SceneGenAgent: Precise Industrial Scene Generation with Coding Agent

Xiao Xia[1*], **Dan Zhang**[1*], **Zibo Liao**[2], **Zhenyu Hou**[1],
Tianrui Sun[2], **Jing Li**[2], **Ling Fu**[2], **Yuxiao Dong**[1]
[1]Tsinghua University [2]Foundational Technologies, Siemens Ltd., China

Abstract

The modeling of industrial scenes is essential for simulations in industrial manufacturing. While large language models (LLMs) have shown significant progress in generating general 3D scenes from textual descriptions, generating industrial scenes with LLMs poses a unique challenge due to their demand for precise measurements and positioning, requiring complex planning over spatial arrangement. To address this challenge, we introduce `SceneGenAgent`, an LLM-based agent for generating industrial scenes through C# code. `SceneGenAgent` ensures precise layout planning through a structured and calculable format, layout verification, and iterative refinement to meet the quantitative requirements of industrial scenarios. Experiment results demonstrate that LLMs powered by `SceneGenAgent` exceed their original performance, reaching up to 81.0% success rate in real-world industrial scene generation tasks and effectively meeting most scene generation requirements. To further enhance accessibility, we construct `SceneInstruct`, a dataset designed for fine-tuning open-source LLMs to integrate into `SceneGenAgent`. Experiments show that fine-tuning open-source LLMs on `SceneInstruct` yields significant performance improvements, with Llama3.1-70B approaching the capabilities of GPT-4o. Our code and dataset are available at `https://github.com/THUDM/SceneGenAgent`.

1 Introduction

The advancement of large language models (LLMs) (Brown et al., 2020; Kaplan et al., 2020; Touvron et al., 2023; Zeng et al., 2023b; GLM et al., 2024) enables the automatic design of scene layouts, providing a promising direction for the automatic generation of industrial scenes. Automatically generated digital modeling of industrial scenes can be used for design visualization, simulation, analysis, and optimization in manufacturing. Recent scene generation research (Sun et al., 2023; Yang et al., 2024b; Hu et al., 2024) has made remarkable achievements in translating textual descriptions into 3D scenes, grasping a delicate understanding of spatial layout.

However, compared to general scene generation, the datasets and evaluations of which (Chang et al., 2017; Fu et al., 2021; Sun et al., 2023; De La Torre et al., 2024; Yang et al., 2024b) often prioritize qualitative plausibility, *industrial scene generation* presents the challenge of requiring quantitative measurements and precise positioning to maintain the functional integrity of machinery and safety compliance. Industrial scenes are described with specified design parameters and constraints such as coordinates, distances, and orientations to which the components must align with strict adherence (cf. Figure 1). However, current LLMs struggle to meet the precision requirements of industrial scene generation without sufficient quantitative analysis of complex layout (Yang et al., 2024a), and we also demonstrate this drawback in Table 3. Meanwhile, LLMs used as agents (Shinn et al., 2023; Yao et al., 2024; Wang et al., 2023; Xu et al., 2024b; Wang et al., 2024; Zhang et al., 2025) bring possibilities to handle complex scenes with planning.

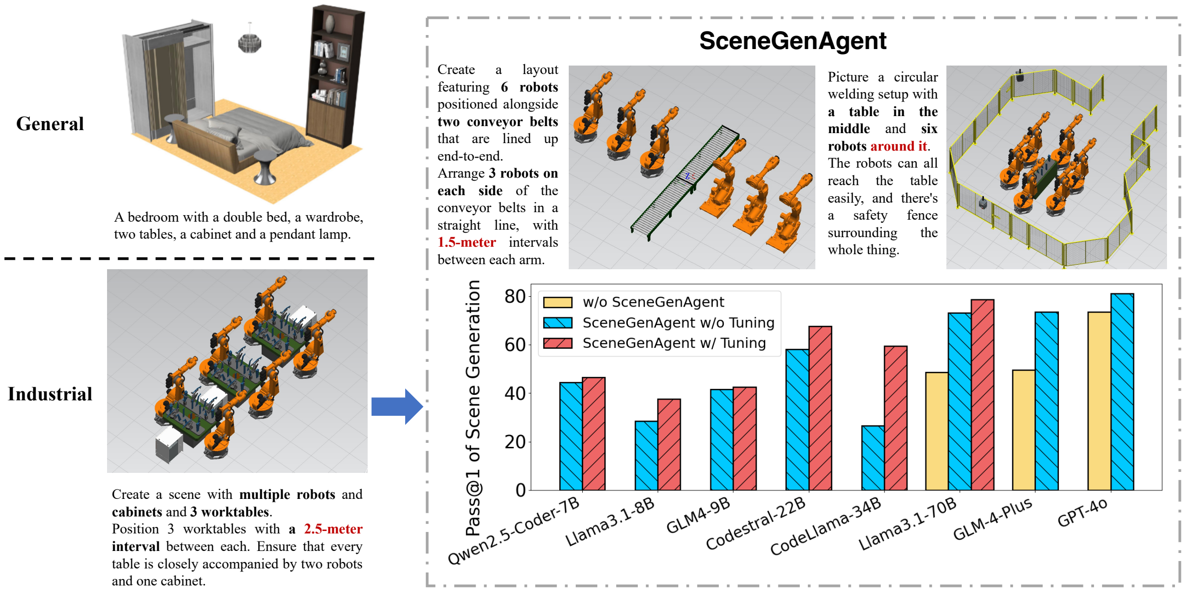
In this paper, we introduce `SceneGenAgent`, an LLM-based agent that enables precise generation of industrial scenes. `SceneGenAgent` translates scene descriptions into C# code, leveraging APIs provided by Tecnomatix[1], a dedicated digital manufacturing software developed by Siemens, and executes the code to render the corresponding scene. We prioritize the accurate placement within the scene in `SceneGenAgent` by designing a layout analysis pipeline. Specifically, `SceneGenAgent` **plans the overall layout** by representing spatial

---

[*] Equal contribution.

[1] https://plm.sw.siemens.com/en-US/tecnomatix/

Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 17847–17875
July 27 - August 1, 2025 ©2025 Association for Computational Linguistics



A bedroom with a double bed, a wardrobe, two tables, a cabinet and a pendant lamp.

Create a scene with **multiple robots** and **cabinets** and **3 worktables**. Position 3 worktables with **a 2.5-meter interval** between each. Ensure that every table is closely accompanied by two robots and one cabinet.

Create a layout featuring **6 robots** positioned alongside **two conveyor belts** that are lined up end-to-end. Arrange **3 robots on each side** of the conveyor belts in a straight line, with **1.5-meter** intervals between each arm.

Picture a circular welding setup with **a table in the middle** and **six robots around it**. The robots can all reach the table easily, and there's a safety fence surrounding the whole thing.

Figure 1: **Left: General and industrial scene generation examples.** Industrial scene generation demands quantitative precision inputs compared to general scenes. **Right: Examples of `SceneGenAgent` and leaderboard of LLMs incorporated in `SceneGenAgent` for industrial scene generation.** Llama3.1-70B and API-based LLMs achieve high pass@1. Fine-tuned LLMs show consistent improvement for different parameter sizes. General scene example is selected from LayoutGPT (Feng et al., 2024) and industrial scene examples are generated by `SceneGenAgent` with the fine-tuned Llama3.1-70B as its backbone.

specifications including positions and relations in a structured, calculable format and computing and reasoning with numerical constraints. After producing a layout plan, `SceneGenAgent` **verifies the layout** for compliance with the description and employs identified flaws as feedback to refine the layout. Finally, `SceneGenAgent` **generates code** based on the layout plan and Tecnomatix API coding guidance. Experimental results show that LLMs powered by `SceneGenAgent` exceed their original performance, with GPT-4o (Achiam et al., 2023) reaching 81.0% success rate in real-world industrial scene generation tasks and effectively meeting most scene generation requirements.

