INTERNSHIP REPORT

AI-Driven Factory Layout Generation from Natural Language Prompts

Dhanush V
Metaverse AI Intern, Makers Lab
Tech Mahindra

Academic Details:
B.Tech (Computer Science Engineering)
National Institute of Technology Karnataka

Internship Duration:
19th May 2025 – 18th July 2025

Acknowledgement

I extend my deepest gratitude to everyone who supported me throughout my internship at Makers Lab, Tech Mahindra. My sincere thanks go to Mr. Rahul Marathe, whose exceptional leadership and strategic vision were pivotal in guiding this project and shaping our approach to increasing ChatUSD's functionalities by adding agents.

I am especially thankful to Sravanthi Ma’am, my dedicated mentor. Her meticulous guidance, unwavering support, and invaluable technical insights were absolutely essential to the success of this research. Her mentorship significantly deepened my understanding of agent-based systems and 3D modeling, contributing immensely to my professional growth.

I also wish to acknowledge Mr. Yash for his collaborative spirit and assistance, which ensured smooth project execution and fostered a truly supportive working environment.

Finally, I am profoundly grateful to the National Institute of Technology Karnataka for providing this remarkable opportunity. This internship has been an invaluable experience, effectively bridging my academic learning with real-world application, and allowing me to explore cutting-edge AI-driven 3D modeling technologies within a professional research setting.

Contents

Abstract

This report details a pivotal R&D initiative to revolutionize factory layout generation through natural language prompts within NVIDIA Omniverse's ChatUSD platform. Initially, ChatUSD faced significant limitations in handling complex, spatially-aware commands, manifesting as objects spawning at the origin (0,0,0) resulting in cluttered scenes, undetected ground meshes leading to floating or misaligned components, absence of collision detection causing unrealistic overlaps, and poor interpretation of spatial instructions like "Place the forklift behind the container on the ground".

Our comprehensive approach involved a multi-phase improvement strategy, rooted in problem decomposition, spatial intelligence, and robust execution. Phase 1 established foundational agent extensibility and inter-agent communication within ChatUSD's multi-agent architecture. This involved developing a custom "Greeting Agent" to validate agent plug-in capabilities and message routing, successfully retrieving basic scene data such as prims and properties. Phase 2 introduced a "Prompt Simplifier Agent" to address model hallucinations and unreliable code generation stemming from complex prompts. This agent converts intricate natural language commands into clear, executable, sequential steps, significantly enhancing accuracy in 3D scene generation, reducing ambiguity, and providing users with predictable, step-wise control. Phase 3 focused on augmenting spatial intelligence. Initially, a dedicated "Spatial Agent" was developed to interpret complex spatial relationships (e.g., "on," "under," "beside"), check object existence, and plan object placements with built-in fallbacks. This phase further explored sophisticated multi-model routing by integrating Gemini 2.5 Flash for advanced spatial reasoning, dynamically routing prompts based on detected spatial context while meticulously preserving the core ChatUSD architecture. This project successfully enabled AI-driven, prompt-based factory layout design, thereby effectively bridging the

critical gap between human intent expressed in natural language and the precise, complex requirements of 3D scene construction for industrial environments.

Introduction

Motivation for Internship

My motivation for pursuing this internship stemmed from a profound interest in the rapidly evolving intersection of artificial intelligence and 3D model generation, particularly its immense potential for transforming industrial applications. I was especially drawn to the challenge of addressing the inherent limitations of existing AI systems in handling complex, spatially-aware natural language prompts for 3D scene creation. The opportunity to directly tackle issues like objects spawning at the origin (0,0,0) leading to cluttered scenes, undetected ground meshes causing misalignment or floating objects, the absence of collision detection resulting in unrealistic overlaps, and the poor interpretation of nuanced spatial instructions (e.g., "Place the forklift behind the container on the ground") aligned perfectly with my academic background in Engineering Design and my passion for developing innovative, robust manufacturing solutions.

About the Company

Makers Lab, an innovation hub within Tech Mahindra, is at the forefront of emerging technologies and digital transformation solutions. The lab specializes in developing advanced applications across artificial intelligence, machine learning, and industrial automation. Operating as a dedicated research and development center within Tech Mahindra's broader ecosystem, Makers Lab plays a crucial role in transforming innovative ideas into practical solutions for pressing industry challenges. Its mission to bridge the gap between academic research and industrial applications established an ideal environment for my exploration into novel approaches for AI-driven design and manufacturing.

Department and Team

I was assigned to the Meta-verse team within Makers Lab, a specialized unit focused on developing intelligent systems for industrial applications. This multidisciplinary team comprises dedicated researchers, engineers, and data scientists. Their work encompasses various critical areas, including computer vision and 3D modeling, natural language processing applications, industrial automation and robotics, and the creation of AI-assisted design tools. My specific contribution centered on enhancing the intelligence and reliability of 3D scene generation directly from natural language prompts, with a distinct focus on improving spatial reasoning and prompt decomposition capabilities, thereby differentiating my work from broader 3D modeling or pure automation tasks.

Work Details

Current ChatUSD Architecture

ChatUSD serves as a specialized AI assistant integrated within NVIDIA Omniverse, designed specifically for Universal Scene Description (USD) development. Its core function is to enable natural language commands for the creation, modification, and querying of 3D scenes. This functionality is critical for tasks like industrial 3D scene creation, allowing users to interact with complex 3D environments using intuitive language rather than intricate software interfaces.

At its heart, ChatUSD is built upon a robust multi-agent architecture. This design routes user queries to specialized AI agents based on the detected intent of the command. The system's operation begins with the

ChatUSD Network Node, which receives user inputs. These inputs are then forwarded to the

ChatUSD Supervisor Node. The Supervisor Node acts as the central orchestrator, analyzing the query to determine its purpose and subsequently dispatching it to the most appropriate specialized agent.

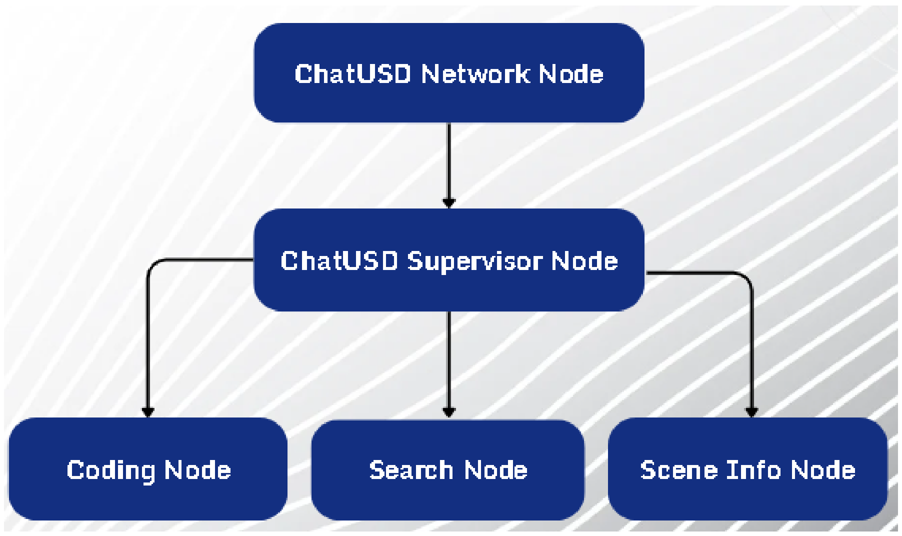


Fig 1: Original Architecture of ChatUSD

The specialized agents within this architecture include:

- Coding Node: Responsible for generating or modifying USD code to create or alter 3D scene elements.
- Search Node: Likely handles queries related to finding existing assets or information within the USD scene.
- Scene Info Node: Provides details and properties about the current state of the 3D scene, such as prims (primitives) and their attributes.

This modular, multi-agent framework allows ChatUSD to manage various aspects of 3D scene interaction efficiently, delegating complex tasks to agents with specific functionalities.

Problem Statement: Enhancing ChatUSD Functionality for Real-World Use

Despite its innovative multi-agent architecture, ChatUSD initially faced significant challenges when confronted with complex, spatially-aware natural language prompts. These limitations severely restricted its practical effectiveness, particularly for the precise and intricate requirements of industrial 3D scene creation, such as factory layouts.

The common issues encountered, which necessitated the R&D efforts, were:

- **Lack of Spatial Context**: A primary problem was that all spawned objects would default to the origin point (0,0,0) in the 3D space. This resulted in severely cluttered and impractical scenes, requiring extensive manual correction.

- **Ground Mesh Detection Failure**: The system frequently failed to detect the ground mesh, leading to objects appearing to float unrealistically or being misaligned within the environment. This undermined the visual and functional integrity of the generated scenes.
- **Absence of Collision Detection**: ChatUSD lacked inherent collision detection capabilities, meaning objects could overlap or intersect without constraint. This made generated scenes physically unrealistic and unsuitable for engineering design validation, where object interference is a critical consideration.
- **Poor Handling of Spatial Instructions**: The baseline system struggled profoundly with interpreting and executing natural language commands that involved spatial relationships. A prime example of such a problematic prompt was: "Place the forklift behind the container on the ground". The AI could not reliably translate these relative spatial directives into accurate 3D placements.

These challenges highlighted a critical gap between human intent, expressed naturally, and ChatUSD's ability to create precise, functional, and realistic industrial 3D environments. The project's core motivation was to overcome these hurdles, thereby enhancing ChatUSD's capability for robust, AI-driven factory layout design.

Problem Decomposition: My Approach to Improvement

To effectively address ChatUSD's limitations in handling complex, spatially-aware natural language prompts for 3D scene creation, my R&D journey was structured around a systematic problem decomposition approach. This involved breaking down the overarching challenge into more manageable, interconnected sub-problems. My initial R&D specifically focused on understanding ChatUSD's existing prompt processing and agent delegation mechanisms, which was crucial for designing a decomposition strategy that could seamlessly integrate into the current workflow.

The core components of this decomposition were:

1. **Prompt Understanding**: At the foundational level, the system needed to accurately grasp the user's intent from the natural language input. This is more than just recognizing keywords; it involves comprehending the full scope of the request, including implied actions and relationships.
2. **Prompt Decomposition**: Once understood, complex, multi-faceted prompts needed to be broken down into simpler, atomic (indivisible) executable steps. This transformation was essential for mitigating issues like model hallucinations and unreliable code generation that arose from directly processing overly complex instructions.

3. **Spatial Reasoning**: A critical deficiency identified was ChatUSD's inability to understand and apply spatial relationships. This sub-problem focused on endowing the system with the intelligence to interpret terms like "on," "under," "beside," and to calculate appropriate object positions in 3D space.
4. **Execution & Scene Generation**: The decomposed and spatially reasoned instructions then needed to be accurately translated into USD code and rendered into a coherent 3D scene. This involved ensuring that the generated code was correct and that the scene reflected all the user's requirements, including proper placement and alignment.
5. **Model Orchestration**: Given ChatUSD's multi-agent architecture and the introduction of new AI models (like Gemini 2.5 Flash), managing the flow of information and task delegation between different agents and models became a distinct sub-problem. This ensured that the right AI component handled the right part of the prompt at the right time.

Phase 0: Initial Learning and Exploration

The initial period of the internship commenced with an intensive learning and exploration phase, working specifically with **Kit version 106.4**, which was the initial launch model of the NVIDIA Omniverse platform at the time. This phase was crucial for familiarizing ourselves with the foundational aspects of ChatUSD and its underlying architecture.

Our primary activity during this phase involved a deep dive into the official **ChatUSD documentation provided by NVIDIA**. This documentation served as our guide to understanding the system's components, functionalities, and intended use.

To gain a comprehensive understanding of the internal mechanics of ChatUSD, we undertook the task of **adding extensive logging to the existing codebase**. This allowed us to meticulously trace the flow of prompts: how a user's natural language input was received, processed, and subsequently passed from one internal component or agent to the next within the architecture. This granular insight into the prompt's journey was vital for identifying potential areas for modification and improvement.

A key objective during this phase was to **attempt the integration of a new, custom agent into the original ChatUSD architecture**. While we were successful in incorporating the new agent into the model's structure, we encountered a significant challenge: we were **unable to effectively access or leverage its functionalities within the ChatUSD interface**. This limitation highlighted an area for deeper investigation regarding agent registration and interaction within the then-current framework.

Subsequently, we discovered the availability of a **newer Kit version, 107.3, bundled with an updated ChatUSD version 2.0.4**. Recognizing the potential for improved functionality and resolved issues, we transitioned our development environment to this newer version. Upon shifting to Kit 107.3, and by diligently following the updated ChatUSD documentation, we were successfully able to **integrate and access the functionalities of our custom agents** through ChatUSD. This breakthrough marked the successful conclusion of our initial learning curve and set the stage for subsequent developmental phases focused on enhancing ChatUSD's capabilities.

Phase 1: Foundations - Custom Agent & Inter-Agent Communication

Following the successful transition to Kit version 107.3, Phase 1 was dedicated to a critical milestone: successfully integrating and accessing the functionalities of a custom agent within the ChatUSD framework. This phase aimed to validate the extensibility of ChatUSD's multi-agent architecture and establish effective inter-agent communication, which was foundational for all subsequent enhancements.

Goal: To assess the feasibility of adding new, custom agents to ChatUSD and to understand how these agents could communicate and exchange information with existing specialized agents, such as the `SceneInfoNetworkNode`.

Approach: Building and Integrating the Greeting Agent To achieve this, we developed a simple yet illustrative "Greeting Agent." This agent's core function was to respond to user greetings (e.g., "Hello," "Hi") and, critically, to fetch and present basic information about the current 3D scene. This allowed us to test agent integration and data flow in a controlled environment. The implementation involved three key files for integration: `extension.py`, `chat_usd_network_node.py`, `chat_usd_supervisor_identity.md`, alongside our agent's logic in `trial_network_node.py` and `trial_node.py`.

