payloads and Code Snippets

In this section, we present selected refined payloads and code snippets generated by the multi-agent system for each node. This structure mirrors the presentation of the Baselines (Appendix B.2), making it easier to compare the two.

C.2.1 Water Intake (F1).

```
### Multi-Agent Refined Payload for "Water Intake":

# Excerpt of the advanced design notes:
# The Water Intake function is responsible for collecting water from natural
# sources (rivers, lakes, or reservoirs), focusing on:
#   - Source Type
#   - Intake Method
#   - Flow Rate
```

```
#   - Initial Water Quality

import numpy as np

def calculate_flow_rate(intake_method, source_type, seasonal_variation):
    """
    Calculates the flow rate based on the intake method, source type,
    and seasonal variation.

    Parameters:
    - intake_method (str): e.g., 'pumping' or 'gravity-fed'
    - source_type (str): e.g., 'river', 'lake'
    - seasonal_variation (float): factor representing seasonal changes

    Returns:
    - float: flow rate in liters per hour
    """
    if intake_method == "pumping" and source_type == "river":
        base_flow_rate = 1000  # L/h
    elif intake_method == "gravity-fed" and source_type == "lake":
        base_flow_rate = 500   # L/h
    else:
        base_flow_rate = 0

    flow_rate = base_flow_rate * seasonal_variation
    return flow_rate

# Example usage
if __name__ == "__main__":
    method = "pumping"
    stype = "river"
    season_factor = 0.8
    flow = calculate_flow_rate(method, stype, season_factor)
    print(f"Calculated Flow Rate: {flow:.1f} L/h")
```

C.2.2 Pre-Filtration (F2).

```
### Multi-Agent Refined Payload for "Pre-Filtration":

import numpy as np

def pre_filtration_efficiency(debris_size, contaminant_type, flow_rate):
    """
    Calculate the efficiency of the pre-filtration process.

    Parameters:
    - debris_size (float): diameter (mm)
    - contaminant_type (str): e.g., 'sediment'
    - flow_rate (float): L/min

    Returns:
    - float: Pre-filtration efficiency as a percentage
    """
    removal_rate = 0.9  # assume 90% base removal
    efficiency = removal_rate * (1 - debris_size / 100) * (flow_rate / 100)
    return efficiency * 100

# Example usage
if __name__ == "__main__":
    eff = pre_filtration_efficiency(debris_size=5, contaminant_type="sediment", flow_rate=50)
    print(f"Pre-Filtration Efficiency: {eff:.2f}%")
```

C.2.3 Primary Filtration (F3).


```
### Multi-Agent Refined Payload for "Primary Filtration":

import numpy as np

def primary_filtration_efficiency(membrane_type, operating_pressure, filtration_rate):
    """
    Calculates the efficiency of the primary filtration process.

    Parameters:
    - membrane_type (str): MF, UF, or NF
    - operating_pressure (float): bar
    - filtration_rate (float): L/h

    Returns:
    - float: Efficiency (%)
    """
    if membrane_type == "MF":
        efficiency = (operating_pressure * 0.1) + (filtration_rate * 0.05)
    elif membrane_type == "UF":
        efficiency = (operating_pressure * 0.2) + (filtration_rate * 0.1)
    elif membrane_type == "NF":
        efficiency = (operating_pressure * 0.3) + (filtration_rate * 0.15)
    else:
        raise ValueError("membrane_type must be MF, UF, or NF")

    return efficiency * 100

# Example usage
if __name__ == "__main__":
    membrane_type = "UF"
    pressure = 5.0  # bar
    rate = 100.0    # L/h
    eff = primary_filtration_efficiency(membrane_type, pressure, rate)
    print(f"Primary Filtration Efficiency: {eff:.2f}%")
```