We introduce an approach for fine-tuning open-source LLMs to integrate into `SceneGenAgent`, allowing individuals to deploy it independently. Given the absence of publicly available datasets for industrial scenes, we construct `SceneInstruct`, a dataset capturing the complexity of industrial scene descriptions and covering key steps in `SceneGenAgent` pipeline. Experiments show that fine-tuned LLMs of different scales exhibit significant improvements over their counterparts without tuning, with Llama3.1-70B (Dubey et al., 2024) nearing the performance of the leading GPT-4o, showcasing the efficacy of our approach.

The contributions are summarized as follows:

- We propose `SceneGenAgent`, an LLM-based coding agent capable of generating industrial scenes from descriptions, analyzing and improving layout for precise generation, shown in Figure 2.
- We construct `SceneInstruct`, a dataset that encompasses the core functions of `SceneGenAgent` and is used to fine-tune open LLMs to enable individual agent deployment, detailed in Table 1.
- Our experiments demonstrate that API-based and open-source fine-tuned LLMs empowered by `SceneGenAgent` achieve high success rates in real-world industrial scene generation tasks, as summarized in Table 2 and Table 3.

2 Preliminary

Modeling Industrial Scenes. A scene can be represented as a collection of 3D models, denoted as objects $\mathcal{O}$, drawn from an object library. Each object $o_i \in \mathcal{O}$ in the scene is characterized by its coordinates c_i and direction d_i. The coordinates specify the object's location on a two-dimensional plane, where the positive x-axis represents the front and the positive y-axis represents the left. The direction defines the object's angle rotated counterclockwise from the positive direction of the x-axis. The placement of an object, $p_i = (o_i, c_i, d_i)$, encapsulates its spatial properties. The Euclidean distance between two objects must be at least 1 meter to avoid collision. The set of all objects' placements $\mathcal{P} = \{p_i = (o_i, c_i, d_i) | o_i \in \mathcal{O}\}$ provides a modeling of the scene. Given an industrial scene's description, we aim to identify the objects and their placements to reconstruct the scene.

Code Representation of Industrial Scenes. We use C# code to leverage Tecnomatix APIs and model industrial scenes. The code aims to place objects from $\mathcal{O}$ according to the predefined placement $\mathcal{P}$. We restrict the code's functionality to loading object models and placing objects to streamline API integration. The first selects and loads the model files of objects according to $\mathcal{O}$, and the second sets coordinates and directions for objects with specifications outlined in $\mathcal{P}$. The code is run in the Process Simulate software[2] to render the scene.

3 Method

This section consists of two parts: **`SceneGenAgent`** and **`SceneInstruct`**. Regarding the first part, we design an agent, `SceneGenAgent`, as shown in Figure 2, which leverages LLMs to translate the description into scene-modeling code. In the second part, we synthesize a new dataset `SceneInstruct` to fine-tune open-source LLMs to achieve performance comparable to API-based models.

3.1 `SceneGenAgent`

We introduce the workflow of an industrial scene-generating agent, which consists of two phases: **Layout Analysis (Agent)** aiming to extract and analyze the scene's layout and **Code Generation** for generating code using Tecnomatix APIs.

3.1.1 Layout Analysis (Agent)

`SceneGenAgent` performs layout analysis to recover the objects $\mathcal{O}$ and placements $\mathcal{P}$. Layout analysis consists of three stages: object retrieval, layout extraction, and placement assignment.

Object retrieval. `SceneGenAgent` identifies the objects in the scene and retrieves the corresponding items from the object library. The agent executes all the following operations in a single LLM call using the description as input.

First, the agent generates a list of objects that should appear in the scene based on the description. This list includes ambiguously mentioned objects like "a random object". The agent then maps the objects to their counterparts within the library by incorporating the objects from the library and objects found in the description into the LLM's input context and tasking the LLM to verify the presence of each object in the description within the library. If an object is not found in the library, the agent attempts to substitute it with a similar object from the library that conforms to the description. In cases where no suitable substitute is available, the agent removes the object. Consequently, the agent retrieves the object collection $\mathcal{O} = \{o_i\}$ from the library. Finally, the agent rewrites the description to reflect the retrieved objects' layout, replacing the original objects' names with those of the retrieved ones and retaining solely layout information. The output of this rewriting is the layout's concise depiction, denoted as the new description s.

Layout extraction. `SceneGenAgent` extracts all layout information from s. We categorize the existing layout information in s into two types:

- **Position**, which refers to the positional state of an object, noted as $\mathrm{p} = (o_i, \mathrm{l}_i, \mathrm{d}_i)$, where o_i denotes the object, l_i the location, and d_i the direction. Note that some objects may have no p or their l_i or d_i may be missing because the description may not provide this information.
- **Relation**, which concerns the interactions between two objects, including aspects such as distance and relative direction. This is expressed as $\mathrm{r} = (o_i, o_j, r_{ij})$, where o_i and o_j denotes the two objects and r_{ij} their relationship. r_{ij} is a text describing relationships such as adjacency, parallel, surrounding, etc. Relations involving multiple objects are represented as a series of relations between object pairs, which, accompanied by the description s, can be comprehended by the agent.

The existing layout information in s can be formalized into two sets $\mathbb{P} = \{\mathrm{p}\}$ and $\mathbb{R} = \{\mathrm{r}\}$, representing the set of positions and relations respectively. The agent extracts all layout information with an LLM call, incorporating s and $\mathcal{O}$ into the input and listing positions and relations in the output.

Placement assignment. `SceneGenAgent` assigns the final placement of every object $\mathcal{P}$ based on $\mathbb{P}$ and $\mathbb{R}$ and improves $\mathcal{P}$ through verification. Beyond adhering to the description, the assignment must also satisfy supplementary constraints, i.e., the specifications concerning object distance and orientation, to ensure the coherence of the scene.

---

[2] https://plm.sw.siemens.com/en-US/tecnomatix/products/process-simulate-software/

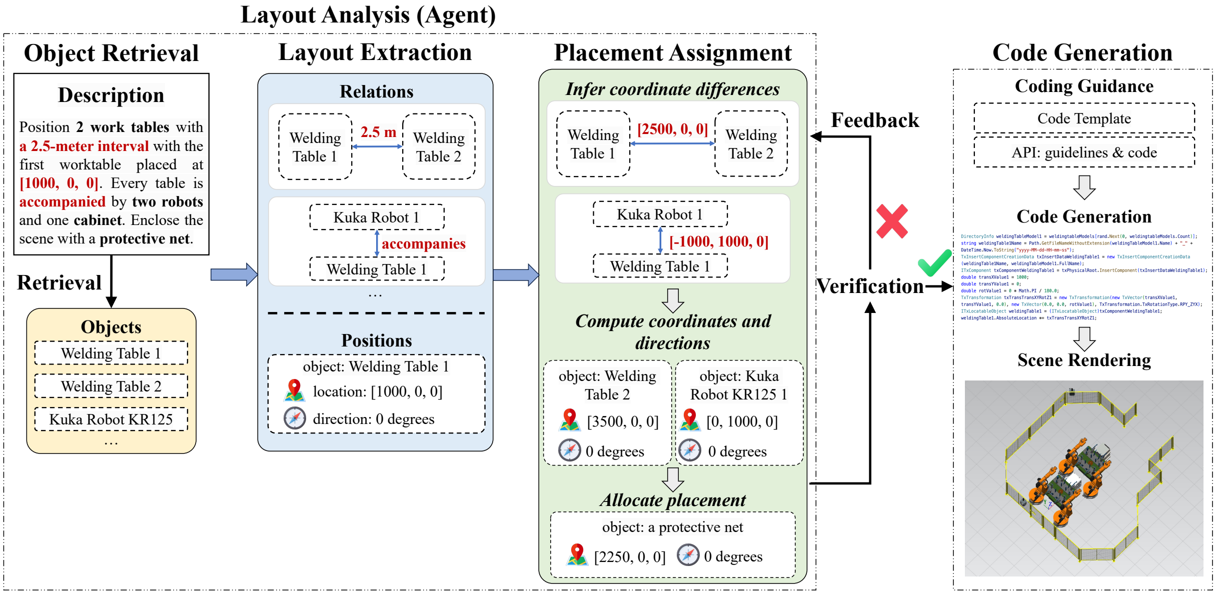


Figure 2: **Overview of `SceneGenAgent` framework.** Based on the description, the agent first retrieves objects from the library and then extracts positions and relations from the description. The relations are converted into coordinate differences, which are used together with the positions to calculate and infer the placement of all objects. After passing verification, the placement is used to generate code.