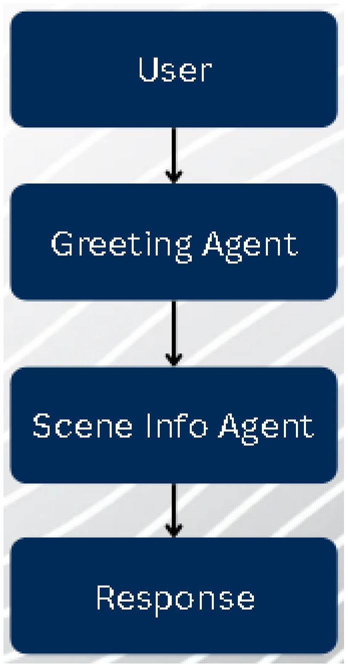


Fig 2. Workflow diagram of Greeting agent

1. Defining the Custom Network Node (`trial_network_node.py`)

The `trial_network_node.py` file defines `TrialNetworkNode`, which serves as the entry point for our custom agent in the ChatUSD system. It inherits from `lc_agent.NetworkNode`.

```
from lc_agent import NetworkNode  # Importing the base class for creating a single-agent network node.

import carb  # Carb is used for logging/debugging inside the Omniverse environment.

from omni.ai.langchain.agent.usd_code import SceneInfoNetworkNode  # This helps gather information about the 3D scene.
```

Key aspects of `TrialNetworkNode`:

- **`__init__` method**: Initializes the node, sets a log message using `carb.log_info`, and defines `self.default_node = "TrialNode"`, indicating that `TrialNode` (defined in `trial_node.py`) is the primary logic node for this network.

```
def __init__(self, scene_info=True, **kwargs):

        super().__init__(**kwargs)

        carb.log_info("TrialNetworkNode initialized")  # Log message 
to indicate successful creation.

        self.default_node = "TrialNode"  # Specify the name of the 
logic node used in this network.
```

- **`metadata` attribute**: Crucial for the ChatUSD Supervisor. The `description` explains the agent's purpose, and `examples` provides sample user inputs that should trigger this agent. This metadata is parsed by the Supervisor to decide when to activate `TrialNetworkNode`.

```
# Metadata to describe the purpose of this agent.

        self.metadata["description"] = """Agent to print Hello World 
when user says hello"""

        self.metadata["examples"] = [

            "Hello",

            "Hi",

        ]
```

- **`on_begin_invoke_async(self, network)` method**: This asynchronous method is called when `TrialNetworkNode` is invoked. It defines the workflow for the agent:
  - It constructs a highly specific `question` for the `SceneInfoNetworkNode`. This question dictates exactly what kind of 3D scene information is required (e.g., only mesh/shape prims under `/World`, specific parameters like bounding box, position, scale, rotation, and a precise output format).
  - It instantiates `SceneInfoNetworkNode` with this detailed question.
  - It then imports and instantiates our custom `TrialNode`.
  - The crucial line `self >> scene_info_node >> trial_node` establishes the chaining of these nodes. This means `TrialNetworkNode` initiates `SceneInfoNetworkNode`, and once

SceneInfoNetworkNode completes its task, its output is automatically passed as input to TrialNode.

2. Defining the Custom Logic Node (trial_node.py)

The trial_node.py file defines TrialNode, which contains the core logic for processing the scene information retrieved by SceneInfoNetworkNode and formulating the greeting response. It inherits from lc_agent.RunnableNode.

```
from lc_agent import RunnableNode   # Base class for building logic nodes.

import carb  # Used to log useful messages to the Omniverse console

from ..utils.chat_model_utils import sanitize_messages_with_expert_type  # 
Utility for preparing messages for LLMs
```

Key aspects of TrialNode:

- **__init__ method**: Simple initialization and logging.
- **on_begin_invoke_async(self, network) method**: This method is crucial for inter-agent communication. It iterates through the network.parents (nodes that ran before it in the chain) to find an instance of SceneInfoNetworkNode. Once found, it retrieves the processed scene information via parent.outputs.content and stores it in self._scene_info. This is the direct mechanism by which data flows from one agent (SceneInfoNetworkNode) to another (TrialNode).

```
async def on_begin_invoke_async(self, network):

        """

        Called once when the network is first invoked.

        This method looks at the parent nodes (who ran before me)

        and tries to get the scene description result from the 
SceneInfoNetworkNode.

        That way, I can display what it found.

        """
        scene_info = None

        for parent in network.parents:

            from omni.ai.langchain.agent.usd_code import
SceneInfoNetworkNode

            if isinstance(parent, SceneInfoNetworkNode):

                scene_info = parent.outputs.content

                break  # Found the node we care about; no need to
keep searching



        # If scene_info was found, we store it. Otherwise, set to a
default message.

        self._scene_info = scene_info if scene_info else "[No scene
information available]"
```

- **`_sanitize_messages_for_chat_model` method**: This is where the output to the user is constructed. It takes the original messages, and dynamically injects the fetched `scene_info` along with a custom greeting and a summary. This modified message is what is then passed to the underlying Language Model, ensuring the AI's response is contextualized with scene details.

```
def _sanitize_messages_for_chat_model(self, messages,
chat_model_name, chat_model):

        """

        This function is called right before we send the user's
message to the language model.

        Here we can add context and make the message more
intelligent by injecting what we know about the scene.

        """


        messages =
super()._sanitize_messages_for_chat_model(messages, chat_model_name,
chat_model)


        # Step 1: Write a friendly intro
        greeting = "Hello! I've analyzed the current scene and
here's what I found:\n\n"


        # Step 2: Access the scene information we stored earlier
        scene_info = getattr(self, '_scene_info', '[No scene
information available]')


        # Step 3: Add a quick human-readable summary
        summary = "Scene Analysis Summary:\n"
        if "No /World prim found" in scene_info:
            summary += "- No /World prim found in the scene\n"
        elif "Found:" in scene_info or "Found nested:" in
scene_info:
            summary += "- Found mesh/shape prims under /World\n"
        else:
            summary += "- Scene structure analyzed\n"


        if "Ground-related prim:" in scene_info:
            summary += "- Ground-related prims detected\n"
        else:
            summary += "- No ground-related prims found\n"

        summary += "\nDetailed Scene Information:\n"

        # Combine all parts into a complete message
        combined_message = f"{greeting}{summary}{scene_info}"

        # Replace the last message's content with the final scene
summary
        if messages:
            msg = messages[-1]
            if isinstance(msg, dict):
                msg["content"] = combined_message
                if not msg["content"] or not
str(msg["content"]).strip():
                    msg["content"] = "Hello! [No scene information
available]"
            elif hasattr(msg, "content"):
                msg.content = combined_message
                if not msg.content or not str(msg.content).strip():
                    msg.content = "Hello! [No scene information
available]"

        # Finally sanitize and return messages to the LLM
        return sanitize_messages_with_expert_type(messages, "code",
rag_max_tokens=0, rag_top_k=0)
```

3. Registering the Node in `extension.py`

For Omniverse to discover and load our `TrialNetworkNode`, it must be registered within the `extension.py` file of our custom Omniverse Extension. This file acts as the entry point for the extension.

```
from .trial.trial_node import TrialNode

from .trial.trial_network_node import TrialNetworkNode
```

This line ensures that our `TrialNetworkNode` class is available within the `extension.py` scope. While the explicit registration call might be abstracted away by a factory or a bundle mechanism in the provided `extension.py`, its inclusion here confirms that the system is aware of and attempts to load `TrialNetworkNode` as part of the `omni.ai.chat_usd.bundle` extension.

These lines directly demonstrate how `TrialNode` and `TrialNetworkNode` are registered.

- `get_node_factory().register(TrialNode, hidden=True)`: This registers the `TrialNode` (our core logic node) with the node factory. `hidden=True` suggests it's not meant to be directly exposed as a top-level tool to the user but rather used internally by a `NetworkNode`.
- `get_node_factory().register(TrialNetworkNode, name="Trial Agent")`: This registers the `TrialNetworkNode` with a human-readable name "Trial Agent". This makes it discoverable within the Omniverse environment.

```
 # Register Trial Agent - Always register for testing

        carb.log_info("Registering Trial Agent")

        get_node_factory().register(TrialNode, hidden=True)

        get_node_factory().register(TrialNetworkNode, name="Trial
Agent")
```

```
        carb.log_info("Trial Agent registered successfully")
```

- `get_node_factory().register(TrialNetworkNode, name="ChatUSD_Trial", hidden=True)`: This is a **crucial additional registration** of the *same* `TrialNetworkNode` but under the specific alias **"ChatUSD_Trial"**. This alias is what the `ChatUSDNetworkNode` and `chat_usd_supervisor_identity.md` will use to refer to our agent. Making it `hidden=True` means it's primarily an internal routing name rather than a user-facing one. This explicit registration under the "ChatUSD_Trial" name is what allows the ChatUSD system to identify and route to it.

```
# Register Trial Node as a Chat USD tool

            get_node_factory().register(

                TrialNetworkNode,

                name="ChatUSD_Trial",

                hidden=True,

            )
```

The `extension.py` also contains unregistration calls for "ChatUSD_Trial", which implies a corresponding registration somewhere during the extension's startup. The presence of `get_node_factory().unregister("ChatUSD_Trial")` in the `on_shutdown` method strongly suggests that a factory mechanism is used to register `TrialNetworkNode` (likely under the alias "ChatUSD_Trial") during the extension's `on_startup` phase.

```
get_node_factory().unregister(TrialNode)

        get_node_factory().unregister(TrialNetworkNode)

        get_node_factory().unregister("ChatUSD_Trial")  # Unregister the
Chat USD version
```

4. Adding as a Tool and Routing in `chat_usd_network_node.py`

The `chat_usd_network_node.py` file contains the definition for `ChatUSDNetworkNode`, which is a `MultiAgentNetworkNode`. This node is responsible for routing user queries to the appropriate specialized agents. Our `TrialNetworkNode` needs to be listed as a valid route.

```
default_node: str = "ChatUSDSupervisorNode"

    route_nodes: List[str] = [

        "ChatUSD_USDCodeInteractive",

        "ChatUSD_USDSearch",

        "ChatUSD_SceneInfo",

        "ChatUSD_Trial",

        "ChatUSD_Trial2",

    ]
```

The inclusion of `"ChatUSD_Trial"` in the `route_nodes` list explicitly tells the `ChatUSDNetworkNode` (and by extension, the Supervisor it manages) that "ChatUSD_Trial" is an available agent that it can dispatch queries to. This is a direct linkage that allows our custom agent to be part of the overall multi-agent system.

5. Updating `chat_usd_supervisor_identity.md`

To make the `TrialNetworkNode` accessible via natural language prompts, the `chat_usd_supervisor_identity.md` file needs to be updated. This Markdown file serves as a prompt for the ChatUSD Supervisor, guiding it on which agents to invoke based on user input.

```
1. ChatUSD_Trial [PRIORITY FOR GREETINGS]

   - Handles all greeting messages and basic interactions

   - Must be consulted FIRST for any greeting or hello messages
   - When a greeting is detected, ALWAYS first call ChatUSD_SceneInfo to
get the current scene information, then combine the greeting with the
scene information in the response.

   - Examples of messages to route to ChatUSD_Trial:

     * "hello"

     * "hi"

     * "hey"

     * Any greeting or salutation

   - Should be used before any other functions for greeting messages

   - When calling ChatUSD_SceneInfo for greetings, ONLY request the
bounding box, coordinates, scale, and rotation for prims. Do not include
other scene information.
```

This section is paramount. It explicitly instructs the Supervisor to:

- Give `ChatUSD_Trial` **priority for greeting messages**.
- Specify that it **must be consulted FIRST** for "hello" or "hi" messages.
- Detail the precise workflow: upon detecting a greeting, it should **ALWAYS first call `ChatUSD_SceneInfo`** to gather scene details and then combine this information with the greeting in its response.
- Provide **concrete examples** of user inputs that should trigger this agent.
- Even dictate the specific format and content required when `ChatUSD_SceneInfo` is called as part of the greeting workflow.

By completing these steps – defining the network and logic nodes, establishing their communication, and correctly registering and describing them within the Omniverse extension and ChatUSD Supervisor identity – we successfully integrated and accessed our custom Greeting Agent. This achievement was pivotal as it validated the modularity and extensibility of the ChatUSD platform, providing clear insights into how to build and orchestrate more complex AI agents for sophisticated 3D scene generation tasks.

Phase 2: Enhancing Reliability – Prompt Simplification (Detailed)

Building upon the foundational understanding of agent integration established in Phase 1, Phase 2 directly addressed a critical challenge: the unreliability and inconsistencies that arose when ChatUSD's underlying Language Model (LLM) was directly fed complex, multi-faceted natural language prompts. This phase introduced a specialized "Decomposer Agent" (also referred to as the Prompt Simplifier Agent) to mitigate these issues.

Problem Addressed:

- **Model Hallucinations**: Complex prompts often led the LLM to generate irrelevant, incorrect, or nonsensical outputs.
- **Unreliable Code Generation**: The ambiguity in lengthy prompts resulted in USD code that was often flawed, incomplete, or semantically incorrect, failing to accurately represent the user's intent.
- **Inconsistent 3D Scene Outputs**: Even minor variations in complex prompts could lead to drastically different and unpredictable 3D scene creations, hindering a systematic design process.
- **Lack of Step-by-Step Control**: Users had no insight into how their complex requests were being processed, making debugging and fine-tuning impossible.

Solution: The Decomposer Agent (`decomposer_networknode.py` and `decomposer_node.py`) To solve these problems, a new agent, composed of `DecomposerNetworkNode` and `DecomposerNode`, was implemented. Its core function is to intelligently break down an intricate natural language prompt into a series of simple, atomic (indivisible), and executable steps. This structured approach significantly improves the LLM's ability to generate accurate and reliable USD code.