C.2.4 Advanced Purification (F4).

```
### Multi-Agent Refined Payload for "Advanced Purification":

import numpy as np

def advanced_purification_efficiency(uv_intensity, chemical_dose, contact_time, pH_level):
    """
    Calculates the efficiency of advanced purification (UV + chemical disinfection).

    Parameters:
    - uv_intensity (float): UV in mJ/cm^2
    - chemical_dose (float): mg/L
    - contact_time (float): minutes
    - pH_level (float)

    Returns:
    - float: Efficiency as a percentage
    """
    efficiency = (uv_intensity * 0.01) + (chemical_dose * 0.05) \
                 + (contact_time * 0.1) + (pH_level * 0.02)
    return efficiency

# Example usage
if __name__ == "__main__":
    eff = advanced_purification_efficiency(uv_intensity=100, chemical_dose=5,
                                           contact_time=30, pH_level=7)
    print(f"Advanced Purification Efficiency: {eff:.2f}%")
```


C.2.5 Solar Power Generation (F5).

```
### Multi-Agent Refined Payload for "Solar Power Generation":

"""
Responsible for harnessing solar energy via photovoltaic (PV) panels
to power the entire water filtration system, including pumping and
filtration processes.
Key parameters:
- Solar Irradiance (W/m^2)
- Panel Efficiency (%)
- Temperature (°C)
- Power Output (W)
"""

import numpy as np

def calculate_solar_efficiency(solar_irradiance, panel_efficiency, temperature):
    """
    Calculates the effective solar power output considering temperature effects.

    Parameters:
    - solar_irradiance (float): W/m^2
    - panel_efficiency (float): fraction (e.g., 0.18 for 18%)
    - temperature (float): °C

    Returns:
    - float: power output in W/m^2
    """
    # Simple temperature correction
    temp_corr_factor = 1 - (temperature - 25) * 0.005
    effective_efficiency = panel_efficiency * max(0.0, temp_corr_factor)
    return solar_irradiance * effective_efficiency

if __name__ == "__main__":
    irradiance = 1000.0  # W/m^2
    eff = 0.20  # 20%
    temp = 35.0
    output = calculate_solar_efficiency(irradiance, eff, temp)
    print(f"Solar Output at {temp}C: {output:.2f} W/m^2")
```

C.2.6 Water Storage (F6).

```
### Multi-Agent Refined Payload for "Water Storage":

"""
Stores purified water in tanks, ensuring a reliable and consistent supply.
Key parameters:
- Storage Capacity (L)
- Tank Material
- Water Quality Monitoring
- Safety Features
"""

import numpy as np

class WaterStorage:
    def __init__(self, storage_capacity, tank_material,
                 water_quality_monitoring, safety_features):
        """
        Parameters:
        - storage_capacity (float): total volume in liters
        - tank_material (str): e.g., 'Stainless Steel'
```



```
        - water_quality_monitoring (bool)
        - safety_features (list)
        """
        self.storage_capacity = storage_capacity
        self.tank_material = tank_material
        self.water_quality_monitoring = water_quality_monitoring
        self.safety_features = safety_features

    def calculate_storage_efficiency(self, water_input_rate, water_output_rate):
        """
        Calculate storage efficiency as a function of net inflow vs capacity.
        """
        net_inflow = water_input_rate - water_output_rate
        # Simplified approach: fraction of capacity still available
        storage_eff = (self.storage_capacity - net_inflow) / self.storage_capacity
        return max(0.0, min(storage_eff, 1.0)) * 100

    def simulate_water_quality(self, initial_water_quality, purification_rate):
        """
        Simulate how water quality evolves in storage.

        - initial_water_quality (float): fraction from 0..1
        - purification_rate (float): fraction per hour
        """
        # Example simplistic approach
        final_quality = initial_water_quality * (1 - purification_rate)
        return final_quality * 100

if __name__ == "__main__":
    storage = WaterStorage(
        storage_capacity=1000,
        tank_material="Stainless Steel",
        water_quality_monitoring=True,
        safety_features=["Overflow Protection", "Secure Access"]
    )

    eff = storage.calculate_storage_efficiency(50, 20)
    wq = storage.simulate_water_quality(0.9, 0.1)
    print(f"Storage Efficiency: {eff:.2f}%")
    print(f"Simulated Water Quality: {wq:.2f}%")
```