The agent undertakes the following steps:

1. *Infer placement from* $\mathbb{P}$: For each position information p $= (o_i, \mathrm{l}_i, \mathrm{d}_i)$, if c_i or d_i of o_i is included in or can be deduced from l_i or d_i, assign c_i or d_i to o_i.
2. *Infer coordinate differences from* $\mathbb{R}$: Relations r $= (o_i, o_j, r_{ij}) \in \mathbb{R}$ are viewed as differences in coordinates between two objects. For instance, the distance between two objects is expressed as the magnitude of their coordinate difference vector. Rewrite all relations into coordinate differences, forming the set $\Delta = \{(o_i, o_j, \delta_{ij}) | \delta_{ij}$ is rewritten from $r_{ij}, (o_i, o_j, r_{ij}) \in \mathbb{R}\}$, where δ_{ij} represents the vector difference between the coordinates of o_i and o_j.
3. *Compute coordinates and directions*: For each coordinate difference $(o_i, o_j, \delta_{ij}) \in \Delta$, if the coordinates of one object are known, compute the other's coordinates based on δ_{ij}. For each relation r $= (o_i, o_j, r_{ij}) \in \mathbb{R}$, if r_{ij} states that o_i is oriented towards o_j and the coordinates of both objects are known, calculate the direction of o_i, and vice versa.
4. *Allocate placement to the rest of the objects*: Allocate placement to objects whose coordinates or directions have not yet been determined without collision, adhering to the description.

All objects are placed after these steps, forming the initial placement $\mathcal{P}$.

Verification. The initial placement may contain flaws since the LLM may make common-sense errors or miscalculate values. Recent studies (Shinn et al., 2023; Madaan et al., 2024) have shown the ability of language models to critique their own decisions. Hence, `SceneGenAgent` iteratively checks $\mathcal{P}$ for the following potential errors: (1) whether the placement violates the constraints described in s and (2) whether any objects collide, measured by the distances between adjacent objects. If errors are identified, the agent uses $\mathbb{P}$, $\mathbb{R}$, $\mathcal{P}$, and the detected errors as input and attempts to fix the errors to generate a new $\mathcal{P}$ until no errors are found.

3.1.2 Code Generation

This phase involves writing C# code with Tecnomatix APIs to visualize the layout and model the scene. We design a code template containing basic code, such as initialization, and leave the parts responsible for model loading and object placement incomplete for the agent to fill. Following prior works (Zhou et al., 2023) that integrate code documentation into the model's input, we provide **coding guidance** alongside the template, aiding the agent in using APIs. The coding guidance includes usage guidelines and example code snippets corresponding to APIs. The agent employs an LLM to **generate code**, integrating $\mathcal{P}$, code template, and coding guidance into the input.

Potential errors can be identified within the generated code, including omitting code or making up methods or variables unsupported by Tecnomatix. The agent conducts a rule-based evaluation by scanning keywords associated with the APIs in the code to assess whether the LLM has correctly implemented the API calls. The detected invalid code is used as feedback to prompt the LLM to revise the code accordingly. Once the code passes the check, we run it to **render the industrial scene**.

3.2 `SceneInstruct`

This section details our efforts in training open-source LLMs to integrate into `SceneGenAgent`. Inspired by previous research (Zeng et al., 2023a; Chen et al., 2023) fine-tuning LLMs on agent trajectories to enhance their agent abilities, we gather the agent's trajectory data to build `SceneInstruct`, constructing a collection of industrial scene descriptions and feeding them into the agent with API-based models as its backbone. Then, we use `SceneInstruct` to fine-tune open-source LLMs.

3.2.1 `SceneInstruct` Construction

Instruction Construction. We construct a description dataset that serves as the agent's instruction to obtain its behavior trajectory. We manually write 20 scene descriptions as the initial description pool, encompassing fundamental variations of industrial scenes. Then, we adopt the Evol-Instruct method (Xu et al., 2024a) to rewrite descriptions iteratively with an LLM, generating diverse variants. We design 6 rewriting methods based on possible variations of descriptions. In each iteration, a random description from the current pool and a rewriting method are sampled. The description is rewritten by Llama3.1-70B (Dubey et al., 2024) into a new description. Appendix A.4.1 details the rewriting strategies and sampling.

Each new description undergoes validation and deduplication. Validation detects errors including common-sense mistakes like collisions and contradictions between different parts of the description. An object's placement might be specified multiple times, but its placements derived in these different ways may not match. The LLM first attempts to compute placements, distances, and relative positions as much as possible based on the description, then examines common-sense errors and conflicts in calculated results. Deduplication evaluates the similarity between new and existing descriptions with the MinHash (Broder, 2000) algorithm. The LLM retries rewriting if validation or deduplication fails, and then we add it to the description pool for sampling in later iterations. The final dataset includes 3002 descriptions, from which we use 95% (2852) for training and the rest for validation.

Table 1: **Overview of `SceneInstruct` for training.** **Assign** represents assigning, **Verify** represents verifying whereas (Pos/Neg) represents data with/without errors, and **Reassign** represents reassigning. "Base" represents data constructed with GPT-4 and "Extra" represents data constructed with Llama3.1-8B assigning placement.

Task	Assign	Verify (Pos/Neg)	Reassign
#Base	2852	5413 (2561/2852)	682
#Extra	2852	6984 (4132/2852)	1905
#Total	5704	12397 (6693/5704)	2587

Trajectory Collection. After constructing the instructions, we collect `SceneGenAgent`'s trajectories with GPT-4 (Achiam et al., 2023) as its backbone, gathering the inputs and outputs of GPT-4 within the workflow. We refer to Llama3.1-70B, given its capabilities for instruction following, multi-turn dialogue, and writing C# code, to select which parts of the agent need data collection. We observe that while it can handle most operations, the performance in assigning placement can be improved. Therefore, we gather data from this stage, where the LLM should complete three tasks: assign, verify, and reassign placement. For each description, Llama3.1-70B first completes the agent's tasks up to extracting layout, then GPT-4 assigns placement and we collect its inputs and outputs.

Data containing inaccuracies is crucial to train the model to be sensitive to errors during placement verification. Nevertheless, the positions assigned by GPT-4 tend to be classified as correct. Thus, we add extra data by changing the LLM that assigns placement to the weaker Llama3.1-8B to obtain more data with errors. We add this data to the existing dataset, forming `SceneInstruct`. Table 1 shows the statistics of `SceneInstruct` for training.

3.2.2 `SceneInstruct` Tuning

We choose the chat versions of Qwen2.5-Coder-7B (Hui et al., 2024), Llama3.1-8B (Dubey et al., 2024), GLM4-9B (GLM et al., 2024), Codestral-22B (Jiang et al., 2023), CodeLlama-34B (Roziere et al., 2023), and Llama3.1-70B (Dubey et al., 2024) as base models for fine-tuning. For a base

Table 2: **Main results.** The best performances for open or closed LLMs are in **bold**. All LLMs are evaluated by incorporating them into the `SceneGenAgent`. API-based LLMs complete all agent operations, and open LLMs are utilized during placement assignment while Llama3.1-70B handles the remaining operations. For pass@1 categories, Geo. represents geometric arrangement, Pos. represents positional details, Quant. represents object quantity, Comp. represents composite description, Fuzz. represents fuzzy description, and Overall represents the overall metric.