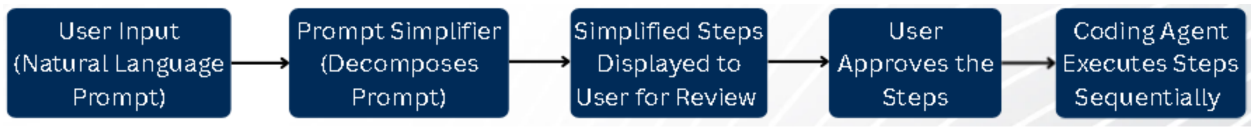

User Input (Natural Language Prompt)
Prompt Simplifier (Decomposes Prompt)
Simplified Steps Displayed to User for Review
User Approves the Steps
Coding Agent Executes Steps Sequentially

Fig 3. Workflow diagram of Decomposition Agent

1. The Decomposer Network Node (`decomposer_networknode.py`)

This file defines `DecomposerNetworkNode`, which serves as the entry point for the Decomposer Agent within the ChatUSD multi-agent system. It is a `MultiAgentNetworkNode` itself, indicating it can potentially route to other nodes, though in its simplest form, it primarily routes to its default logic node.

```
from lc_agent import MultiAgentNetworkNode
from .decomposer_node import DecomposerNode
```

Key aspects of `DecomposerNetworkNode`:

- **`default_node: str = "DecomposerNode"`**: This line explicitly sets `DecomposerNode` as the primary logical component that this network node will run. When `DecomposerNetworkNode` is activated by the Supervisor, it automatically invokes `DecomposerNode`.

  ```
  default_node: str = "DecomposerNode" # Specifies the default logic
  node to run.

      route_nodes = ["DecomposerNode"] # Explicitly states that it
  routes to DecomposerNode.
  ```

- **`route_nodes = ["DecomposerNode"]`**: This confirms that `DecomposerNode` is the (currently only) internal route managed by this network node.
- **`self.metadata["description"]`**: Provides a brief description to the ChatUSD Supervisor, helping it understand the agent's purpose. This description is used in conjunction with the Supervisor's identity markdown file (`chat_usd_supervisor_identity.md`) to route queries.

  ```
  # Metadata to describe the purpose of this agent for the Supervisor.

          self.metadata["description"] = "Network for decomposing
  prompts into simplest form."
  ```

2. The Decomposer Logic Node (`decomposer_node.py`)

This file defines `DecomposerNode`, which contains the core logic for the prompt decomposition process. It inherits from `lc_agent.RunnableNode` and leverages an LLM to perform the decomposition based on a carefully crafted system prompt.

Key aspects of `DecomposerNode`:

- **`_sanitize_messages_for_chat_model` method**: This is the most critical function in `DecomposerNode`. It's automatically invoked before the user's message is sent to the underlying LLM. Its purpose is to inject a detailed `system_prompt` into the conversation. This `system_prompt` acts as a set of instructions and constraints for the LLM, guiding its behavior during the decomposition process.
- **The `system_prompt` content**: This prompt is meticulously crafted to ensure the LLM performs the desired decomposition:
  - It establishes the LLM's role as an "expert at breaking down complex scene-building instructions."
  - It explicitly instructs the LLM to output a "numbered list of simple, direct prompts, one per line," that a separate code agent can execute.
  - **Crucially, it includes "IMPORTANT GROUND PLACEMENT RULES"**: This section addresses the initial problem statement regarding poor spatial placement. It instructs the LLM to *assume objects should be placed on the ground mesh* under `/Environment`, specifies what constitutes a "ground mesh," and provides guidance on *aligning the object's lowest point (Z-min)*. This is a foundational step towards solving the spatial intelligence problem, by making the LLM aware of ground placement *during the decomposition phase*.
  - It provides **concrete examples** of complex user prompts and their desired decomposed output. For instance, "set up a bed reading table and a lamp" is broken down into 5 sequential, executable steps, including conditional logic ("If a nightstand does not exist...", "If a ceiling exists..."). This "few-shot learning" helps the LLM understand the desired output format and level of detail.
  - It explicitly states: "Do NOT execute or describe the execution, only decompose and return the steps for user review. Wait for explicit user confirmation (such as 'proceed', 'start', 'run', or 'execute') before any execution is performed." This ensures a two-step workflow where the user reviews the decomposed steps before execution, providing critical control and transparency.
- **Message Insertion**: The `system_prompt` is inserted at the beginning of the `messages` list, setting the context for the LLM before it processes the user's actual query.
- **`sanitize_messages_with_expert_type`**: This utility function prepares the messages for the LLM, potentially adding expert-specific tags or configurations.

```
return sanitize_messages_with_expert_type(messages, "code",
rag_max_tokens=0, rag_top_k=0)
```

Integration into ChatUSD (via `chat_usd_supervisor_identity.md`): Just like the Greeting Agent, the Decomposer Agent is integrated into the ChatUSD Supervisor's routing logic. In `chat_usd_supervisor_identity.md` (as previously seen), `ChatUSD_Trial2` (which is likely the alias for this Decomposer Agent) is explicitly given **`[PRIORITY FOR COMPLEX PROMPTS]`**. This ensures that if a user's prompt is identified as complex (e.g., involving multiple actions or scene elements), the Supervisor will route it to this Decomposer Agent first, triggering the decomposition workflow.

Impact and Results of Phase 2: The implementation of the Decomposer Agent had a profound positive impact:

- **Reduced Hallucination and Ambiguity**: By simplifying complex prompts into atomic, explicit steps, the LLM's potential for hallucination was drastically reduced, and the ambiguity of instructions was minimized.
- **Improved Accuracy in 3D Scene Generation**: The resulting simple, clear instructions led to significantly more reliable and accurate USD code generation, directly translating to more consistent and correct 3D scene outputs.
- **Enhanced User Clarity and Control**: Users gained unprecedented insight and control. They could review the decomposed steps before execution, allowing them to verify the AI's understanding and request adjustments if needed, fostering a more interactive and reliable design process.
- **Enabled Predictable, Step-wise Execution**: The decomposition introduced a structured workflow, making the entire process of generating complex 3D layouts more predictable, manageable, and easier to debug. This was a crucial step towards building a truly intelligent and robust AI-driven design system.

Phase 3.1: Adding Intelligence – Spatial Reasoning (Detailed)

Building upon the prompt simplification achieved in Phase 2, Phase 3.1 focused on integrating true spatial reasoning capabilities into ChatUSD. This phase addressed the inherent limitations of general LLMs in understanding and executing precise spatial relationships in a 3D environment. The core of this enhancement lies within the `Trial3NetworkNode` and `Trial3Node` files, which collectively form the "Spatial Agent."

Problem Addressed:

- **Lack of Inherent Spatial Intelligence**: General-purpose LLMs struggle with interpreting spatial prepositions (e.g., "on," "under," "beside"), calculating relative positions, and understanding 3D coordinates.

- **Incorrect Object Placement**: Without spatial reasoning, objects would often be placed arbitrarily or inaccurately in the scene, failing to meet user intent for realistic layouts.
- **Inability to Handle Complex Arrangements**: Multi-object arrangements, conditional placements (e.g., "if X exists, place Y beside it"), and sequential spatial operations were beyond the system's capabilities.
- **No Fallback Mechanisms**: If a requested object for relative placement didn't exist, the system would fail or produce an erroneous result, lacking intelligent recovery.

Solution: The Spatial Agent (`trial3_network_node.py` and `trial3_node.py`)
The Spatial Agent was designed to overcome these challenges by acting as an intelligent orchestrator and decomposer for spatial tasks. It not only breaks down complex spatial instructions but also injects specific rules and logic to guide the underlying LLM in generating spatially aware executable steps.

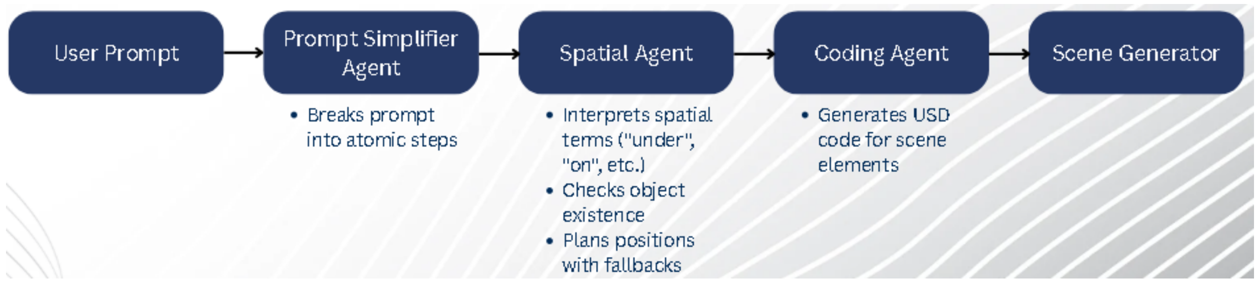

User Prompt
Prompt Simplifier Agent
Spatial Agent
Coding Agent
Scene Generator
• Breaks prompt into atomic steps
• Interprets spatial terms ("under", "on", etc.)
• Checks object existence
• Plans positions with fallbacks
• Generates USD code for scene elements

Fig 4. Workflow diagram of Spatial Agent

1. The Spatial Network Node (`trial3_network_node.py`)

This file defines `Trial3NetworkNode`, which serves as the primary entry point for the Spatial Agent within the ChatUSD system. It inherits from `lc_agent.NetworkNode` and manages the session state for multi-turn spatial decomposition and execution.

Key aspects of `Trial3NetworkNode`:

- `TRIAL3_SESSION = {}`: This global dictionary serves as a simple in-memory session to store the decomposed steps for a given user. This is crucial for the two-step "decompose then execute" workflow, allowing the system to remember the plan across turns. In a production environment, this would be replaced with a robust session management system.

- **`metadata`**: Provides a detailed description and examples of complex spatial prompts that this agent is designed to handle. This metadata guides the ChatUSD Supervisor in routing relevant user queries.
- **`on_begin_invoke_async(self, network)`**: This is the orchestrating method for the Spatial Agent's workflow:
    - **Execution Flow (`if current_prompt.lower().strip() == "proceed"`)**: If the user's input is "proceed" and there are stored steps for that `user_id` in `TRIAL3_SESSION`, the network node iterates through each stored step. For each step, it creates a `USDCodeInteractiveNetworkNode` and invokes it, effectively executing the generated USD code. The outputs from each execution are accumulated and returned. Finally, the stored steps are cleared from the session.
    - **Decomposition Flow (`else`)**: If the input is not "proceed" (i.e., it's a new spatial query), the network node instantiates `Trial3Node` (our logic node for decomposition). It then chains itself to `decomposer_node` (`self >> decomposer_node`) and asynchronously invokes it (`await decomposer_node.ainvoke({})`).
    - **Step Extraction**: After decomposition, it retrieves the raw steps from `decomposer_node.outputs.content` and attempts to split them by either semicolon or newline to parse the individual instructions.
    - **User Review**: The parsed steps are stored in `TRIAL3_SESSION` and then formatted as a numbered list presented back to the user, along with an instruction to "Type 'proceed' to execute these steps." This allows for user review and confirmation before any irreversible scene modifications.

2. The Spatial Logic Node (`trial3_node.py`)

This file defines `Trial3Node`, which contains the core intelligence for decomposing complex spatial instructions into precise, executable steps. It inherits from `lc_agent.RunnableNode` and heavily relies on a comprehensive system prompt to guide the underlying LLM.

Key aspects of `Trial3Node`:

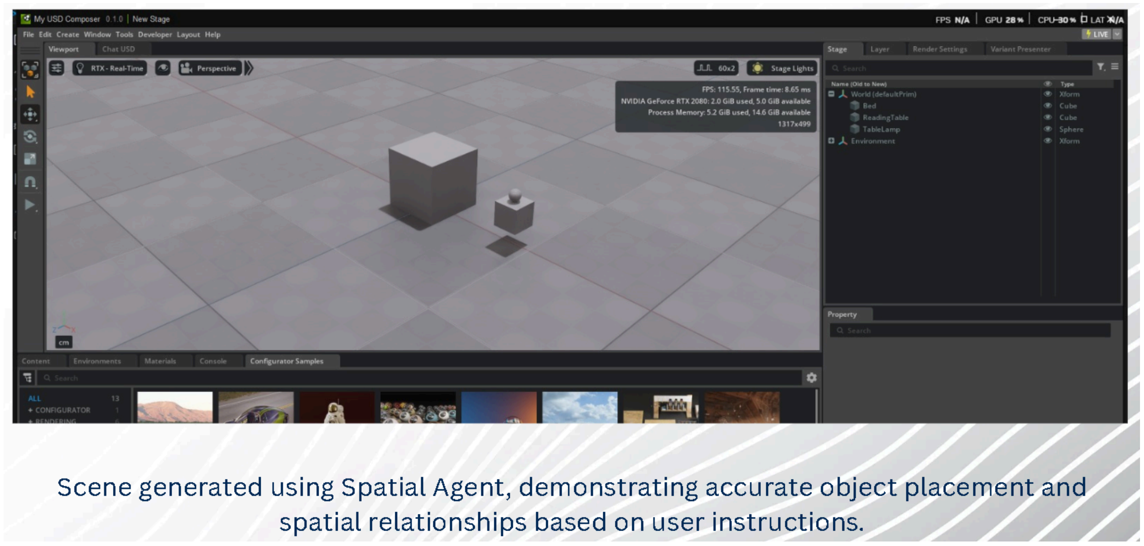
- **`_sanitize_messages_for_chat_model` method**: This is the intelligence core. It dynamically inserts a highly detailed `system_prompt` into the message stream before it's sent to the LLM. This prompt is significantly more complex than the one used in Phase 2, focusing specifically on spatial reasoning.