C.2.7 Distribution (F7).

```
### Multi-Agent Refined Payload for "Distribution":

"""
Delivers purified water to end-users via pipes and faucets.
Key parameters:
- Flow Rate (L/min)
- Water Pressure (Pa or bar)
- Pipe Material (e.g., PVC, copper)
- Faucet and Fixture Type
"""

import numpy as np

def calculate_flow_rate(pipe_diameter, pipe_length, water_pressure):
    """
    Example: flow rate (m^3/s) from simplified laminar flow assumption.

    pipe_diameter (m)
    pipe_length (m)
    water_pressure (Pa)
```



```
    """
    # Simple demonstration formula (not physically rigorous)
    flow_rate = (water_pressure * np.pi * (pipe_diameter / 2)**4) / (8 * pipe_length * 0.001)
    return flow_rate

def calculate_water_pressure(pipe_diameter, pipe_length, flow_rate):
    """
    Inverse of the above: estimate water pressure given flow_rate, pipe specs.
    """
    water_pressure = (8 * pipe_length * 0.001 * flow_rate) / (np.pi * (pipe_diameter / 2)**4)
    return water_pressure

if __name__ == "__main__":
    d = 0.1       # m
    L = 100.0     # m
    P = 100000.0  # Pa
    FR = 0.01     # m^3/s

    calc_fr = calculate_flow_rate(d, L, P)
    calc_p = calculate_water_pressure(d, L, FR)
    print(f"Calculated Flow Rate: {calc_fr:.6f} m^3/s")
    print(f"Calculated Water Pressure: {calc_p:.2f} Pa")
```

C.3 Subfunctions (SF1–SF5)

C.3.1 Sedimentation (SF1).

```
### Multi-Agent Refined Payload for "Sedimentation" (Subfunction):

"""
Removes suspended solids via gravitational settling.
Key parameters:
- Sedimentation Rate
- Retention Time
- Basin Depth and Area
"""

import numpy as np

def calculate_sedimentation_efficiency(particle_size, retention_time, basin_depth):
    """
    Example approach: if the settling distance (velocity * time)
    >= basin_depth => 100% for that particle size.
    """
    settling_velocity = 0.1  # m/h, simplified
    settling_distance = settling_velocity * retention_time

    if settling_distance >= basin_depth:
        return 100.0
    else:
        return (settling_distance / basin_depth) * 100

if __name__ == "__main__":
    ps = 0.1        # micrometers
    rt = 2.0        # hours
    depth = 1.0     # m
    eff = calculate_sedimentation_efficiency(ps, rt, depth)
    print(f"Sedimentation Efficiency: {eff:.2f}%")
```

C.3.2 Membrane Filtration (SF2).

```
### Multi-Agent Refined Payload for "Membrane Filtration" (Subfunction):

"""
```



```
Removes dissolved solids using MF/UF/NF/RO membranes.
Key parameters:
- Membrane Type
- Membrane Material
- Pore Size
- Operating Pressure
- Flow Rate
"""

def membrane_filtration_efficiency(membrane_permeability, operating_pressure, feed_water_turbidity):
    """
    Simplistic: efficiency = (permeability * pressure) / (1 + turbidity)
    """
    filtration_eff = (membrane_permeability * operating_pressure) / (1 + feed_water_turbidity)
    return filtration_eff * 100

if __name__ == "__main__":
    perm = 10.0  # L/m^2*h*bar
    press = 5.0  # bar
    turb = 1.0   # NTU
    eff = membrane_filtration_efficiency(perm, press, turb)
    print(f"Membrane Filtration Efficiency: {eff:.2f}%")
```

C.3.3 Activated Carbon Filtration (SF3).

```
### Multi-Agent Refined Payload for "Activated Carbon Filtration":

"""
Adsorption process removing organic compounds, chlorine, improving taste/odor.
Key parameters:
- Filtration Rate (L/min)
- Contaminant Size (micrometers)
- Activated Carbon Type (coconut/coal)
"""

import numpy as np

def activated_carbon_filtration_efficiency(filtration_rate, contaminant_size, carbon_type):
    if carbon_type == 'coconut':
        efficiency = (filtration_rate / (1 + contaminant_size)) * 0.9
    elif carbon_type == 'coal':
        efficiency = (filtration_rate / (1 + contaminant_size)) * 0.8
    else:
        raise ValueError("carbon_type must be 'coconut' or 'coal'")
    return efficiency * 100

if __name__ == "__main__":
    rate = 5.0  # L/min
    size = 5.0  # micrometers
    ctype = 'coconut'
    eff = activated_carbon_filtration_efficiency(rate, size, ctype)
    print(f"Activated Carbon Filtration Efficiency: {eff:.2f}%")
```