Model	**Type**	**Fine-tuned**	**pass@1**					
(w/ `SceneGenAgent`)			Geo.	Pos.	Quant.	Comp.	Fuzz.	Overall
GLM-4-Plus	closed	-	72.0	68.9	**76.7**	68.0	**90.0**	73.5
GPT-4o	closed	-	**81.3**	**82.2**	73.3	**84.0**	**90.0**	**81.0**
Qwen2.5-Coder-7B	open	✗	40.0	42.2	46.7	32.0	80.0	44.5
		✓	53.3	33.3	53.3	40.0	60.0	46.5
Llama3.1-8B	open	✗	16.0	15.6	40.0	24.0	90.0	28.5
		✓	22.7	33.3	46.7	44.0	85.0	37.5
GLM4-9B	open	✗	33.3	35.6	46.7	48.0	75.0	41.5
		✓	36.0	26.7	50.0	44.0	85.0	42.5
Codestral-22B	open	✗	54.7	46.7	70.0	48.0	90.0	58.0
		✓	62.7	62.2	**80.0**	52.0	90.0	67.5
CodeLlama-34B	open	✗	26.7	17.8	26.7	28.0	50.0	26.5
		✓	50.7	55.6	60.0	**64.0**	85.0	59.5
Llama3.1-70B	open	✗	64.0	**80.0**	76.7	**64.0**	**95.0**	73.0
		✓	**78.7**	77.8	**80.0**	60.0	90.0	**78.5**

model, we utilize LoRA (Hu et al., 2022), fine-tune multiple LoRA modules corresponding to tasks, and dynamically switch modules in deployment. We organize data for assignment and verification as question-answer pairs and data for reassigning as conversations. We calculate the loss of outputs in the final rounds during fine-tuning. Detailed training hyper-parameters are in Appendix A.6.

4 Experiments

4.1 Experiment Setup

Benchmark. We develop a comprehensive industrial scene layout generation benchmark containing 40 descriptions of industrial scenes. These descriptions are written by 15 engineers based on their experience and split into 5 categories: geometric arrangement, focusing on geometric patterns; positional details, dealing with numerical measurements; object quantity, about the number of objects present; composite description, involving detailed, long-form textual descriptions; fuzzy description, an overview without precise specifications. The proportion of each category reflects its frequency of occurrence. Details are provided in Appendix A.9.

Metric and evaluation. We adopt the pass@k metric (Chen et al., 2021), a commonly used metric for coding performance that measures the ratio of programming problems where at least one out of k generated code samples qualifies. In the absence of automated testing tool suites, we rely on manual verification, carefully comparing each scene against its description. Our evaluation team consists of 3 members with expertise in industrial scenes and proficiency in the Process Simulate software. Due to the labor-intensive nature of the manual inspection, we sample 5 times for each description with temperature $t = 1$ and evaluate pass@1 with the estimator pass@$k = \mathbb{E}[1 - \binom{n-c}{k}/\binom{n}{k}]$, where n is the number of samples for each problem (5 in this work), c is the number of correct samples, and k is set to 1. Each evaluator reviews at least one of the five samples of each description to minimize the influence of subjectivity, mitigating potential biases introduced by different evaluators.

Models. We evaluate the performance of multiple LLMs in `SceneGenAgent`. For API-based LLMs, we select GPT-4o (Achiam et al., 2023) and GLM-4-Plus (GLM et al., 2024), both known for their strong coding and reasoning capabilities. As for

Table 3: **Ablation study on the effect of layout analysis.** "w/o agent" indicates obtaining objects and layout in one LLM call, "w/o verification" indicates removing verification during placement assignment, and "w/ `SceneGenAgent`" indicates the designed workflow. For pass@1 categories, Geo.& Pos. represents combining geometric arrangement and positional details, and the remaining categories are the same as Table 2.

Model	**Layout Analysis**	**pass@1**				
		Geo. & Pos.	Quant.	Comp.	Fuzz.	Overall
GPT-4o	w/ `SceneGenAgent`	81.7	73.3	84.0	90.0	81.0
	w/o agent	72.5	83.3	64.0	90.0	73.5 (**-9.3%**)
	w/o verification	73.3	80.0	76.0	95.0	77.0 (**-4.9%**)
GLM-4-Plus	w/ `SceneGenAgent`	70.8	76.7	68.0	90.0	73.5
	w/o agent	46.7	60.0	52.0	60.0	49.5 (**-32.7%**)
	w/o verification	58.3	56.7	44.0	90.0	60.5 (**-17.7%**)
Llama3.1-70B	w/ `SceneGenAgent`	70.0	76.7	64.0	95.0	73.0
	w/o agent	44.2	43.3	40.0	80.0	48.5 (**-33.6%**)
	w/o verification	70.0	73.3	60.0	80.0	70.5 (**-3.4%**)
Llama3.1-70B (Fine-tuned)	w/ `SceneGenAgent`	78.3	80.0	60.0	90.0	78.5
	w/o verification	69.2	76.7	68.0	85.0	72.5 (**-7.6%**)

open-source LLMs, we include those we fine-tuned with `SceneInstruct` in Section 3.2.2. When incorporated into `SceneGenAgent`, API-based LLMs handle all agent operations, and open-source LLMs are applied during placement assignment with Llama3.1-70B managing the remaining operations.

4.2 Main Results

Table 2 presents the performance of LLMs integrated into the agent. Larger models achieve high accuracy in the scene generation task, with GPT-4o exhibiting the highest pass@1 among all models, effectively solving most tasks. The open models' pass@1 generally remain lower than closed models. Smaller models solve only a limited subset of tasks, whereas Llama3.1-70B achieves performance comparable to closed models. The results also suggest that fine-tuning with `SceneInstruct` significantly benefits open models. After fine-tuning, all open models improve overall pass@1, with most models also exhibiting higher pass@1 in most categories. Llama3.1-70B approaches the top-performing GPT-4o, increasing its pass@1 from 73.0% to 78.5%.

Overall, the results highlight the efficacy of `SceneGenAgent` supported by closed and open LLMs for industrial scene generation. Furthermore, the findings emphasize the effectiveness of our proposed data construction and fine-tuning methodology, with fine-tuning identified as a critical factor in enhancing the performance of open LLMs.

4.3 Ablation Study

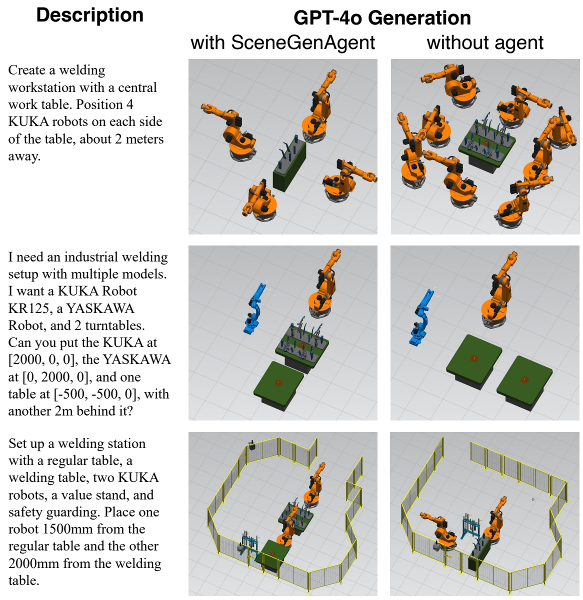
We conduct detailed ablation studies to demonstrate the effectiveness of our approach in addressing key challenges in industrial scene generation, including the precise placement of objects and the limited availability of training data.

4.3.1 Necessity of agent integration

We implement a sophisticated agent workflow to detect objects within the scene, assign spatial placements, and verify and adjust them. To confirm the necessity of this agent design, we test two settings: (1) obtaining objects and layout in a single LLM call, that is, not adopting the agent methodology, and (2) omitting verification after assigning placement, simplifying the agent. In the first case, the LLM should directly identify the objects and layout from the description. We also add all requirements to the input, including object library and specifications concerning object distance and orientation, to guarantee that the LLM can access the same knowledge as the agent. The prompt for this call is in Appendix A.7.2. In the second case, we skip verifying placement and directly proceed to code generation. We evaluate the performance under both settings and compare it with the original `SceneGenAgent`.