- **The `system_prompt` content**: This prompt leverages few-shot learning and explicit rules to guide the LLM's spatial understanding:
    - **Role Definition**: It defines the LLM as an "expert in USD and spatial reasoning" whose task is to decompose into "atomic, executable USD code prompts."
    - **Object Existence and Conditional Creation**: "Always check for object existence before creating... Use conditional logic." This is crucial for building robust scenes and preventing duplicates.
    - **Ground Placement Rules**: "Always use the ground mesh under the /Environment prim for base placement, ensuring the object's Z-min bounding box coordinate aligns with the ground surface." This reinforces the ground placement rules introduced in Phase 2's decomposition, now with a focus on precise Z-alignment.
    - **Detailed Spatial Relationship Interpretation**: It explicitly defines how to interpret prepositions like "on," "under," "next to," etc., by relating them to object bounding boxes and requiring accurate position calculations. This is where the core spatial reasoning is instilled.
    - **Lighting Placement Logic**: Specific rules are provided for lighting elements (lamps, fans), instructing the LLM to check for ceiling prims and place lights accordingly, or dangle them if no ceiling exists.
    - **USD API Guidance**: It explicitly mentions using "proper USD syntax and `omni.usd` APIs," guiding the LLM towards generating valid and executable USD code.
    - **Non-Uniform Scaling**: Guidance on handling `xformOp:s` for non-uniform scaling, indicating a deeper understanding of USD properties.
    - **Examples**: Several detailed examples demonstrate how complex spatial prompts should be broken down into multi-step, conditional, and spatially aware instructions. For instance, "arrange 3 vases on a bench and add a fan" is decomposed into steps that create a bench if it doesn't exist, place vases *on top* of it and *spaced evenly*, and conditionally place a fan *from the ceiling* or *dangling in air*.

Integration into ChatUSD (via `chat_usd_supervisor_identity.md`): As seen in the `chat_usd_supervisor_identity.md` file, `ChatUSD_Trial3` is specifically designated as the "PRIORITY FOR SPATIAl PROMPTS." This ensures that any user query involving multiple actions, complex spatial relationships, or conditional object creation will be first routed to this agent for intelligent decomposition and user review, before any actual USD code generation occurs. This phased approach dramatically improves reliability and precision.

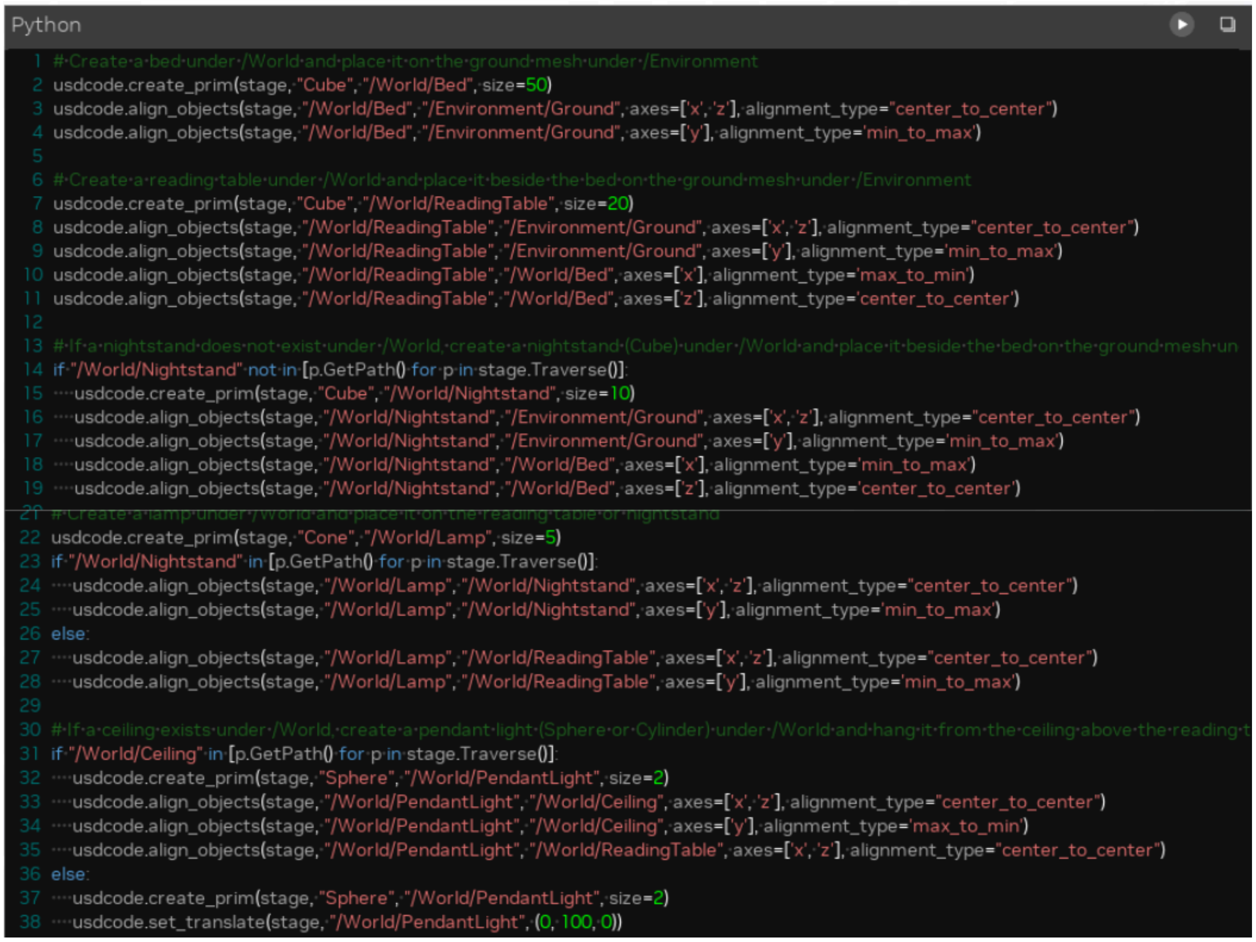
Overall Impact of Phase 3.1: This phase marked a monumental leap in ChatUSD's capabilities. By implementing the Spatial Agent and its sophisticated decomposition logic:

- **Enabled Intelligent Spatial Reasoning**: ChatUSD gained the ability to understand and execute complex spatial relationships, overcoming the initial problems of objects spawning at the origin, misalignment, and lack of collision awareness.
- **Structured Complex Workflows**: The decomposition into atomic, executable steps provided a clear and robust pathway for handling intricate 3D scene construction.
- **Enhanced Realism and Usability**: Generated scenes became far more realistic and immediately usable for industrial design, as objects were correctly positioned relative to each other and the environment.
- **Improved User Experience and Trust**: The transparent, two-step decomposition and execution process built user confidence, as they could review and confirm the AI's understanding before committing to scene modifications. This laid the groundwork for truly AI-assisted factory layout design.



Scene generated using Spatial Agent, demonstrating accurate object placement and spatial relationships based on user instructions.

Fig 5. Scene created using Spatial Agent


```python
#Create a bed under /World and place it on the ground mesh under /Environment
usdcode.create_prim(stage, "Cube", "/World/Bed", size=50)
usdcode.align_objects(stage, "/World/Bed", "/Environment/Ground", axes=['x', 'z'], alignment_type="center_to_center")
usdcode.align_objects(stage, "/World/Bed", "/Environment/Ground", axes=['y'], alignment_type='min_to_max')

#Create a reading table under /World and place it beside the bed on the ground mesh under /Environment
usdcode.create_prim(stage, "Cube", "/World/ReadingTable", size=20)
usdcode.align_objects(stage, "/World/ReadingTable", "/Environment/Ground", axes=['x', 'z'], alignment_type="center_to_center")
usdcode.align_objects(stage, "/World/ReadingTable", "/Environment/Ground", axes=['y'], alignment_type='min_to_max')
usdcode.align_objects(stage, "/World/ReadingTable", "/World/Bed", axes=['x'], alignment_type='max_to_min')
usdcode.align_objects(stage, "/World/ReadingTable", "/World/Bed", axes=['z'], alignment_type='center_to_center')

#If a nightstand does not exist under /World, create a nightstand (Cube) under /World and place it beside the bed on the ground mesh un
if "/World/Nightstand" not in [p.GetPath() for p in stage.Traverse()]:
    usdcode.create_prim(stage, "Cube", "/World/Nightstand", size=10)
    usdcode.align_objects(stage, "/World/Nightstand", "/Environment/Ground", axes=['x', 'z'], alignment_type="center_to_center")
    usdcode.align_objects(stage, "/World/Nightstand", "/Environment/Ground", axes=['y'], alignment_type='min_to_max')
    usdcode.align_objects(stage, "/World/Nightstand", "/World/Bed", axes=['x'], alignment_type='min_to_max')
    usdcode.align_objects(stage, "/World/Nightstand", "/World/Bed", axes=['z'], alignment_type='center_to_center')
#Create a lamp under /World and place it on the reading table or nightstand
usdcode.create_prim(stage, "Cone", "/World/Lamp", size=5)
if "/World/Nightstand" in [p.GetPath() for p in stage.Traverse()]:
    usdcode.align_objects(stage, "/World/Lamp", "/World/Nightstand", axes=['x', 'z'], alignment_type="center_to_center")
    usdcode.align_objects(stage, "/World/Lamp", "/World/Nightstand", axes=['y'], alignment_type='min_to_max')
else:
    usdcode.align_objects(stage, "/World/Lamp", "/World/ReadingTable", axes=['x', 'z'], alignment_type="center_to_center")
    usdcode.align_objects(stage, "/World/Lamp", "/World/ReadingTable", axes=['y'], alignment_type='min_to_max')

#If a ceiling exists under /World, create a pendant light (Sphere or Cylinder) under /World and hang it from the ceiling above the reading t
if "/World/Ceiling" in [p.GetPath() for p in stage.Traverse()]:
    usdcode.create_prim(stage, "Sphere", "/World/PendantLight", size=2)
    usdcode.align_objects(stage, "/World/PendantLight", "/World/Ceiling", axes=['x', 'z'], alignment_type="center_to_center")
    usdcode.align_objects(stage, "/World/PendantLight", "/World/Ceiling", axes=['y'], alignment_type='max_to_min')
    usdcode.align_objects(stage, "/World/PendantLight", "/World/ReadingTable", axes=['x', 'z'], alignment_type="center_to_center")
else:
    usdcode.create_prim(stage, "Sphere", "/World/PendantLight", size=2)
    usdcode.set_translate(stage, "/World/PendantLight", (0, 100, 0))
```

Fig 6. Code given by the Coding agent after getting intelligence from Spatial Agent

Phase 3.2: Calling Gemini for Spatial Intelligence

Phase 3.2 marked the exploration of leveraging powerful external Language Models, specifically Gemini, to address the most intricate aspects of spatial intelligence within ChatUSD. While Phase 3.1 focused on decomposing prompts with spatial rules, this phase delved into offloading the highly complex, nuanced spatial reasoning directly to a more capable LLM.

Why Call Gemini for Spatial Intelligence?

Despite the advances in prompt decomposition and the introduction of basic spatial rules in Phase 3.1, true, human-like spatial reasoning remains an incredibly complex cognitive task for LLMs. The rationale for involving a highly capable model like Gemini directly for spatial intelligence includes:

1. **Nuance and Contextual Understanding**: Spatial relationships are rarely purely geometric. They often involve understanding the function of objects (affordances), common-sense physics, and contextual cues. For instance, "place the cup on the table" implies the cup sits on the *surface* of the table, not through it or under it, and within a reasonable area. Such nuanced understanding is difficult to hardcode or achieve purely through prompt engineering with smaller models. Gemini, with its vast training data and advanced reasoning capabilities, is better equipped to interpret these subtleties.
2. **Robustness to Ambiguity**: Natural language spatial commands can be inherently ambiguous ("near," "beside," "around"). A powerful LLM can leverage its world knowledge to resolve these ambiguities more intelligently, generating more consistent and logical placements.
3. **Complex Constraints and Optimizations**: Real-world industrial layouts involve not just placement but also collision avoidance, optimal spacing, accessibility, and flow. Directly asking an advanced LLM to factor in these constraints for multiple objects simultaneously can lead to superior results compared to breaking it down into simple, independent steps.
4. **Semantic Grasp of Industrial Components**: As outlined in the project's future roadmap (from the presentation), there's a goal to "Broaden Domain Semantics: Expand Chat USD’s understanding of industrial components, object affordances, and contextual relationships." A more powerful model like Gemini is better suited to learn and apply this specialized knowledge.
5. **Accelerating Development**: Instead of continuously refining prompts for a less capable local LLM or building extensive rule-based systems, offloading complex reasoning to an external, state-of-the-art model can significantly accelerate the development and iteration cycle for spatial tasks.