C.3.4 UV Treatment (SF4).

```
### Multi-Agent Refined Payload for "UV Treatment" (Subfunction):

"""
Removes bacteria, viruses, pathogens using ultraviolet light.
Key parameters:
- UV Intensity (mJ/cm^2)
- Exposure Time (s)
- UV Lamp Type
```



```
- Water Flow Rate (L/min)
"""

import numpy as np

def calculate_uv_treatment_efficiency(uv_intensity, exposure_time, water_flow_rate):
    """
    Simplistic model: efficiency ~ (uv_intensity * exposure_time)/(flow_rate+1)
    """
    efficiency = (uv_intensity * exposure_time) / (water_flow_rate + 1)
    return efficiency * 100

if __name__ == "__main__":
    intensity = 100.0
    time_s = 10.0
    flow = 5.0
    eff = calculate_uv_treatment_efficiency(intensity, time_s, flow)
    print(f"UV Treatment Efficiency: {eff:.2f}%")
```

C.3.5 Chemical Disinfection (SF5).

```
### Multi-Agent Refined Payload for "Chemical Disinfection" (Subfunction):

"""
Removes pathogens via chemical dose (chlorine, ozone, etc.).
Key parameters:
- Disinfectant Type
- Disinfectant Dose (mg/L)
- Contact Time (min)
- pH Level
"""

import numpy as np

def calculate_disinfection_efficiency(disinfectant_dose, contact_time, pH_level):
    """
    Example calculation: efficiency ~ (dose * 0.01) + (time * 0.05) + (pH * 0.02)
    """
    eff = (disinfectant_dose * 0.01) + (contact_time * 0.05) + (pH_level * 0.02)
    return eff

if __name__ == "__main__":
    dose = 5.0
    ctime = 30.0
    pH = 7.0
    eff = calculate_disinfection_efficiency(dose, ctime, pH)
    print(f"Chemical Disinfection Efficiency: {eff:.2f}%")
```

C.4 Requirements (R1–R3)

C.4.1 Water Quality Standards (R1).

```
{
  "requirement_name": "Water Quality Standards",
  "description": "Must meet or exceed WHO/EPA standards.",
  "standards": [
    "WHO Guidelines for Drinking-water Quality",
    "EPA National Primary Drinking Water Regulations"
  ],
  "parameters": [
    "pH",
    "Turbidity",
    "Total Coliform",
    "E. coli",
```

```
    "Lead",
    "Copper",
    "Total Trihalomethanes (TTHM)"
  ],
  "thresholds": {
    "pH": [6.5, 8.5],
    "Turbidity": 0.3,
    "Total Coliform": 0,
    "E. coli": 0,
    "Lead": 0.015,
    "Copper": 1.3,
    "TTHM": 0.08
  },
  "testing_frequency": "Quarterly",
  "testing_methodology": "EPA-approved methods"
}
```

C.4.2 Energy Efficiency Targets (R2).

```
### Multi-Agent Refined Payload for "Energy Efficiency Targets":

"""
Minimize energy consumption of the Solar-Powered Water Filtration System
while maintaining performance and water quality.
Key requirements:
- Reduce total system energy by 20% vs. traditional.
- Operate with <= 5 kWh/m^3 overall input.
"""

import numpy as np

def calculate_energy_efficiency(solar_irradiance, panel_efficiency, filtration_energy_demand, pumping_energy):
    """
    Calculate overall system energy efficiency.

    - solar_irradiance (float): W/m^2
    - panel_efficiency (float): fraction
    - filtration_energy_demand (float): kWh
    - pumping_energy (float): kWh
    """
    generated_energy = solar_irradiance * panel_efficiency
    total_energy = filtration_energy_demand + pumping_energy
    if total_energy == 0:
        return 0.0
    return generated_energy / total_energy

if __name__ == "__main__":
    irr = 1000.0
    eff = 0.20
    filt_demand = 2.0
    pump_demand = 1.0
    energy_eff = calculate_energy_efficiency(irr, eff, filt_demand, pump_demand)
    print(f"Energy Efficiency ratio: {energy_eff:.3f}")
```