We evaluate GPT-4o, GLM-4-Plus, and both base and fine-tuned Llama3.1-70B. Since the fine-tuned Llama3.1-70B is trained with prompts different from those in the first setting, we only evaluate it in the second setting. The results, provided in Table 3, show performance degradation in all models without the agent or verification. The performance drop in geometric arrangement and positional details marks the incidence of spatial errors rising, proving that the simplified settings lack the complexity required for precise scene generation.

To demonstrate the improvements achieved by powering GPT-4o with `SceneGenAgent`, we present several examples of scenes generated by GPT-4o, with and without the integration of `SceneGenAgent`, in Figure 3. In the examples, all scenes generated by GPT-4o with `SceneGenAgent` accurately reflect the descriptions, whereas those generated without `SceneGenAgent` exhibit various inaccuracies. Specifically, in the scenes generated without `SceneGenAgent`, the first example includes eight tables instead of the described four; the second misplaces the second table to the right of the first rather than behind it; and the third erroneously contains only one table instead of two and assumes the two robots are positioned relative to the same table rather than distinct ones. These examples highlight that even advanced models like GPT-4o can be susceptible to errors concerning the number of objects, their spatial orientations, and inter-object relationships. In contrast, `SceneGenAgent`, providing a nuanced understanding of placement, contributes notably to complex scene generation ability, even improving the performance of GPT-4o, which already has strong capabilities.



Create a welding workstation with a central work table. Position 4 KUKA robots on each side of the table, about 2 meters away.

I need an industrial welding setup with multiple models. I want a KUKA Robot KR125, a YASKAWA Robot, and 2 turntables. Can you put the KUKA at [2000, 0, 0], the YASKAWA at [0, 2000, 0], and one table at [-500, -500, 0], with another 2m behind it?

Set up a welding station with a regular table, a welding table, two KUKA robots, a value stand, and safety guarding. Place one robot 1500mm from the regular table and the other 2000mm from the welding table.

Figure 3: **Examples of scenes generated by GPT-4o with and without `SceneGenAgent`.**

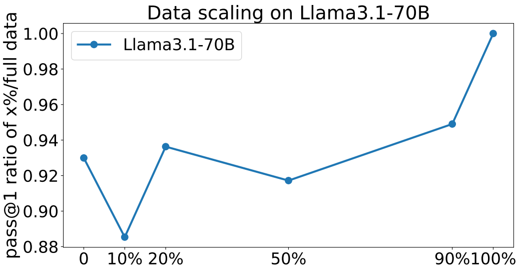


Figure 4: **Performance over different proportions of training data.** The x-axis represents data proportion and the y-axis represents the pass@1 *ratio* between training with a proportion of data and with the full dataset.

4.3.2 Necessity of guidance in code generation

During the code generation phase, we provide the LLM with coding guidance related to Tecnomatix APIs. We assess whether the models require targeted guidance to utilize the APIs. We experiment by removing the API-specific instructions for object placement and instead instructing the LLMs to write Tecnomatix-based code to set the coordinates and directions of objects, drawing on their inherent knowledge regarding the Tecnomatix framework. This allows us to evaluate the models' performance without explicit coding guidance. We present this prompt for code generation in Appendix A.7.2.

We conduct these tests on GPT-4o and Llama3.1-70B without fine-tuning. **All generated code fails to compile without API guidance, yielding 0 in pass@1 for both models.** Rather than invoking the correct methods, the models often resort to fabricated methods not supported by Tecnomatix. This finding reveals that LLMs struggle to produce functional code without framework-specific instruction and validates our approach, emphasizing the necessity of API guidance for generating code.

4.3.3 Influence of data scaling

Due to the limited availability of high-quality, domain-specific data in industrial applications, we investigate the impact of the quantity of training data on performance to explore how much data is needed for scene generation capability (Zhang et al., 2024a,b). We conduct experiments using varying proportions of the original training data and assess the agent's performance across different data scales. We retrain Llama3.1-70B on different subsets of the original `SceneInstruct`, with 10%, 20%, 50%, and 90% ratios, and compare the performance to that obtained with the full dataset.

The results are in Figure 4. The pass@1 fluctuates as the ratio rises to 50%, at times dropping below that of the model without training. The performance moderately increases at the ratio of 90%. This result suggests that, despite introducing task-relevant information, the limited data size hinders the model's generalization across scene layouts and may degrade performance. A critical threshold of training data is required to maximize the agent's ability to generalize and handle the diversity of industrial scene generation. Our experiment confirms that the amount of training data is vital in achieving high-precision scene generation.

5 Related Work

Generating scenes from textual descriptions has broad applicability across various fields such as indoor environments (Chang et al., 2017; Song et al., 2023), road traffic (Kar et al., 2019; Tan et al., 2021), and city planning (Lipp et al., 2011; Qin et al., 2024). A significant body of work focuses on generating structured 3D scenes by arranging modular and interactive objects with LLMs based on textual inputs. Unlike approaches that directly synthesize 3D representations such as radiance fields and point clouds (Bai et al., 2023; Cohen-Bar et al., 2023; Po and Wetzstein, 2024), these methods utilize pre-defined object assets, enabling the integration of generated scenes into downstream applications like programming in software. Studies using LLMs to generate 3D scenes introduce various techniques to control scene components. LayoutGPT (Feng et al., 2024), 3D-GPT (Sun et al., 2023), and SceneX (Zhou et al., 2024a) directly generate configurations of the scene, assuming that the LLMs are reliable for accurate arrangement. Other studies adjust the scene's spatial layout to better adhere to the text. LLMR (De La Torre et al., 2024) relies on validation to refine the LLM's layout planning. Some methods treat layout information in the description as constraints. SceneCraft (Hu et al., 2024) models these constraints using learnable scoring functions and optimizes these scores. Holodeck (Yang et al., 2024b) employs depth-first search on a relational constraint graph to find a placement satisfying the constraints. GALA3D (Zhou et al., 2024b) expresses the constraints with 3D Gaussian splatting representations to refine the layout. `SceneGenAgent` differentiates from them by regarding layout information as computable and refinable data structures, enabling precise placement through calculation and reasoning.

6 Conclusion

This paper introduces `SceneGenAgent`, an LLM-based agent that automates the precise generation of industrial scenes, translating descriptions into C# code using Tecnomatix API. We propose a layout analysis pipeline within `SceneGenAgent` that ensures accurate spatial positioning and compliance with complex numerical constraints in industrial layouts. `SceneGenAgent` uses a structured, calculable format to represent spatial specifications and infer overall layout planning through computation, iterative verification, and refinement. Furthermore, we fine-tune open-source LLMs with our constructed dataset `SceneInstruct` to improve their performance in `SceneGenAgent`, approaching the capabilities of API-based models. We believe that `SceneGenAgent` can significantly enhance the efficiency and accuracy of industrial manufacturing and provide insights for further advancements in the scene generation field.

Limitations

Broader applications. Our experiments predominantly focus on object assets and textual descriptions within industrial contexts. However, our proposed method for precise layout generation is not limited to such environments and holds potential for broader application in general scene generation tasks. Specifically, this method can enhance the quantitative accuracy of generated results across various domains. We plan to expand this work to more generalized scene generation by incorporating standardized benchmarks and robust evaluation methodologies in the future.

Evaluation methods. Due to the lack of automated tool suites for industrial software, our evaluation relies on manual evaluation, which is labor-intensive and inherently susceptible to subjective bias. Future work could explore developing automated evaluation methods to assess the generated scene's compliance with quantitative constraints, particularly in industrial software applications.

Acknowledgments

This work is supported by the Natural Science Foundation of China (NSFC) 62495063, 62276148, and research grants from Siemens and China Datang Corporation Ltd. The GPU compute used in this work is sponsored by Zhipu AI. We would like to thank the members of Siemens Ltd. for their technical support and assistance in the evaluation. The corresponding author: Yuxiao Dong (yuxiaod@tsinghua.edu.cn).

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Haotian Bai, Yuanhuiyi Lyu, Lutao Jiang, Sijia Li, Haonan Lu, Xiaodong Lin, and Lin Wang. 2023. Componerf: Text-guided multi-object compositional nerf with editable 3d scene layout. *arXiv preprint arXiv:2303.13843*.