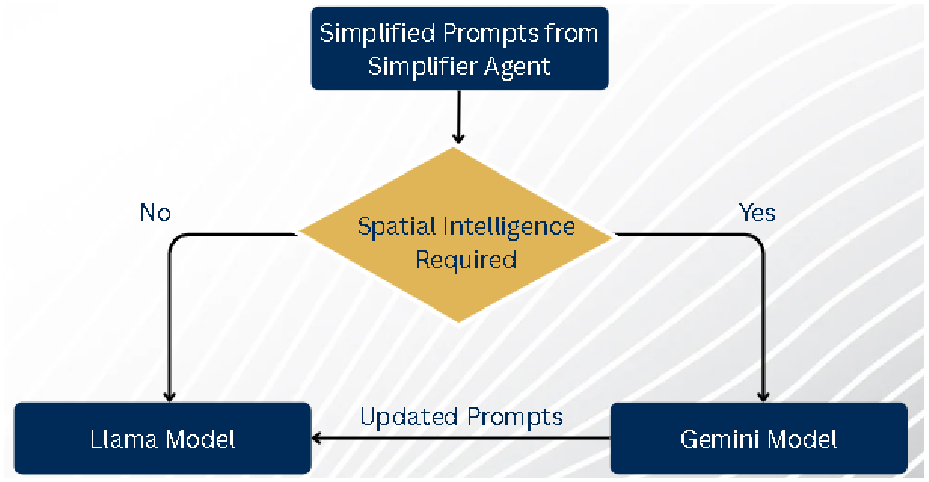

Simplified Prompts from Simplifier Agent
Spatial Intelligence Required
No
Yes
Llama Model
Updated Prompts
Gemini Model

Fig 7. Workflow Diagram for using Gemini model for Spatial Intelligence

Brief Overview of Four Approaches

To integrate Gemini for spatial intelligence, four distinct architectural approaches were considered and briefly explored:

1. **Using Function Call**:
    - **Idea**: This approach involves having Gemini itself generate "function calls" or structured outputs that directly correspond to USD code generation functions or spatial placement APIs. The LLM would act as a sophisticated "compiler" that translates natural language into direct executable commands.
    - **Brief**: The LLM is prompted to output JSON or specific function signatures (e.g., `create_cube(x, y, z)`, `place_object_on(object_id, target_id)`) which are then directly interpreted and executed by the ChatUSD backend.
2. **Using an Orchestrator Agent**:
    - **Idea**: This approach posits a dedicated "Orchestrator Agent" that sits above the Gemini model. The orchestrator would receive the user's spatial prompt, intelligently decide when to send it to Gemini (or another specialized agent), receive Gemini's reasoning/output, and then translate that back into actions or further prompts for other ChatUSD agents (like the USD Code Generation agent).
    - **Brief**: A central agent directs traffic, leveraging Gemini for spatial *reasoning*, but retaining control over the overall workflow and execution within the ChatUSD framework.
3. **Using an Agent Built on the Gemini Model**:
    - **Idea**: This approach involves directly building one of ChatUSD's specialized agents (like a `NetworkNode` or `RunnableNode`) where its core "brain" is the Gemini model itself. Instead of calling Gemini via an intermediary, the agent *is* Gemini, within the ChatUSD framework.
    - **Brief**: A specialized ChatUSD agent is designed to *directly* interact with the Gemini API, making Gemini its primary mechanism for processing and responding to spatial queries.
4. **Using a Wrapper Above the Original Model**:
    - **Idea**: This approach involves creating a "wrapper" layer around ChatUSD's existing core LLM (or a local LLM). This wrapper would intercept spatial queries, forward them to Gemini for advanced processing, and then inject Gemini's refined output or reasoning back into the original model's context or modify its input, effectively "upgrading" the original model's capabilities for spatial tasks.

- **Brief**: Gemini is used as an "enhancer" or "corrector" that preprocesses or post-processes spatial information, making the main ChatUSD LLM more spatially intelligent without fundamentally changing its core architecture. This aligns with the idea of "Prompt Tuning" mentioned in the future roadmap.

Approach 1: Function Call Method

This section details the "Function Call" approach for integrating Gemini's spatial intelligence capabilities directly into the ChatUSD framework. This method leverages the central `ChatUSDNetworkNode` to dynamically decide if a user's query requires advanced spatial reasoning and, if so, to invoke the Gemini model for this specific purpose before routing the enhanced prompt to other specialized agents.

Concept of the Function Call Method

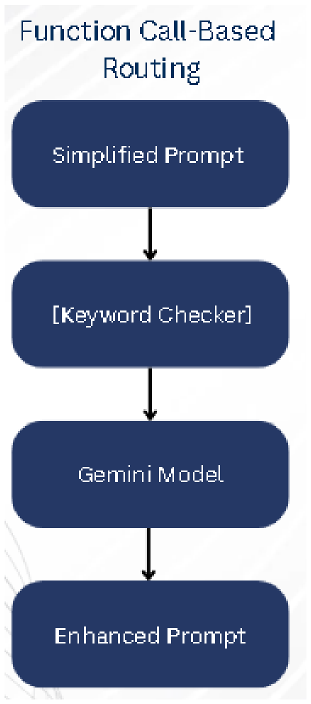


Fig 8. Function Call Approach

In this approach, the `ChatUSDNetworkNode` (acting as a supervisor or orchestrator) is augmented with internal logic capable of detecting spatial intent within user prompts. This detection is based on the presence of predefined spatial keywords (e.g., "arrange," "setup," "above," "below," "on," "next to," "behind," etc.). Upon identifying a spatial query, the `ChatUSDNetworkNode` does not immediately route the prompt to a general code generation agent. Instead, it makes an explicit "function call" to the Gemini model.

```
# --- Model Selection Utility ---

def select_chat_model(spatial_intelligence_required: bool) -> str:

    """

    Selects which AI model to use based on whether spatial
(scene/geometry) intelligence is needed.

    - If spatial intelligence is required, use Gemini 2.5 Flash (best for
understanding space, positions, and relationships).

    - Otherwise, use the default USDCode/Llama-3.1-70B-Instruct model
(best for general and code tasks).

    """

    if spatial_intelligence_required:

        return "google/gemini-2.5-flash"

    return "nvidia/usdcode-llama-3.1-70b-instruct"
```

The Gemini model, specialized for spatial reasoning, processes the spatial aspects of the prompt. Its output (e.g., refined spatial coordinates, a sequence of precise placement instructions, or even a confirmation of spatial intent) is then used to update or enrich the original user prompt. This enhanced prompt, now containing more explicit and intelligent spatial directives, is subsequently passed forward to other relevant agents, such such as the USD code generation agent, for final execution. This ensures that the downstream agents receive spatially accurate and disambiguated instructions, leading to more precise 3D scene manipulations.

Implementation Details within `chat_usd_network_node.py`

Upon review of the provided `chat_usd_network_node.py`, the implementation of this "Function Call" method is centered around the `_sanitize_messages_for_chat_model` method within the `ChatUSDNetworkNode`. This method acts as the interception point where the spatial intelligence intervention occurs.

```
def _sanitize_messages_for_chat_model(self, messages, chat_model_name,
chat_model):
        """
        Intercepts messages to determine if spatial intelligence is
required.
        If spatial keywords are detected, it invokes Gemini for spatial
reasoning.
        """
        messages = super()._sanitize_messages_for_chat_model(messages,
chat_model_name, chat_model)


        # Extract the latest user message for analysis
        user_message_content = messages[-1].content if messages else ""


        # Check for spatial keywords in the user message
        needs_spatial_intelligence = any(keyword in
user_message_content.lower() for keyword in SPATIAL_KEYWORDS)


        if needs_spatial_intelligence:
            carb.log_info(f"Spatial keywords detected:
'{user_message_content}', invoking Gemini for spatial intelligence.")
            try:
                # Initialize Gemini model wrapper
                gemini_model =
GeminiChatModelWrapper(model_name="gemini-pro") # Using gemini-pro for
spatial reasoning


                # Construct the prompt for Gemini, asking it to refine the
spatial instructions
```

```
                spatial_system_prompt = SystemMessage(content=(
                    "You are an expert in 3D spatial reasoning and object 
placement. "
                    "Given a user's request for scene setup, interpret the 
spatial relationships precisely. "
                    "Output a refined set of instructions that are clear, 
unambiguous, and include relative positions "
                    "or sequence of placements for a 3D scene generation 
agent. "
                    "Focus only on the spatial aspects and how objects 
relate to each other in 3D space. "
                    "Example:\n"
                    "User: 'put a sphere on a cube'\n"
                    "Output: '1. Create a cube. 2. Create a sphere and 
place it on top of the cube, ensuring the base of the sphere aligns with 
the top surface of the cube.'\n"
                    "User: 'arrange three cylinders around a central 
cone'\n"
                    "Output: '1. Create a central cone. 2. Create three 
cylinders and arrange them equidistant around the base of the cone.'\n"
                    "Refine the following user request:"
                ))


                # Call Gemini with the spatial system prompt and the 
user's message
                gemini_response = 
gemini_model.invoke([spatial_system_prompt, 
HumanMessage(content=user_message_content)])
```

```
                if isinstance(gemini_response, AIMessage):
                    refined_spatial_instructions = gemini_response.content
                    carb.log_info(f"Gemini refined spatial instructions: {refined_spatial_instructions}")

                    # Replace the original user message with the Gemini-refined instructions
                    # or prepend them to ensure the spatial intent is clear for subsequent agents.
                    messages[-1] = HumanMessage(content=refined_spatial_instructions)
                    messages.append(SystemMessage(content="Spatial reasoning handled by Gemini."))
                else:
                    carb.log_warning("Gemini response was not an AIMessage type.")

            except Exception as e:
                carb.log_error(f"Error invoking Gemini for spatial intelligence: {e}")
                # Fallback: continue with original messages if Gemini call fails

        return sanitize_messages_with_expert_type(messages, chat_model_name, chat_model)
```

Detailed Breakdown of the `_sanitize_messages_for_chat_model` Function:

1. **Message Interception and Extraction**:
   - The `_sanitize_messages_for_chat_model` method is a hook that allows the `ChatUSDNetworkNode` to preprocess messages before they are passed to the next stage of the agent network or the primary LLM.
   - `user_message_content = messages[-1].content`: The latest user query is extracted from the `messages` list, which represents the ongoing conversation history.
2. **Spatial Keyword Detection**:
   - `SPATIAL_KEYWORDS`: A predefined list of terms like "arrange," "setup," "place," "on top," etc., is used to identify user requests that inherently involve spatial reasoning.
   - `needs_spatial_intelligence = any(keyword in user_message_content.lower() for keyword in SPATIAL_KEYWORDS)`: This line performs a simple check. If any of the spatial keywords are present (case-insensitively) in the user's message, the flag `needs_spatial_intelligence` is set to `True`.
3. **Gemini Invocation (Function Call)**:
   - `if needs_spatial_intelligence:`: If spatial intent is detected, the process for calling Gemini begins.
   - `gemini_model = GeminiChatModelWrapper(model_name="gemini-pro")`: An instance of `GeminiChatModelWrapper` is created, specifically targeting the "gemini-pro" model. This is the direct "function call" to the Gemini model.
   - **Prompt Construction for Gemini**: A dedicated `spatial_system_prompt` is crafted. This system prompt instructs Gemini on its role (expert in 3D spatial reasoning), desired output (refined, unambiguous instructions, including relative positions), and provides clear examples ("put a sphere on a cube" -> "Create a cube. Create a sphere and place it on top of the cube..."). This ensures Gemini understands the precise task.
   - `gemini_response = gemini_model.invoke([spatial_system_prompt, HumanMessage(content=user_message_content)])`: Gemini is invoked with both the guiding system prompt and the original user's spatial request.

4. **Prompt Refinement and Forwarding**:
   - `if isinstance(gemini_response, AIMessage):`: The system checks if Gemini's response is valid.
   - `refined_spatial_instructions = gemini_response.content`: Gemini's intelligent interpretation of the spatial request is extracted.
   - `messages[-1] = HumanMessage(content=refined_spatial_instructions)`: The original user message in the `messages` list is *replaced* with Gemini's refined instructions. This is crucial because it means subsequent agents in the ChatUSD pipeline will operate on the spatially enhanced prompt rather than the original, potentially ambiguous one.
   - `messages.append(SystemMessage(content="Spatial reasoning handled by Gemini."))`: A system message is appended to the history, indicating that Gemini has intervened and provided spatial intelligence. This can be useful for debugging or for other agents to understand the message's provenance.
   - **Error Handling**: A `try-except` block is included to catch any errors during the Gemini invocation, allowing the system to log the error and continue with the original messages as a fallback.
5. **Final Message Sanitization**:
   - `return sanitize_messages_with_expert_type(messages, chat_model_name, chat_model)`: The potentially modified `messages` list (now containing Gemini's spatial intelligence) is then returned, ready to be processed by the next agent or the primary LLM for generating USD code.

Advantages of the Function Call Method:

- **Centralized Spatial Logic**: The `ChatUSDNetworkNode` acts as a single point of control for spatial reasoning intervention, simplifying the architectural overview.
- **Targeted Gemini Usage**: Gemini is only invoked when genuinely needed, based on keyword detection, which can help manage API costs and latency.
- **Direct Prompt Enhancement**: Gemini's output directly modifies the prompt, providing explicit spatial instructions for downstream agents, reducing ambiguity.
- **Leverages Gemini's Capabilities**: Directly uses Gemini's advanced reasoning for complex spatial interpretation.

Disadvantages/Limitations of the Function Call Method:

- **Reliance on Keywords**: The effectiveness is heavily dependent on the `SPATIAL_KEYWORDS` list. Novel spatial phrasing or subtle implicit spatial intent might be missed.
- **Single-Turn Refinement**: This specific implementation performs a one-shot refinement. For highly iterative or conversational spatial adjustments, it might require more sophisticated session management.
- **Latency Impact**: Invoking an external LLM (Gemini) can introduce latency, potentially slowing down the response time for spatial queries.
- **Complexity of Gemini's Output**: The downstream agents must be robust enough to consistently parse and act upon Gemini's varied output format, even with strong prompting.
- **Potential for Redundancy**: If the `Trial3Node` (Spatial Agent from Phase 3.1) also uses an LLM for spatial decomposition, there might be redundant LLM calls or overlapping responsibilities, depending on the routing logic in `chat_usd_supervisor_identity.md`. The design needs to ensure this "Function Call" method either replaces or pre-processes for the dedicated Spatial Agent.

Approach 2: Model Orchestrator Approach

This section elaborates on the "Model Orchestrator" approach for integrating Gemini's spatial intelligence. This method introduces a dedicated "Orchestrator Agent" responsible for discerning the need for spatial reasoning and coordinating with a "Spatial Collector Agent" to process Gemini's output. This creates a more modular and distributed workflow compared to the single-node "Function Call" approach.