C.4.3 Maintenance Access (R3).

```
### Multi-Agent Refined Payload for "Maintenance Access":

"""
Ensures the system is designed for easy maintenance and repair,
minimizing downtime.
"""

```



```python
import numpy as np

class MaintenanceAccess:
    def __init__(self, accessibility, tooling, spare_parts, maintenance_schedule):
        self.accessibility = accessibility  # fraction from 0..1
        self.tooling = tooling
        self.spare_parts = spare_parts
        self.maintenance_schedule = maintenance_schedule

    def calculate_maintenance_time(self, component_failure_rate):
        """
        Example: time ~ (1 / failure_rate) * (1 - accessibility)
        """
        return (1 / component_failure_rate) * (1 - self.accessibility)

    def calculate_system_downtime(self, maintenance_time, maintenance_frequency):
        """
        total downtime = maintenance_time * maintenance_frequency
        """
        return maintenance_time * maintenance_frequency

if __name__ == "__main__":
    ma = MaintenanceAccess(0.8, True, True, "Quarterly")
    fail_rate = 0.01  # 1% failure
    mt = ma.calculate_maintenance_time(fail_rate)
    freq = 4
    downtime = ma.calculate_system_downtime(mt, freq)
    print(f"Maintenance Time: {mt:.2f} hrs, System Downtime: {downtime:.2f} hrs")
```

C.5 Constraint: System Boundaries (C1)

```python
### Multi-Agent Refined Payload for "System Boundaries":

"""
Defines the system's scope, including:
- Water Intake Point (raw sources)
- Distribution Point (end-users)
- External Interfaces (electrical grid, etc.)
"""

import numpy as np
import matplotlib.pyplot as plt

def water_flow_model(intake_rate, distribution_rate, hours=24):
    """
    Simulates water storage levels over 'hours' hours.
    """
    storage_levels = np.zeros(hours)
    for i in range(1, hours):
        storage_levels[i] = storage_levels[i-1] + intake_rate - distribution_rate
    return storage_levels

def energy_balance_model(solar_irradiance, panel_efficiency, load_demand, hours=24):
    """
    Simulates net energy over time.
    """
    # For demonstration, treat them as constants for each hour
    net_energy = np.zeros(hours)
    for i in range(hours):
        net_energy[i] = (solar_irradiance * panel_efficiency) - load_demand
    return net_energy

if __name__ == "__main__":
```

```python
    # Example usage
    intk = 10.0    # m^3/h
    dist = 5.0     # m^3/h
    sol_irr = 1000.0
    eff = 0.20
    load = 500.0

    storage = water_flow_model(intk, dist)
    net_en = energy_balance_model(sol_irr, eff, load)

    plt.figure(figsize=(10,5))
    plt.subplot(1,2,1)
    plt.plot(storage)
    plt.title("Water Storage (m^3)")
    plt.subplot(1,2,2)
    plt.plot(net_en)
    plt.title("Net Energy (W)")
    plt.show()
```

C.6 Multi-Agent Design-State Graph Data Structure

```
{
  "nodes": {
    "F1": {
      "node_id": "F1",
      "node_type": "function",
      "name": "Water Intake",
      "payload": "### **Refined Payload:**\n#### Water Intake Function\n... (omitted for brevity) ...",
      "status": "refined",
      "edges_in": [],
      "edges_out": []
    },
    "F2": {
      "node_id": "F2",
      "node_type": "function",
      "name": "Pre-Filtration",
      "payload": "### **Refined Payload:**\n#### Pre-Filtration Function\n... (omitted) ...",
      "status": "refined",
      "edges_in": [],
      "edges_out": []
    },
    "...": "...Additional nodes for F3, F4, etc..."
  },
  "edges": [
    ["F3", "SF1"],
    ["F3", "SF2"],
    ["F3", "SF3"],
    ["F1", "F2"],
    ["F2", "F3"],
    ["F3", "F4"],
    ["F4", "F6"],
    ["F6", "F7"],
    ["F5", "F1"],
    ["R1", "F4"],
    ["R2", "F5"],
    ["C1", "F1"],
    ["C1", "F7"],
    ["F1", "F2"],
    ["F2", "F3"],
    ["F3", "F4"],
    ["F4", "F6"],
    ["F6", "F7"],
    ["F5", "F1"],
```



```
      ["R1", "F4"],
      ["R2", "F5"],
      ["R3", "F7"]
    ]
  }
```