Andrei Z Broder. 2000. Identifying and filtering near-duplicate documents. In *Annual symposium on combinatorial pattern matching*, pages 1–10. Springer.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Angel X Chang, Mihail Eric, Manolis Savva, and Christopher D Manning. 2017. Sceneseer: 3d scene design with natural language. *arXiv preprint arXiv:1703.00050*.

Baian Chen, Chang Shu, Ehsan Shareghi, Nigel Collier, Karthik Narasimhan, and Shunyu Yao. 2023. Fireact: Toward language agent fine-tuning. *arXiv preprint arXiv:2310.05915*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Dana Cohen-Bar, Elad Richardson, Gal Metzer, Raja Giryes, and Daniel Cohen-Or. 2023. Set-the-scene: Global-local training for generating controllable nerf scenes. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2920–2929.

Fernanda De La Torre, Cathy Mengying Fang, Han Huang, Andrzej Banburski-Fahey, Judith Amores Fernandez, and Jaron Lanier. 2024. Llmr: Real-time prompting of interactive worlds using large language models. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, pages 1–22.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Weixi Feng, Wanrong Zhu, Tsu-jui Fu, Varun Jampani, Arjun Akula, Xuehai He, Sugato Basu, Xin Eric Wang, and William Yang Wang. 2024. Layoutgpt: Compositional visual planning and generation with large language models. *Advances in Neural Information Processing Systems*, 36.

Huan Fu, Bowen Cai, Lin Gao, Ling-Xiao Zhang, Jiaming Wang, Cao Li, Qixun Zeng, Chengyue Sun, Rongfei Jia, Binqiang Zhao, et al. 2021. 3d-front: 3d furnished rooms with layouts and semantics. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 10933–10942.

Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna Wallach, Hal Daumé Iii, and Kate Crawford. 2021. Datasheets for datasets. *Communications of the ACM*, 64(12):86–92.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Diego Rojas, Guanyu Feng, Hanlin Zhao, Hanyu Lai, et al. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations*.

Ziniu Hu, Ahmet Iscen, Aashi Jain, Thomas Kipf, Yisong Yue, David A Ross, Cordelia Schmid, and Alireza Fathi. 2024. Scenecraft: An llm agent for synthesizing 3d scenes as blender code. In *Forty-first International Conference on Machine Learning*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Kai Dang, et al. 2024. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*.

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. 2023. Mistral 7b. *arXiv preprint arXiv:2310.06825*.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*.

Amlan Kar, Aayush Prakash, Ming-Yu Liu, Eric Cameracci, Justin Yuan, Matt Rusiniak, David Acuna, Antonio Torralba, and Sanja Fidler. 2019. Meta-sim: Learning to generate synthetic datasets. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4551–4560.

Markus Lipp, Daniel Scherzer, Peter Wonka, and Michael Wimmer. 2011. Interactive modeling of city layouts using layers of procedural content. In *Computer Graphics Forum*, volume 30, pages 345–354. Wiley Online Library.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2024. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36.

Ryan Po and Gordon Wetzstein. 2024. Compositional 3d scene generation using locally conditioned diffusion. In *2024 International Conference on 3D Vision (3DV)*, pages 651–663. IEEE.

Yiming Qin, Nanxuan Zhao, Jiale Yang, Siyuan Pan, Bin Sheng, and Rynson WH Lau. 2024. Urbanevolver: function-aware urban layout regeneration. *International Journal of Computer Vision*, pages 1–20.

Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. 2021. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PMLR.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. 2023. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*.

Noah Shinn, Beck Labash, and Ashwin Gopinath. 2023. Reflexion: an autonomous agent with dynamic memory and self-reflection. *arXiv preprint arXiv:2303.11366*.

Liangchen Song, Liangliang Cao, Hongyu Xu, Kai Kang, Feng Tang, Junsong Yuan, and Yang Zhao. 2023. Roomdreamer: Text-driven 3d indoor scene synthesis with coherent geometry and texture. *arXiv preprint arXiv:2305.11337*.

Chunyi Sun, Junlin Han, Weijian Deng, Xinlong Wang, Zishan Qin, and Stephen Gould. 2023. 3d-gpt: Procedural 3d modeling with large language models. *arXiv preprint arXiv:2310.12945*.

Shuhan Tan, Kelvin Wong, Shenlong Wang, Sivabalan Manivasagam, Mengye Ren, and Raquel Urtasun. 2021. Scenegen: Learning to generate realistic traffic scenes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 892–901.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*.

Wei Wang, Dan Zhang, Tao Feng, Boyan Wang, and Jie Tang. 2024. Battleagentbench: A benchmark for evaluating cooperation and competition capabilities of language models in multi-agent systems. *arXiv preprint arXiv:2408.15971*.

Zihao Wang, Shaofei Cai, Guanzhou Chen, Anji Liu, Xiaojian Ma, Yitao Liang, and Team CraftJarvis. 2023. Describe, explain, plan and select: interactive planning with large language models enables open-world multi-task agents. In *Proceedings of the 37th International Conference on Neural Information Processing Systems*, pages 34153–34189.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. 2024a. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In *The Twelfth International Conference on Learning Representations*.

Yifan Xu, Xiao Liu, Xueqiao Sun, Siyi Cheng, Hao Yu, Hanyu Lai, Shudan Zhang, Dan Zhang, Jie Tang, and Yuxiao Dong. 2024b. Androidlab: Training and systematic benchmarking of android autonomous agents. *arXiv preprint arXiv:2410.24024*.

Jianing Yang, Xuweiyi Chen, Shengyi Qian, Nikhil Madaan, Madhavan Iyengar, David F Fouhey, and Joyce Chai. 2024a. Llm-grounder: Open-vocabulary 3d visual grounding with large language model as an agent. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 7694–7701. IEEE.

Yue Yang, Fan-Yun Sun, Luca Weihs, Eli VanderBilt, Alvaro Herrasti, Winson Han, Jiajun Wu, Nick Haber, Ranjay Krishna, Lingjie Liu, et al. 2024b. Holodeck: Language guided generation of 3d embodied ai environments. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16227–16237.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36.

Aohan Zeng, Mingdao Liu, Rui Lu, Bowen Wang, Xiao Liu, Yuxiao Dong, and Jie Tang. 2023a. Agenttuning: Enabling generalized agent abilities for llms. *arXiv preprint arXiv:2310.12823*.

Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, et al. 2023b. Glm-130b: An open bilingual pre-trained model. In *The Eleventh International Conference on Learning Representations*.

Weihao Zeng, Can Xu, Yingxiu Zhao, Jian-Guang Lou, and Weizhu Chen. 2024. Automatic instruction evolving for large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 6998–7018, Miami, Florida, USA. Association for Computational Linguistics.

Dan Zhang, Ziniu Hu, Sining Zhoubian, Zhengxiao Du, Kaiyu Yang, Zihan Wang, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024a. Sciglm: Training scientific language models with self-reflective instruction annotation and tuning. *arXiv preprint arXiv:2401.07950*.

Dan Zhang, Sining Zhoubian, Min Cai, Fengzu Li, Lekang Yang, Wei Wang, Tianjiao Dong, Ziniu Hu, Jie Tang, and Yisong Yue. 2025. Datascibench: An llm agent benchmark for data science. *arXiv preprint arXiv:2502.13897*.

Dan Zhang, Sining Zhoubian, Yisong Yue, Yuxiao Dong, and Jie Tang. 2024b. Rest-mcts*: Llm self-training via process reward guided tree search. *arXiv preprint arXiv:2406.03816*.

Mengqi Zhou, Jun Hou, Chuanchen Luo, Yuxi Wang, Zhaoxiang Zhang, and Junran Peng. 2024a. Scenex: Procedural controllable large-scale scene generation via large-language models. *arXiv preprint arXiv:2403.15698*.

Shuyan Zhou, Uri Alon, Frank F Xu, Zhengbao Jiang, and Graham Neubig. 2023. Docprompting: Generating code by retrieving the docs. In *The Eleventh International Conference on Learning Representations*.