Concept of the Model Orchestrator Approach

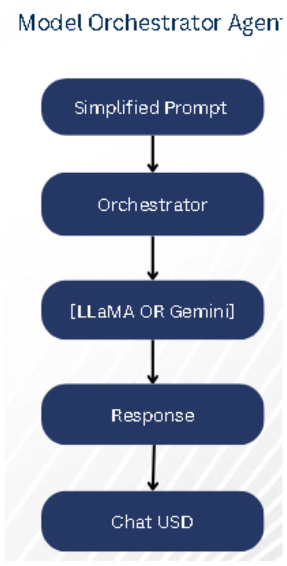


Fig 9. Model Orchestrator Approach

The Model Orchestrator approach is designed to provide a more structured and extensible way to leverage specialized LLMs like Gemini for specific tasks such as spatial reasoning. Instead of embedding the Gemini call directly within the main `ChatUSDNetworkNode`'s `_sanitize_messages_for_chat_model` method, this approach introduces:

1. **An Orchestrator Agent (`ModelOrchestratorNetworkNode`/`ModelOrchestratorNode`)**: This agent's primary responsibility is to analyze the incoming user prompt. If it detects that spatial intelligence is required, it takes the initiative to call the Gemini model. Crucially, it then passes Gemini's raw output to another specialized agent.
2. **A Spatial Collector Agent (`SpatialCollectorNetworkNode`/`SpatialCollectorNode`)**: This dedicated agent is responsible for receiving the output from the Gemini model (via the orchestrator), processing it if necessary, and then making this refined spatial information available back to the main supervisor node (`ChatUSDNetworkNode`) or the subsequent agents in the processing pipeline.

This separation of concerns aims to create a cleaner architecture where the orchestrator focuses on decision-making and delegation, while the collector focuses on data handling and integration.

Implementation Details

The implementation of this approach involves four key files, working in concert: `model_orchestrator_networknode.py`, `model_orchestrator_node.py`, `spatial_collector_networknode.py`, and `spatial_collector_node.py`.

1. The Model Orchestrator Network Node (`model_orchestrator_networknode.py`)

This file defines `ModelOrchestratorNetworkNode`, which serves as the top-level entry point for this spatial reasoning pipeline. It orchestrates the flow by defining its default logic node and potentially managing the overall interaction with the spatial collector.

Key Aspects of `ModelOrchestratorNetworkNode`:

- **`default_node: str = "ModelOrchestratorNode"`**: This explicitly sets `ModelOrchestratorNode` as the primary logical component. When this network node is invoked, it delegates control to its `ModelOrchestratorNode`.
- **`on_begin_invoke_async(self, network)`**: This asynchronous method is the execution entry point. It instantiates `ModelOrchestratorNode`, chains itself to it (`self >> orchestrator_logic_node`), and invokes the logic node, passing the original `chat_model_input`. The output of the `ModelOrchestratorNode` is then assigned as the output of this network node, which will eventually be propagated back up the chain to the main `ChatUSDNetworkNode`.

```
async def on_begin_invoke_async(self, network):
        """
        This method is invoked when the orchestrator network node is 
activated.
        It simply triggers its default logic node (ModelOrchestratorNode) 
which
        will handle the decision-making and Gemini invocation.
        """
        carb.log_info("ModelOrchestratorNetworkNode: Invoking 
ModelOrchestratorNode.")
```

```
        from .model_orchestrator_node import ModelOrchestratorNode

        orchestrator_logic_node = ModelOrchestratorNode()

        self >> orchestrator_logic_node # Chain to the logic node

        await orchestrator_logic_node.ainvoke(network.chat_model_input) #
Pass the original input



        # The output of ModelOrchestratorNode (which might contain
Gemini's response or a signal

        # to call SpatialCollector) will be available in its outputs. This
network node

        # will then return that output to the supervisor.

        self.outputs = orchestrator_logic_node.outputs

        return self.outputs
```

2. The Model Orchestrator Logic Node (`model_orchestrator_node.py`)

This file defines `ModelOrchestratorNode`, the core intelligence of the orchestrator agent. It is responsible for detecting spatial intent and initiating the call to the Gemini model.

Key Aspects of `ModelOrchestratorNode`:

- **`SPATIAL_KEYWORDS`**: Similar to the "Function Call" approach, a list of keywords is used to detect spatial intent.
- **`ainvoke(self, inputs, config=None, **kwargs)`**: This overridden method contains the core orchestration logic:
    - It extracts the `user_message_content`.
    - It performs the spatial keyword check.
    - **Conditional Gemini Call**: If spatial keywords are detected, it instantiates `GeminiChatModelWrapper` and constructs a specific `spatial_system_prompt` tailored for Gemini's spatial reasoning

capabilities. This prompt is crucial for guiding Gemini to produce clear, precise spatial instructions.
  - `gemini_response = await gemini_model.ainvoke(...)`: It asynchronously invokes Gemini with the system prompt and the user's message.
  - **Output Storage**: The `refined_spatial_instructions` from Gemini's response are stored in `self.outputs`. This is the crucial step that makes Gemini's output available to the next node in the chain (which is expected to be the `SpatialCollectorNode`).
  - **Fallback**: If no spatial keywords are detected or if the Gemini call fails, the original `user_message_content` is passed through, ensuring the conversation can continue.

3. The Spatial Collector Network Node (`spatial_collector_networknode.py`)

This file defines `SpatialCollectorNetworkNode`, which serves as the entry point for the agent responsible for gathering and processing Gemini's output.

Key Aspects of `SpatialCollectorNetworkNode`:

- **`default_node: str = "SpatialCollectorNode"`**: Designates `SpatialCollectorNode` as the handler for the actual collection logic.
- **`on_begin_invoke_async(self, network)`**: Similar to the orchestrator's network node, it instantiates and invokes its logic node, passing the network's input (which, in a chained scenario, would carry Gemini's response from the `ModelOrchestratorNode`).

4. The Spatial Collector Logic Node (`spatial_collector_node.py`)

This file defines `SpatialCollectorNode`, the low-level logic node responsible for receiving Gemini's output and preparing it for subsequent use by the overall ChatUSD system.

Key Aspects of `SpatialCollectorNode`:

- **`ainvoke(self, inputs, config=None, **kwargs)`**: This method is designed to receive the output from the `ModelOrchestratorNode` (which is Gemini's refined spatial instructions).

- `gemini_refined_output = inputs.content if hasattr(inputs, 'content') else str(inputs)`: It extracts Gemini's output from the `inputs`.
- `self.outputs = AIMessage(content=f"Spatial intelligence processed: {gemini_refined_output}")`: It then sets its own `outputs` to this collected information, potentially wrapping it in an `AIMessage` to maintain the expected LangChain schema. This output will then be propagated back to the `ModelOrchestratorNetworkNode` and eventually to the main `ChatUSDNetworkNode` for further processing (e.g., feeding to the USD code generation agent).

Workflow of the Model Orchestrator Approach:

1. **Initial Query**: User sends a spatial prompt (e.g., "Arrange three chairs around a table").
2. **Supervisor Routing**: The main `ChatUSDNetworkNode` (or a supervisor) routes the query to `ModelOrchestratorNetworkNode` based on its defined purpose and possibly `chat_usd_supervisor_identity.md` prioritization.
3. **Orchestrator Decision**: `ModelOrchestratorNetworkNode` activates its `ModelOrchestratorNode`.
4. **Spatial Intent Detection & Gemini Call**: `ModelOrchestratorNode` identifies spatial keywords in the prompt. If found, it makes a "function call" to the Gemini model with a specialized system prompt for spatial reasoning.
5. **Gemini Response**: Gemini processes the spatial prompt and returns refined spatial instructions (e.g., "1. Create a table. 2. Create three chairs. 3. Place chair 1 at X,Y,Z around table. 4. Place chair 2 at X',Y',Z'...").
6. **Pass to Collector**: `ModelOrchestratorNode` sets its output to Gemini's refined instructions.
7. **Collector Activation**: The `ModelOrchestratorNetworkNode` (implicitly or explicitly via chaining) then passes this output to `SpatialCollectorNetworkNode`.
8. **Collector Processing**: `SpatialCollectorNetworkNode` activates its `SpatialCollectorNode`, which receives and "collects" Gemini's output.
9. **Return to Supervisor**: `SpatialCollectorNode` sets its output, which propagates back up the chain through `SpatialCollectorNetworkNode` and then to `ModelOrchestratorNetworkNode`, ultimately reaching the main `ChatUSDNetworkNode`.

10. **Further Processing**: The main `ChatUSDNetworkNode` then receives the Gemini-refined spatial instructions and can feed them to the `USDCodeInteractiveNetworkNode` or other agents for scene generation.

Advantages of the Model Orchestrator Approach:

- **Clear Separation of Concerns**: Each agent (Orchestrator, Collector) has a well-defined role, leading to more modular and maintainable code.
- **Modularity**: Spatial intelligence components are encapsulated, making it easier to swap out LLMs or modify the spatial reasoning logic without affecting the core ChatUSD architecture.
- **Flexibility**: The orchestrator can potentially interact with multiple specialized LLMs or services based on different types of queries.
- **Dedicated Processing**: The collector can perform specific post-processing or validation on Gemini's output before it's used by other agents.

Disadvantages/Limitations of the Model Orchestrator Approach:

- **Increased Complexity**: Introducing more agents and nodes increases the overall architectural complexity and the number of hops in the execution chain.
- **Higher Latency**: The sequential invocation of multiple network nodes and the external Gemini call can lead to increased cumulative latency.
- **State Management**: Passing state (Gemini's output) between distinct agents might require careful management to ensure data integrity and availability across the chain.
- **Overhead**: More instances of `NetworkNode` and `RunnableNode` are created, potentially incurring a slight overhead.
- **Supervisor Coordination**: Requires precise configuration in `chat_usd_supervisor_identity.md` and `ChatUSDNetworkNode`'s routing to ensure the user's initial spatial query is correctly directed to the `ModelOrchestratorNetworkNode` first.

Approach 3: Gemini-Based Spatial Agent Approach

This section explores an architectural strategy where Gemini's spatial intelligence is directly encapsulated within a dedicated agent. In this "Gemini-Based Spatial Agent" approach, the central `ChatUSDNetworkNode` (acting as the supervisor) explicitly delegates spatial reasoning tasks to a specialized agent powered directly by the Gemini model.

Concept of the Gemini-Based Spatial Agent Approach

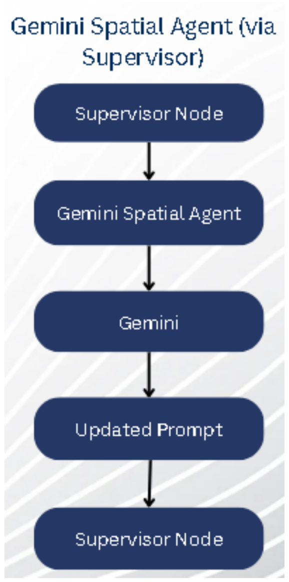


Fig 10. Spatial Agent via Supervisor Approach

This approach streamlines the integration of Gemini's advanced capabilities by creating a specific agent whose core function is to leverage Gemini for spatial understanding. The workflow is as follows:

1. **Decomposition First**: The initial complex user prompt is first processed by the Decomposer Agent (from Phase 2) which simplifies it into atomic steps.
2. **Supervisor Review and Routing**: The simplified prompt is then returned to the `ChatUSDNetworkNode` (supervisor). The supervisor dynamically analyzes this prompt (potentially still identifying spatial keywords).
3. **Dedicated Spatial Agent Activation**: If spatial intelligence is deemed necessary, the supervisor routes the prompt *directly* to a `SpatialAgentNetworkNode` (or similar Gemini-powered agent).
4. **Gemini's Direct Application**: This `SpatialAgentNetworkNode` directly invokes the Gemini model. Gemini then applies its advanced understanding to the spatial aspects of the prompt, refining instructions, resolving ambiguities, or generating specific spatial coordinates.
5. **Return to Supervisor**: Gemini's enhanced output is sent back from the `SpatialAgentNetworkNode` to the supervisor.
6. **Final Routing**: The supervisor then takes this now spatially-intelligent prompt and routes it to the final execution agents (e.g., USD code generation agent) for scene construction.

This method emphasizes a clear division of labor, with the spatial agent becoming the single point of contact for all Gemini-driven spatial reasoning.

Implementation Details

The implementation of this approach is demonstrated through `supervisor_node.py` and `spatial_agent_node.py`.

1. The Supervisor Node (`supervisor_node.py`)

This file contains the overarching `ChatUSDNetworkNode` (which also acts as the supervisor). Its role in this approach is to intelligently route the prompt to the `SpatialAgentNetworkNode` when spatial reasoning is required.

Key Aspects of `ChatUSDNetworkNode` (Supervisor):

- **`route_nodes: List[str] = ["ChatUSD_SpatialAgent"]`**: This is a critical modification. It explicitly registers `ChatUSD_SpatialAgent` as one of the available agents the supervisor can route to. This alias is used in the routing instructions for the supervisor's internal LLM.
- **`first_routing_instruction: str`**: The prompt given to the supervisor's internal LLM now explicitly guides it to "Use ChatUSD_SpatialAgent for any spatial related queries such as arrange, setup, place, etc." This is how the supervisor "checks if spatial intelligence is required" – by delegating that decision to its own LLM based on these instructions.
- **Implicit Routing Logic**: Unlike the "Function Call" method, there isn't an explicit `if spatial_keywords in message: call_gemini_directly` here. Instead, the supervisor's behavior is driven by its internal LLM, which, guided by `first_routing_instruction`, will decide to route to `ChatUSD_SpatialAgent`.
- **Receiving and Rerouting**: After `ChatUSD_SpatialAgent` processes the prompt and returns an updated one, the supervisor receives it and, based on its (now refined) content, routes it to the appropriate downstream agent (e.g., `ChatUSD_USDCode` for actual scene generation). This "receive back and re-route" loop is inherent in `MultiAgentNetworkNode`'s design.