APPENDIX D. TWO-AGENT SYSTEM RESULTS (GENERATOR & REFLECTION)

This appendix presents the final design produced by the two-agent system: a generator agent proposing designs and a reflection agent that refines and concludes the process when satisfied. The resulting Solar-Powered Water Filtration system includes a functional decomposition, subsystem mapping, and a comprehensive Python implementation. Table 6 summarizes the final functional decomposition, subsystems, and numerical models.

TABLE 6: FUNCTIONAL DECOMPOSITION, SUBSYSTEMS, AND NUMERICAL MODELS (TWO-AGENT SYSTEM)

Function	Subsystem	Numerical Model(s)
Water Intake & Pre-Filtration	Water Collection System	`simulate_operation()` (within class), `calculate_flow_rate()` (conceptual)
Primary Filtration	Filtration Unit	`calculate_efficiency()`, `simulate_operation()`
Advanced Purification	Filtration Unit (Extended)	`life_cycle_assessment()`, `calculate_fail_safe_cost()`
Solar Power Generation & Storage	Solar Power System	`calculate_efficiency()`, `calculate_sensitivity_analysis()`
Water Storage & Distribution	Storage Tank + Distribution	`calculate_storage_level()`, `calculate_scalability_cost()`
Monitoring & Automation	Control & Monitoring System	`calculate_automation_cost()`, `calculate_user_interface_cost()`