Xiaoyu Zhou, Xingjian Ran, Yajiao Xiong, Jinlin He, Zhiwei Lin, Yongtao Wang, Deqing Sun, and Ming-Hsuan Yang. 2024b. Gala3d: Towards text-to-3d complex scene generation via layout-guided generative gaussian splatting. *arXiv preprint arXiv:2402.07207*.

A Appendix

A.1 Comparison of `SceneGenAgent` with Holodeck

To facilitate a comparative analysis with `SceneGenAgent`, we select Holodeck (Yang et al., 2024b), a recently developed scene generation framework with publicly available source code. Specifically, we integrate Holodeck's placement planning module into the `SceneGenAgent`'s framework by replacing the layout extraction and placement assignment components of `SceneGenAgent` with the corresponding modules from Holodeck. We directly utilize Holodeck's placement planning code, making minimal modifications to the scene configurations and examples in the prompts to align with the requirements of industrial scene generation tasks, while leaving the remainder of the code unchanged. To evaluate performance, we compare the original `SceneGenAgent` and the modified version incorporating Holodeck's placement planning, employing Llama3.1-70B without fine-tuning as the backbone.

The pass@1 results for both variants are summarized in Table 4. The results indicate that the original `SceneGenAgent` substantially outperforms the version employing Holodeck's placement planning module. The substantial performance gap across all categories, particularly in geometric arrangement and positional details, highlights Holodeck's limitations in addressing the quantitative spatial constraints critical to industrial scene generation. These results underscore the effectiveness of `SceneGenAgent`'s layout analysis pipeline, demonstrating that previous scene generation methods, while effective in general domains, are inadequate for the specialized demands of industrial scene generation.

A.2 Datasheet

We provide the datasheet of `SceneInstruct` following the recommendations of Gebru et al. (2021).

A.2.1 Motivation

For what purpose was the dataset created? The dataset was created for fine-tuning language models to enhance their performance in industrial scene generation.

Who created the dataset (e.g., which team, research group) and on behalf of which entity (e.g., company, institution, organization)? The dataset was created by the authors of this paper.

A.2.2 Compositions

What do the instances that comprise the dataset represent (e.g., documents, photos, people, countries)? The instances that comprise the dataset represent the action trajectories of SceneGenAgent, a language model-based agent. The actions are of three types: assign placement, verify placement, and reassign placement.

How many instances are there in total (of each type, if appropriate)? There are 21756 instances in total. 20688 instances are for training, including 5704 for assigning placements, 12397 for verifying placements, and 2587 for reassigning placements. 1068 instances are for validation, including 300 for assigning placements, 636 for verifying placements, and 132 for reassigning placements.

Does the dataset contain all possible instances or is it a sample (not necessarily random) of instances from a larger set? The dataset is synthesized using language models and contains all possible instances we can sample.

What data does each instance consist of? Each instance consists of the input and output of a language model on assigning, verifying, and reassigning placement tasks, as described in Section 3.2.1.

Is there a label or target associated with each instance? Yes, every instance contains a model output as the target.

Is any information missing from individual instances? No.

Are there recommended data splits (e.g., training, development/validation, testing)? Yes, as described in Section 3.2.1.

Are there any errors, sources of noise, or redundancies in the dataset? Potential errors may be introduced by the language models used for data synthesis during reasoning and calculation.

Is the dataset self-contained, or does it link to or otherwise rely on external resources (e.g., websites, tweets, other datasets)? The dataset is self-contained.

Does the dataset contain data that might be considered confidential (e.g., data that is protected by legal privilege or by doctor-patient confidentiality, data that includes the content of individuals' non-public communications)? No.

Does the dataset contain data that, if viewed directly, might be offensive, insulting, threatening, or might otherwise cause anxiety? No.

Table 4: **Results of the original `SceneGenAgent` and `SceneGenAgent` with Holodeck's placement planning approach.**

`SceneGenAgent`'s placement planning approach	pass@1					
	Geo.	Pos.	Quant.	Comp.	Fuzz.	Overall
original	64.0	80.0	76.7	64.0	95.0	73.0
from Holodeck	21.3	11.1	23.3	24.0	90.0	28.0

A.2.3 Collection Process

How was the data associated with each instance acquired? The descriptions as the agent's input are synthesized by iteratively rewriting 20 hand-written descriptions with Llama3.1-70B. The inputs and outputs of the instances are acquired by feeding the synthesized descriptions into the agent and collecting the inputs and outputs from multiple stages of the agent.

What mechanisms or procedures were used to collect the data (e.g., hardware apparatuses or sensors, manual human curation, software programs, software APIs)? The data is collected using Llama3.1-70B, GPT-4 model API, and Llama3.1-8B for generation.

Over what timeframe was the data collected? August 2024.

Were any ethical review processes conducted (e.g., by an institutional review board)? No.

A.2.4 Preprocessing/cleaning/labeling

Was any preprocessing/cleaning/labeling of the data done (e.g., discretization or bucketing, tokenization, part-of-speech tagging, SIFT feature extraction, removal of instances, processing of missing values)? Yes, as described in Section 3.2.1 of this paper.

Was the "raw" data saved in addition to the preprocessed/cleaned/labeled data? No.

Is the software that was used to preprocess/clean/label the data available? Yes.

A.2.5 Uses

Has the dataset been used for any tasks already? Yes, as described in Section 3.2.2 of this paper.

Is there a repository that links to any or all papers or systems that use the dataset? Yes.

What (other) tasks could the dataset be used for? N/A.

Is there anything about the composition of the dataset or the way it was collected and preprocessed/cleaned/labeled that might impact future uses? No.

A.2.6 Distribution

Will the dataset be distributed to third parties outside of the entity (e.g., company, institution, organization) on behalf of which the dataset was created? Yes, the dataset is publicly distributed.

How will the dataset be distributed (e.g., tarball on website, API, GitHub)? The dataset is publicly available on GitHub.

When will the dataset be distributed? The dataset has been distributed since October 2024.

Will the dataset be distributed under a copyright or other intellectual property (IP) license, and/or under applicable terms of use (ToU)? The dataset is distributed under an Apache-2.0 license.

Have any third parties imposed IP-based or other restrictions on the data associated with the instances? No.

Do any export controls or other regulatory restrictions apply to the dataset or to individual instances? No.

A.2.7 Maintenance

Who will be supporting/hosting/maintaining the dataset? The authors of this paper.

How can the owner/curator/manager of the dataset be contacted (e.g., email address)? Via the authors' email addresses or GiHub Issues.

Is there an erratum? Not at this time.

Will the dataset be updated (e.g., to correct labeling errors, add new instances, delete instances)? Yes, when labeling errors are found in the dataset.

Will older versions of the dataset continue to be supported/hosted/maintained? Yes.

If others want to extend/augment/build on/contribute to the dataset, is there a mechanism for them to do so? Yes, by creating pull requests on GitHub.