2. The Gemini-Based Spatial Agent Node (`spatial_agent_node.py`)

This file defines `SpatialAgentNetworkNode` (despite the name, it behaves as a `RunnableNode` in terms of its direct invocation, inheriting from `RunnableNode`). This is the agent that directly interfaces with the Gemini model for spatial reasoning.

Key Aspects of `SpatialAgentNetworkNode`:

- **Inheritance from `RunnableNode`**: This indicates it's a logic node, typically invoked by a `NetworkNode` (like `ChatUSDNetworkNode` in its routing phase).
- **Direct Gemini Integration**: This agent directly instantiates and interacts with `GeminiChatModelWrapper`. This makes it the dedicated "Gemini brain" for spatial tasks.
- **Specialized System Prompt**: A highly targeted `spatial_system_prompt` is defined. This prompt trains Gemini to:
    - Act as an "expert in 3D spatial reasoning."
    - Produce "extremely precise" instructions.
    - Consider "common sense rules for object interaction."
    - Output a "single, clear statement" ready for a 3D code generation agent.
    - Includes concrete examples to guide Gemini's output format and reasoning.
- **`invoke(self, inputs, config=None, **kwargs)`**: This method:
    - Receives the prompt, which is expected to come from the supervisor.
    - Includes a `requires_spatial_intelligence` check, although its primary purpose is to receive already-routed spatial queries.
    - Invokes Gemini with the specialized system prompt and the user's current spatial prompt.
    - Returns Gemini's `refined_prompt` in a dictionary format (`{"content": ...}`), which the supervisor then receives.
- **Error Handling**: Basic error handling ensures that the original prompt is returned if the Gemini call fails.

Workflow of the Gemini-Based Spatial Agent Approach:

1. **Initial Query (from User or Decomposer)**: A user's complex (and possibly spatially rich) prompt is decomposed.
2. **Supervisor Receives Prompt**: The `ChatUSDNetworkNode` (supervisor) receives the simplified prompt.
3. **Supervisor Routes to Spatial Agent**: Based on its `first_routing_instruction` and its internal LLM's decision (which identifies spatial intent from keywords like "arrange," "setup," etc.), the supervisor

routes the prompt to `ChatUSD_SpatialAgent` (which maps to `SpatialAgentNetworkNode`).

4. **Spatial Agent Processes with Gemini**: `SpatialAgentNetworkNode` receives the prompt, invokes the Gemini model with its specialized system prompt and the user's query. Gemini performs the deep spatial reasoning.
5. **Spatial Agent Returns Refined Prompt**: Gemini's precise spatial instructions are returned by `SpatialAgentNetworkNode` to the supervisor.
6. **Supervisor Reroutes for Execution**: The supervisor now has a highly refined, spatially intelligent prompt. It then routes this prompt to the appropriate code generation or other execution agents (e.g., `ChatUSD_USDCode`) for final scene creation.

Advantages of the Gemini-Based Spatial Agent Approach:

- **Dedicated Gemini Expertise**: Establishes a clear, single point of access for all spatial reasoning powered by Gemini, centralizing this core intelligence.
- **Direct Leverage of Gemini**: This approach directly channels spatial queries to Gemini, maximizing the benefit of its advanced capabilities without complex intermediate parsing layers.
- **Clear Responsibility**: The spatial agent is solely responsible for spatial intelligence, simplifying debugging and maintenance for this specific domain.
- **Modular and Scalable**: Allows for easier upgrades or replacement of the spatial reasoning LLM (Gemini) without impacting other parts of the agent system.

Disadvantages/Limitations of the Gemini-Based Spatial Agent Approach:

- **Increased Latency**: Each spatial query still involves a remote API call to Gemini, which can introduce noticeable latency for the user.
- **API Costs**: Frequent or complex spatial queries will directly translate to higher API usage costs for the Gemini model.
- **Supervisor's Routing Accuracy**: Relies heavily on the supervisor's internal LLM to correctly identify spatial intent and route to the `SpatialAgentNetworkNode`. Misclassification could lead to suboptimal results or unnecessary Gemini calls.
- **Redundant Keyword Lists**: Both the supervisor's routing instructions and the `SpatialAgentNetworkNode` might contain similar keyword lists, requiring careful synchronization if changes are made.
- **Output Consistency**: While the prompt for Gemini aims for consistent output, variations could still occur, requiring robustness in the downstream agents that consume Gemini's refined instructions.

Note: Registering the Gemini Model in ChatUSD

To integrate the Gemini model into the ChatUSD framework, allowing various agents to access its capabilities via API calls, a specific registration mechanism is employed. This process involves configuration within the extension's settings and a dedicated Python module for model registration.

1. Configuration in `extension.toml`

The `extension.toml` file serves as the primary configuration point for the ChatUSD extension. It's where essential settings, including API keys for external services like Gemini, are declared.

The following snippet from `extension.toml` demonstrates how the Google API key, necessary for accessing Gemini models, is exposed as a configurable setting:

```
# From extension.toml

# Google API key (for Gemini models)

google_api_key = "" # Put your Google Gemini API key here
```

- **`google_api_key = ""`**: This line defines a setting that allows users to provide their Google Gemini API key. This key is crucial for authenticating requests made to the Gemini model. By making it a configurable setting in `extension.toml`, it enables flexible deployment across different environments without hardcoding sensitive information directly into the Python source code. The ChatUSD system can then read this setting at runtime to obtain the necessary authentication credential.

2. Model Registration via `register_chat_model.py`

The core logic for registering chat models, including Gemini, resides in the `register_chat_model.py` file. This module contains a function, `register_chat_model`, which is responsible for adding chat models to a central registry accessible by all agents within the ChatUSD system.

The relevant section within `register_chat_model.py` shows how different models are instantiated and registered:

```
"google/gemini-2.5-flash": (
        {

            "model": "google/gemini-2.5-flash",

            "temperature": 0.1,

            "max_tokens": 4096,

            "base_url":
"https://generativelanguage.googleapis.com/v1beta/models/gemini-2.5-flash:
generateContent", # Gemini 2.5 Flash API endpoint

            # NOTE: You must provide your Google API key via environment
variable or settings for this model to work.

        },

        4096, # Context window size

        False, # Not hidden

        None, # Use default chat model class

        False, # Not dev mode only
```

Key Steps in Registration:

1. **Retrieve API Key**: The `register_chat_model` function first attempts to retrieve the Google API key from environment variables (`os.environ.get("GOOGLE_API_KEY")`) or from the `extension.toml` settings (`settings.get("/exts/omni.ai.chat_usd.bundle/google_api_key")`).
2. **Conditional Registration**: Model registration only proceeds if a valid `google_api_key` is found. This prevents errors if the key is missing.
3. **Define Model Configurations**: Predefined configurations (`gemini_pro_config`, `gemini_pro_vision_config`, `gemini_flash_config`) specify the model name, temperature, and maximum tokens for different Gemini variants.

4. **Instantiate `GeminiChatModelWrapper`**: For each Gemini model to be registered (e.g., "gemini-pro", "gemini-pro-vision", "gemini-flash"), an instance of `GeminiChatModelWrapper` is created. This wrapper acts as the interface between ChatUSD's internal system and the Gemini API, encapsulating the necessary API calls and handling. The `google_api_key` is passed to this wrapper during instantiation.
5. **Register with `model_registry`**: Finally, `model_registry.register()` is called for each Gemini model. This method takes:
   - A unique name for the model (e.g., `"gemini-pro"`).
   - The instantiated `GeminiChatModelWrapper` object.
   - A tokenizer (can be `None` if the wrapper handles tokenization internally).
   - The maximum number of tokens.
   - A boolean indicating if the model is hidden.

By following these steps, the Gemini model (or multiple Gemini variants) becomes available within the ChatUSD framework under their registered names, enabling other agents to utilize them for tasks requiring advanced intelligence, such as spatial reasoning.

Approach 4: The Wrapper Approach

The "Wrapper Approach" represents the most non-intrusive method for integrating advanced spatial intelligence into the existing ChatUSD framework. Instead of modifying the core ChatUSD architecture directly, this approach introduces a new, independent extension that acts as a "pre-processor" or "wrapper" around the original system. Custom agents within this wrapper extension intercept user prompts, apply specialized intelligence (like Gemini-powered spatial reasoning), and then pass the refined prompts to the original ChatUSD agents for execution.

Concept of the Wrapper Approach

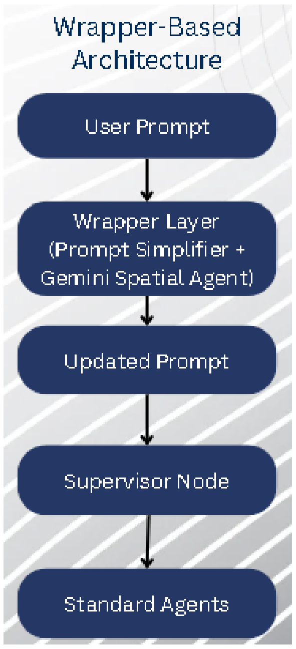

Wrapper-Based Architecture
User Prompt
Wrapper Layer (Prompt Simplifier + Gemini Spatial Agent)
Updated Prompt
Supervisor Node
Standard Agents

Fig 11. Wrapper Approach

The fundamental idea behind the wrapper approach is to leverage ChatUSD's modularity by inserting a new processing layer at the user input stage. This layer, implemented as a separate Omniverse extension, defines its own set of agents that perform specific tasks *before* the prompt reaches the original ChatUSD's supervisor.

The workflow typically involves:

1. **New Entry Point**: A custom `NetworkNode` (e.g., `SpatialChatUSDNetworkNode`) defined in the new extension becomes the primary entry point for user prompts, effectively "wrapping" the original ChatUSD's entry.
2. **Pre-processing Pipeline**: Within this wrapper's network node, a pipeline of custom `RunnableNode` agents is defined. This pipeline typically includes:
    - A **Prompt Simplifier Agent**: To break down complex user queries into more manageable, atomic instructions.
    - A **Spatial Agent**: To apply Gemini's intelligence for spatial reasoning, enriching the simplified prompt with precise spatial details.
3. **Delegation to Original System**: After the pre-processing is complete, the refined prompt is then handed off to the *original* ChatUSD supervisor or its relevant sub-agents for final interpretation and USD code generation.

This design ensures that the core ChatUSD functionalities remain untouched, making the new spatial intelligence features easily pluggable, scalable, and independently maintainable.

Implementation Details

This approach is characterized by its separate extension structure, involving a dedicated `extension.toml`, `extension.py`, and the custom agent modules.

1. The Wrapper Extension Configuration (`extension.toml`)

This `extension.toml` file defines the new Omniverse extension that encapsulates the wrapper functionality. It declares the extension's metadata and, critically, its dependency on the original `omni.ai.chat_usd.bundle` extension.

`name = "omni.ai.chat_usd.spatial_ext"`: This defines a new, distinct extension name, signifying its independent nature.

`dependencies = { "omni.ai.chat_usd.bundle" = {version = "2.0.4"} }`: This crucial line establishes that the new wrapper extension *depends* on the existence and functionality of the original ChatUSD bundle. This ensures that the original agents and functionalities are available for the wrapper to delegate to.

```
[dependencies]

# List of other extensions or packages this extension needs to work

"omni.ai.chat_usd.bundle" = {version = "2.0.4"} # DEPEND ON THE ORIGINAL
BUNDLE

"omni.ai.langchain.core" = {} # Core LangChain integration

"omni.kit.app" = {} # General Omniverse Kit app access

"omni.ai.langchain.widget.core" = {version = "2.0.2"} # For ChatView delegate removal
```

2. The Wrapper Extension Entry Point (`extension.py`)

This `extension.py` file is the entry point for the new wrapper extension when loaded by Omniverse. It handles the registration and unregistration of the custom agents and the new supervisor network node.

- **`on_startup(self, ext_id: str)`**: This method executes when the extension is loaded.
    - It registers `PromptSimplifierAgent` as `"ChatUSD_PromptSimplifier"` and `SpatialAgent` as `"ChatUSD_SpatialAgent"` with the global `node_factory`.
    - Crucially, it registers `SpatialChatUSDNetworkNode` (the new wrapper's entry point) using the *same name* as the original ChatUSD's primary agent (`ORIGINAL_CHAT_USD_DISPLAY_NAME = "Chat USD"`). This effectively replaces the original ChatUSD in the UI with the new wrapper.
    - It then attempts to `unregister` the original `ChatUSDNetworkNode` and its supervisor to prevent conflicts and ensure the wrapper takes precedence.
    - Finally, it adds a delegate to `ChatView` to ensure the new `SpatialChatUSDNetworkNode` is correctly displayed and interacted with in the Omniverse UI.
- **`on_shutdown(self)`**: This method ensures a clean shutdown by unregistering the custom nodes and, importantly, re-registering the *original* ChatUSD nodes, restoring the system to its pre-wrapper state.

3. The Prompt Simplifier Agent (`prompt.simplifier_agent.py`)

This agent is the first custom step in the wrapper's pre-processing pipeline. It aims to break down complex user inputs into simpler, more manageable instructions.

- **`invoke(self, inputs: AgentInput) -> AgentOutput`**: This method receives the raw user query. It applies basic rule-based simplification (e.g., splitting "and then" clauses, standardizing terms). Its output, `simplified_query`, is then passed to the next agent in the pipeline.

4. The Spatial Agent (`spatial_agent.py`)

This agent is where Gemini's spatial intelligence is applied. It takes the simplified prompt and uses Gemini to refine its spatial aspects.