D.1 Python Implementation

Below is the final Python code integrating subfunction logic, system variants, cost analysis, and sustainability metrics. Since the 2AS does not use a DSG, it provides all the code in a single Python script:

```python
import numpy as np

class SolarWaterFiltrationSystem:
    def __init__(
        self,
        filtration_rate,         # L/h
        solar_efficiency,        # fraction (e.g. 0.5 for 50%)
        energy_consumption,      # W
        storage_capacity,        # L
        material_cost,           # $
        labor_cost,              # $
        maintenance_cost         # $ per year
    ):
        self.filtration_rate = filtration_rate
        self.solar_efficiency = solar_efficiency
        self.energy_consumption = energy_consumption
        self.storage_capacity = storage_capacity
        self.material_cost = material_cost
        self.labor_cost = labor_cost
        self.maintenance_cost = maintenance_cost

    def calculate_efficiency(self):
        """
        Overall system efficiency (%).
        Efficiency ~ (filtration_rate * solar_efficiency) / energy_consumption
        """
        if self.energy_consumption == 0:
            return 0.0
        eff = (self.filtration_rate * self.solar_efficiency) / self.energy_consumption
        return eff * 100

    def simulate_operation(self, hours):
```



```
        """
        Total liters filtered over 'hours'.
        """
        return self.filtration_rate * hours

    def calculate_storage_level(self, hours):
        """
        Storage tank level (%) after 'hours'.
        """
        total_liters = self.simulate_operation(hours)
        if total_liters >= self.storage_capacity:
            return 100.0
        return (total_liters / self.storage_capacity) * 100.0

    def calculate_total_cost(self, years):
        """
        Materials + labor + (maintenance * years).
        """
        return self.material_cost + self.labor_cost + (self.maintenance_cost * years)

    def life_cycle_assessment(self, ec, mat_cost, lab_cost, maint_cost, years):
        """
        Rough LCA score.
        ec: energy_consumption (kWh eq.)
        mat_cost, lab_cost, maint_cost, years
        """
        total_energy = ec * years * 24 * 365
        total_material = mat_cost / 10.0
        total_labor = lab_cost / 50.0
        total_maintenance = maint_cost * years
        return total_energy * 0.5 + total_material * 10 + total_labor * 50 + total_maintenance

    def calculate_eco_friendly_materials_cost(self, material_type, quantity):
        if material_type == "recyclable":
            return quantity * 10
        elif material_type == "biodegradable":
            return quantity * 15
        return 0

    def calculate_sensitivity_analysis(
        self,
        filtration_rate, solar_eff, energy_cons, stor_cap,
        mat_cost, lab_cost, maint_cost
    ):
        """
        Dummy sensitivity function combining multiple factors.
        """
        return (
            filtration_rate * 0.1 +
            solar_eff * 0.2 +
            energy_cons * 0.3 +
            stor_cap * 0.4 +
            mat_cost * 0.5 +
            lab_cost * 0.6 +
            maint_cost * 0.7
        )

    def calculate_user_interface_cost(self, interface_type, complexity):
        if interface_type == "simple":
            return complexity * 100
        elif interface_type == "medium":
            return complexity * 200
```

```
        elif interface_type == "complex":
            return complexity * 500
        return 0

    def calculate_automation_cost(self, automation_level, complexity):
        if automation_level == "low":
            return complexity * 500
        elif automation_level == "medium":
            return complexity * 1000
        elif automation_level == "high":
            return complexity * 2000
        return 0

    def calculate_scalability_cost(self, scale_factor):
        return scale_factor * (self.material_cost + self.labor_cost)

    def calculate_maintenance_cost(self, maintenance_frequency, cost_per_event):
        return maintenance_frequency * cost_per_event

    def calculate_compliance_cost(self, compliance_level):
        if compliance_level == "low":
            return 1000
        elif compliance_level == "medium":
            return 2000
        elif compliance_level == "high":
            return 5000
        return 0

    def calculate_fail_safe_cost(self, fail_safe_level):
        if fail_safe_level == "low":
            return 500
        elif fail_safe_level == "medium":
            return 1000
        elif fail_safe_level == "high":
            return 2000
        return 0

# Example usage
if __name__ == "__main__":
    system = SolarWaterFiltrationSystem(
        filtration_rate=10,
        solar_efficiency=0.5,
        energy_consumption=20,
        storage_capacity=1000,
        material_cost=200,
        labor_cost=100,
        maintenance_cost=50
    )
    eff = system.calculate_efficiency()
    total_liters_24h = system.simulate_operation(24)
    storage_level_24h = system.calculate_storage_level(24)
    cost_10y = system.calculate_total_cost(10)
    lca_score = system.life_cycle_assessment(20, 200, 100, 50, 10)

    print(f"System Efficiency: {eff:.2f}%")
    print(f"Liters in 24h: {total_liters_24h}")
    print(f"Storage Level after 24h: {storage_level_24h:.2f}%")
    print(f"10-Year Cost: ${cost_10y:.2f}")
    print(f"LCA Score: {lca_score:.2f}")

    # Sample variants
    design_variants = [
```

```
    {"filtration_rate":10,"solar_efficiency":0.5,"energy_consumption":20,"storage_capacity":1000,"material_cost":200,"
    {"filtration_rate":15,"solar_efficiency":0.6,"energy_consumption":25,"storage_capacity":1500,"material_cost":300,"
    {"filtration_rate":20,"solar_efficiency":0.7,"energy_consumption":30,"storage_capacity":2000,"material_cost":400,"
]

for variant in design_variants:
    sys_variant = SolarWaterFiltrationSystem(
        variant["filtration_rate"],
        variant["solar_efficiency"],
        variant["energy_consumption"],
        variant["storage_capacity"],
        variant["material_cost"],
        variant["labor_cost"],
        variant["maintenance_cost"]
    )
    var_eff = sys_variant.calculate_efficiency()
    var_storage = sys_variant.calculate_storage_level(24)
    var_cost_10y = sys_variant.calculate_total_cost(10)
    var_lca = sys_variant.life_cycle_assessment(
        variant["energy_consumption"],
        variant["material_cost"],
        variant["labor_cost"],
        variant["maintenance_cost"],
        10
    )
    print(f"Design Variant => Filtration Rate={variant['filtration_rate']}, Solar Eff={variant['solar_efficiency']}, St
    print(f" - Efficiency: {var_eff:.2f}%")
    print(f" - 24h Storage: {var_storage:.2f}%")
    print(f" - 10-Year Cost: ${var_cost_10y:.2f}")
    print(f" - LCA Score: {var_lca:.2f}\n")
```