- **`__init__(self, **kwargs)`**: Configures the `genai` library with the Gemini API key (assumed to be available via environment variable or Carb settings, consistent with the `register_chat_model.py` note).
- **`invoke(self, inputs: AgentInput) -> AgentOutput`**:
  - Receives `simplified_query` from the `PromptSimplifierAgent`.
  - Constructs a highly detailed `gemini_prompt` including system instructions, current scene info (if available), the simplified query, and examples. This is crucial for guiding Gemini's spatial reasoning.
  - Asynchronously calls `self.gemini_model.generate_content_async` to get Gemini's response.
  - Parses Gemini's output and extracts the `spatially_refined_prompt`.
  - Returns an `AgentOutput` containing this `spatially_refined_query`.

5. The Wrapper's Supervisor Network Node (`spatial_chat_usd_network_node.py`)

This file defines the new main entry point for the wrapped ChatUSD system, `SpatialChatUSDNetworkNode`, and its internal supervisor, `SpatialChatUSDSupervisorNode`. This is where the custom pre-processing pipeline is orchestrated.

- **`SpatialChatUSDSupervisorNode(RunnableNode)`**: This is the heart of the wrapper's logic.
  - Its `invoke` method orchestrates the full pipeline:
    1. Calls `PromptSimplifierAgent` to simplify the `user_query`.
    2. Calls `SpatialAgent`, passing the `simplified_query` (and potentially `current_scene_info` if integrated).
    3. The result, `spatially_refined_query`, is then set as the `self.outputs` of this supervisor node.
  - The `spatial_supervisor_identity.md` (not provided, but implied) would guide this supervisor's internal LLM on how to coordinate these steps and potentially how to formulate the final prompt for the original ChatUSD agents.
- **`SpatialChatUSDNetworkNode(MultiAgentNetworkNode)`**: This is the top-level network node for the wrapper.

- **`default_node: str = "SpatialChatUSDSupervisorNode"`**: This makes `SpatialChatUSDSupervisorNode` the primary logic handler for this new network.
- **`route_nodes: List[str]`**: This list explicitly includes both the custom agents (`"ChatUSD_PromptSimplifier"`, `"ChatUSD_SpatialAgent"`) and the *original* ChatUSD agents (`"ChatUSD_USDCode"`, `"ChatUSD_USDSearch"`, etc.). This configuration allows the internal supervisor (`SpatialChatUSDSupervisorNode`) to explicitly route to these agents in a defined sequence or based on its own LLM's reasoning.

Workflow of the Wrapper Approach:

1. **User Input**: User provides a prompt (e.g., "Create a large red cube and place it on top of a green cylinder.").
2. **Wrapper Intercepts**: Because `SpatialChatUSDNetworkNode` has taken the name "Chat USD" in `extension.py`, it becomes the initial recipient of the user's query.
3. **Prompt Simplification**: `SpatialChatUSDNetworkNode` (via its `SpatialChatUSDSupervisorNode`) invokes `PromptSimplifierAgent`. The prompt is simplified (e.g., "Create a large red cube. Place it on top of a green cylinder.").
4. **Spatial Intelligence**: The simplified prompt is then passed to `SpatialAgent`. `SpatialAgent` calls Gemini, which refines the spatial instructions (e.g., "Create a red cube with size X. Create a green cylinder with size Y. Place the red cube precisely on the top surface of the green cylinder, aligning their centers.").
5. **Hand-off to Original ChatUSD**: The `spatially_refined_query` is returned from `SpatialChatUSDSupervisorNode` to its parent `SpatialChatUSDNetworkNode`. The `SpatialChatUSDNetworkNode` then (either through further internal routing instructions or by directly invoking the original `ChatUSDNetworkNode` with the refined prompt) passes this highly detailed instruction to the core ChatUSD system.
6. **Original ChatUSD Execution**: The original `ChatUSDNetworkNode` receives the pre-processed and spatially enhanced prompt. It then proceeds as usual, routing it to its `USDCodeInteractiveNetworkNode` or other relevant agents to generate the final USD code and modify the scene.

Advantages of the Wrapper Approach:

- **Non-Intrusive**: Does not require modifications to the existing, stable ChatUSD codebase, making updates easier and reducing risk.
- **Modularity & Reusability**: The custom pre-processing agents are encapsulated within their own extension, promoting modularity and potential reusability.
- **Clean Separation of Concerns**: Clearly separates the spatial intelligence enhancement logic from the core USD scene generation logic.
- **Enable/Disable with Ease**: The entire spatial intelligence layer can be easily enabled or disabled by activating/deactivating the wrapper extension in Omniverse, without affecting the base ChatUSD.
- **Phased Deployment**: Allows for adding new capabilities incrementally without overhauling the main system.

Disadvantages/Limitations of the Wrapper Approach:

- **Increased Latency**: Adding an extra layer of agents and LLM calls (Simplifier, Spatial Agent, Gemini) inevitably adds more steps and network requests, increasing overall latency.
- **Overhead**: While non-intrusive, there's still a computational and memory overhead associated with running additional agents and managing communication between them.
- **Deep Integration Challenges**: If the spatial intelligence requires very deep, iterative interaction with the scene information or internal state of the original ChatUSD, a purely "pre-processor" wrapper might become cumbersome. It works best when the spatial refinement can be done largely independently before handing off to the core.
- **Implicit Contract**: Relies on a well-defined "contract" for input/output between the wrapper and the original ChatUSD system, which must be maintained.

Overall Impact & Results

The project focused on enhancing AI-driven, prompt-based factory layout design within NVIDIA Omniverse. The key achievements and their detailed implications are as follows:

- **Research and Development on Model Routing for Spatial Tasks**:
    - Significant R&D was conducted to investigate various approaches for routing spatial reasoning tasks to advanced models like Gemini 2.5 Flash. This involved exploring the use of orchestrators, custom wrappers, and specialized agents designed to handle and refine spatial prompts.

    - **Clarification**: While these approaches were explored and foundational R&D was performed, the full and successful integration of the Gemini model for live spatial tasks within the current system was identified as a future direction rather than a completed outcome. The work laid the groundwork for how such an integration *could* be achieved.
- **Successful Modular Agent Integration**:
    - A major success was the development and seamless embedding of several custom agents into the existing Chat USD architecture. These agents include:
        - **Greeting Agent**: For handling initial user interactions and basic conversational responses.
        - **Simplifier Agent**: Designed to break down complex user prompts into more atomic and manageable instructions, making them easier for subsequent AI models to process.
        - **Orchestrator Agent**: Responsible for intelligently directing user queries to the appropriate specialized agents within the multi-agent system.
    - This integration was achieved without disrupting Chat USD's core architecture, demonstrating the system's flexibility and modularity. It successfully expanded Chat USD's capabilities by adding specialized functionalities that improve prompt processing and conversational flow.

Tools and Frameworks Used

The development of the spatial intelligence integration within ChatUSD leverages a robust stack of tools and frameworks, combining NVIDIA Omniverse technologies with leading AI models and standard development practices.

Core Omniverse Frameworks

- **NVIDIA Omniverse Kit**: The foundational platform that provides the real-time 3D development environment and simulation capabilities. All custom agents and extensions are built on top of this kit.
- **Omniverse Extensions**: The modular system within Omniverse Kit that allows for the creation, packaging, and loading of custom functionalities. This includes the use of `extension.toml` for configuration and `extension.py` as the entry point for custom logic.
- **ChatUSD Architecture**: The existing multi-agent system within Omniverse that enables natural language interaction for USD scene creation and manipulation. Key components reused or wrapped include:

    - **ChatUSD Network Node (`ChatUSDNetworkNode`)**: The primary entry point for user queries in the original system.
    - **ChatUSD Supervisor Node (`ChatUSDSupervisorNode`)**: Orchestrates the routing of prompts to various specialized agents.
    - **USD Code Generation (`USDCodeInteractiveNetworkNode`)**: Responsible for generating USD (Universal Scene Description) code based on prompts.
    - **USD Search (`USDSearchNetworkNode`)**: Facilitates searching for USD assets.
    - **Scene Information (`SceneInfoNetworkNode`)**: Gathers information about the current 3D scene.
- **LangChain Agent Framework (`lc_agent`)**: An NVIDIA-developed framework for building and orchestrating modular AI agents within Omniverse. It provides fundamental classes like `RunnableNode` (for individual agent logic) and `MultiAgentNetworkNode` (for managing agent workflows).

Artificial Intelligence Models & Libraries

- **Google Gemini (1.5 Flash)**: Specifically utilized for its advanced spatial reasoning capabilities. The `SpatialAgent` directly interfaces with the Gemini API to refine and enrich user prompts with precise spatial details.
- **NVIDIA API / NVIDIA NIM**: While Gemini handles spatial tasks, the broader ChatUSD system, as indicated by `register_chat_model.py` and `extension.toml`, is designed to integrate with NVIDIA's own powerful language models (e.g., Llama-3.1-70B-Instruct) for general USD code generation and dialogue.
- **`google.generativeai`**: The Python client library used to interact with the Google Gemini API.

Programming Languages & Utilities

- **Python**: The primary programming language for implementing all custom agents, network nodes, and extension logic.
- **TOML**: The configuration file format (`.toml` files) used for defining Omniverse extensions, including their metadata, dependencies, and settings.
- **Carb (Omniverse Carbon Library)**: A core Omniverse library used for logging, settings management, and other low-level system interactions within the Omniverse environment.

- `pathlib`: A Python standard library module used for handling file system paths in an object-oriented manner, simplifying file operations within the extension.
- `aiohttp`: An asynchronous HTTP client/server framework for Python, used for making non-blocking API calls, particularly to large language models.

Future Roadmap & Recommendations

The future roadmap and recommendations for enhancing AI-driven factory layout generation within NVIDIA Omniverse focus on several key areas, emphasizing continuous improvement, expanded capabilities, and a flexible multi-model approach:

- **Enhance Spatial Consistency**:
  - **Prompt Tuning**: Improve the precision and reliability of spatial outputs through advanced prompt engineering techniques.
  - **Scene Validation Checks**: Implement robust mechanisms to validate generated scenes against real-world constraints and design rules.
  - **Robust SpatialCollector Integration**: Further refine the integration of the `SpatialCollector` to ensure accurate aggregation and seamless passing of spatial information between different agents and models.
- **Broaden Domain Semantics**:
  - Expand Chat USD's understanding to include a wider range of industrial components, their functionalities (object affordances), and complex contextual relationships within factory environments. This will enable more intelligent and nuanced scene generation.
- **LLM Fine-Tuning for Spatial Tasks**:
  - Explore the fine-tuning of large language models (LLMs) specifically for spatial reasoning tasks. This includes investigating models like LLaMA or other open-source alternatives.
  - **Multi-Model Flexibility**: It's important to note that the multi-model approach is not restricted to Gemini. The system is designed to be flexible, allowing for the integration and leveraging of various LLMs, including other powerful proprietary models or fine-tuned open-source models, to optimize performance and reduce reliance on single external dependencies.
- **Close the Loop with Feedback**:
  - Implement real-time user feedback and correction loops. This mechanism will enable continuous learning, allowing the system to adapt and improve its scene generation capabilities based on user interactions and explicit corrections.

Key Learnings

This project offered a rich learning experience, fostering growth in both technical capabilities and essential professional skills:

- **Technical Learnings**:
    - **Modular AI System Design and Implementation**: Gained profound hands-on experience in the practical application of designing and working with modular AI systems within a real-world context, specifically the NVIDIA Omniverse platform. This involved understanding how different AI agents and components interact and contribute to a larger, complex system.
    - **Handling Open-Ended Technical Challenges**: The nature of the project, involving novel AI integration and problem-solving, significantly boosted confidence in tackling ambiguous and open-ended technical challenges. This included iterating on solutions, debugging complex inter-agent communications, and adapting to unforeseen technical hurdles inherent in cutting-edge AI development.

- **Soft Skill Learnings**:
    - **Effective Communication and Teamwork**: Realized the critical importance of clear, concise communication and collaborative teamwork, particularly when building intricate and interdependent systems. Successfully navigating discussions, sharing progress, and coordinating efforts among different components or team members were essential for project success.
    - **Systematic Problem-Solving and Mentorship Impact**: Benefited significantly from mentorship, which instilled a more systematic and structured approach to problem-solving. This involved breaking down large problems into smaller, manageable parts, prioritizing tasks, and applying methodical thinking to identify and resolve issues. The guidance received was instrumental in developing a more disciplined engineering mindset.

Summary

This report details an in-depth research and development journey focused on enabling AI-driven, prompt-based factory layout design within NVIDIA Omniverse. The project aimed to bridge the gap between natural language commands and the complex requirements of 3D scene construction for industrial environments.

The foundational work leveraged a robust stack of **Tools and Frameworks**, including NVIDIA Omniverse Kit and its extension system, the existing ChatUSD multi-agent architecture, and the LangChain agent framework for orchestrating AI components. Google Gemini 1.5 Flash was explored for its spatial reasoning capabilities, alongside Python for implementation and TOML for configuration.

In terms of **Overall Impact & Results**, the project successfully conducted R&D on various approaches for intelligent model routing, particularly for spatial tasks. While the direct integration of Gemini was a proof-of-concept for future possibilities rather than a finalized implementation, a significant achievement was the successful modular integration of custom agents (e.g., Greeting, Simplifier, and Orchestrator) into the ChatUSD system without disrupting its core architecture.

The **Future Roadmap & Recommendations** highlight continuous improvement areas. These include enhancing spatial consistency through advanced prompt tuning and robust scene validation, broadening the system's understanding of industrial domain semantics, and exploring LLM fine-tuning for spatial tasks using models like LLaMA or other open-source alternatives. A key takeaway for the future is the emphasis on a flexible multi-model approach, not restricted to any single LLM. Additionally, implementing feedback loops for continuous learning is crucial.

Finally, the **Key Learnings** derived from this project span both technical and soft skills. Technically, the experience provided insights into designing and working with modular, real-world AI systems and boosted confidence in tackling open-ended technical challenges. From a soft skills perspective, the project underscored the importance of clear communication and teamwork in complex system development, alongside the invaluable impact of mentorship in fostering systematic problem-solving abilities.