A.3 `SceneInstruct` Data Examples

Assign placement

Input:

~~~
You are given a description of a workstation
wherein a series of objects exist, with their
respective positions mentioned in the form of
coordinates and relative positioning to one
another. The description is as follows:
```
One Welding Table for spot welding, with a
Turntable at [1500, 2500, 0], is equipped with
an ABB Robot IRB6600, positioned 2.6 meters to
the right and 2.5 meters back from the
Turntable.
```
The objects mentioned in the description are:
[”Welding Table”, ”Turntable”, ”ABB Robot
IRB6600”]
Their known positions are as follows:
```
[
    {
        ”name”: ”Turntable”,
        ”position”: ”[1500, 2500, 0]”,
        ”orientation”: ”0”
    }
]
```
Their relative positions are as follows:
```
[
    {
        ”object 1”: ”ABB Robot IRB6600”,
        ”relation”: ”2.6 meters to the right and
         2.5 meters back”,
        ”object 2”: ”Turntable”
    }
]
```
For directions and orientations, consider the
front as the positive direction of the x-axis,
the back as the negative direction of the x-
axis, the left as the positive direction of the
 y-axis, and the right as the negative
direction of the y-axis.
The orientation of an object is what direction
the object faces, including an angle rotated
counterclockwise on the ground in degrees or
towards other objects. For example, rotating
for 0 degrees is to face the front, 90 degrees
is to face the left, 180 degrees is to face the
 back, and 270 degrees is to face the right.
Based on the description and position
information, you should assign coordinates for
every objects. The coordinates are three values
 in millimeters in the form of ”[x, y, 0]” with
 no unit after the brackets, with x and y
representing the components of the coordinate
on the x-axis and y-axis respectively.
You should do the following:
1. Rewrite every relative position into the
increment of one object's coordinate relative
to another object's coordinate.
Write the increment in specific values [+-x, +-
y, 0] instead of placeholders. One relative
position should only be listed once.
If you need to set distances or directions
yourself, the Euclidean distances between
objects should generally be greater than 1
meter, which is 1000 in coordinates, unless
stated otherwise in the description.
2. For each relative position, if one object
has a declared coordinate, calculate the
coordinate of the other object. Then, list the
positional information of each object,
including coordinates and orientations.
You should first perform calculations to get
the coordinates in the analysis part, then come
 up with the final positions.
3. For the remaining objects without
coordinates, assign valid coordinates for them
based on the restrictions of object position,
direction, orientation, and relation.
The coordinate values should generally be from
[-5000, 0] or [0, 5000] intervals unless stated
 otherwise in the description.
The Euclidean distances between objects should
generally be greater than 1 meter, which is
1000 in coordinates, unless stated otherwise in
 the description.
All objects are on the ground with no vertical
relative positions.
To avoid overlapping, coordinates assigned in
this step should be different from existing
objects and each other, with the exception of
Guarding which surrounds the objects and whose
position represents its center which is not
occupied.
You should first perform calculations to get
the coordinates and deduce whether the
coordinates of all objects meet the above
conditions in the analysis part, then come up
with the final positions.
After this step, every object in the
description should have a coordinate.
You should list the positional information,
including coordinates and orientations, of all
objects in Step 3.
You must not skip any step.
You should return in the following format:
```
#Step 1: Rewrite Relative Position#
Analysis: <analysis>
New Relative Positions:
[
    {
        ”object 1”: ”<object name 1>”,
        ”relation”: ”[+-x, +-y, 0]”,
        ”object 2”: ”<object name 2>”
    },
    ...
]

#Step 2: Calculate Coordinates#
Analysis: <analysis and calculation>
Positions:
[
    {
        ”name”: ”<object name>”,
        ”position”: ”[x, y, 0]”,
        ”orientation”: ”<orientation>”
    },
    ...
]

#Step 3: Assign Positions#
Analysis: <analysis and calculation>
Positions:
[
    {
        ”name”: ”<object name>”,
~~~

~~~
        ”position”: ”[x, y, 0]”,
        ”orientation”: ”<orientation>”
    },
    ...
]
```
Do not say anything else.
~~~

Output:

````
#Step 1: Rewrite Relative Position#
Analysis: Given that the direction towards the
right is considered negative on the y-axis and
the direction back is considered negative on
the x-axis, the ABB Robot IRB6600's position
relative to the Turntable can be precisely
defined in terms of coordinate changes. As it
is 2.6 meters to the right, this translates to
-2600 on the y-axis, and being 2.5 meters back
translates to -2500 on the x-axis, considering
the origin at the Turntable's position [1500,
2500, 0].
New Relative Positions:
[
    {
        ”object 1”: ”ABB Robot IRB6600”,
        ”relation”: ”[-2500, -2600, 0]”,
        ”object 2”: ”Turntable”
    }
]

#Step 2: Calculate Coordinates#
Analysis: Given that the Turntable is located
at [1500, 2500, 0], applying the relative
position to find the ABB Robot IRB6600's
position would involve subtracting 2500 from
the x coordinate and 2600 from the y coordinate
 of the Turntable's position.
Positions:
[
    {
        ”name”: ”Turntable”,
        ”position”: ”[1500, 2500, 0]”,
        ”orientation”: ”0”
    },
    {
        ”name”: ”ABB Robot IRB6600”,
        ”position”: ”[-1000, -100, 0]”,
        ”orientation”: ”0”
    }
]

#Step 3: Assign Positions#
Analysis: With both the Turntable and the ABB
Robot IRB6600's positions now known, the
Welding Table's position needs to be set while
ensuring it respects the minimum distance
criterion from other objects. Given its absence
 in the relative position description, a new
position must be chosen. Considering the
spatial layout, placing it at a considerable
distance to avoid collision and maintain
operational efficiency is optimal. A valid
choice is ensuring it's more than 1000 mm (1
meter) away from both the Turntable and the ABB
 Robot.
Positions:
[
    {
        ”name”: ”Turntable”,
        ”position”: ”[1500, 2500, 0]”,
        ”orientation”: ”0”
    },
    {
        ”name”: ”ABB Robot IRB6600”,
        ”position”: ”[-1000, -100, 0]”,
        ”orientation”: ”0”
    },
    {
        ”name”: ”Welding Table”,
        ”position”: ”[0, 4500, 0]”,
        ”orientation”: ”0”
    }
]
````

Check placement

Input:

~~~
You are given a description of a workstation,
which is used to build a scene in Process
Simulate (PS).
The description includes several objects and
their positions. The description is as follows:
```
One Welding Table for spot welding, with a
Turntable at [1500, 2500, 0], is equipped with
an ABB Robot IRB6600, positioned 2.6 meters to
the right and 2.5 meters back from the
Turntable.
```
For directions and orientations, consider the
front as the positive direction of the x-axis,
the back as the negative direction of the x-
axis, the left as the positive direction of the
 y-axis, and the right as the negative
direction of the y-axis.
The orientation of an object is what direction
the object faces, including an angle rotated
counterclockwise on the ground in degrees or
towards other objects. For example, rotating
for 0 degrees is to face the front, 90 degrees
is to face the left, 180 degrees is to face the
 back, and 270 degrees is to face the right.
The default orientation is to rotate for 0
degrees which makes the object face the
positive direction of the x-axis.
Based on the description, we allocate positions
 for the objects. The allocated positions of
objects are as follows:
```
[
  {
    ”name”: ”Turntable”,
    ”position”: ”[1500, 2500, 0]”,
    ”orientation”: ”0”
  },
  {
    ”name”: ”Welding Table”,
    ”position”: ”[-5000, 0, 0]”,
    ”orientation”: ”0”
  },
  {
    ”name”: ”ABB Robot IRB6600”,
    ”position”: ”[-1900, 0, 0]”,
    ”orientation”: ”0”
  }
~~~

~~~~
]
```
However, the positions may represent certain
errors which need to be identified.
You should carry out the following actions to
check whether there are conflicts in object
positions:
1. Calculate each object's positional
arrangement. In case objects' positions are
described relatively, deduce the corresponding
coordinates. All objects exist in a 2D plane
with z-coordinate being 0.
2. Identify any existing errors among the
positions of the objects:
   - Constraints: The description may provide
   additional positional constraints which the
   objects should not violate. The front is the
    positive direction of the x-axis and the
   left is the positive direction of the y-axis.

   - Conflicts: The position description may
   contain inconsistencies among the objects.
   The position of an object may be calculated
   in many ways, and if the results obtained
   from different calculations are not
   consistent, there are conflicts in the
   description.
   - Overlap: The Euclidean distances between
   objects should be greater than 1 meter,
   which is 1000 in coordinates, or the objects
    will overlap with each other. Note that one
    exception is the Guarding, which surrounds
   the objects inside it and whose position
   represents its center which is not occupied.
    Also note that objects can overlap with
   each other if the description requires them
   to do so, e.g. their coordinates are
   explicitly specified.
3. Report whether there are positional errors.
If you cannot deduce whether there are any
errors - for example, when precise positions
cannot be computed, please report that there
are no errors.
You should return in the following format:
```
Relations: <Calculate the coordinates and
positional relations>
Analysis: <Based on the calculated positions,
deduce whether there are errors>
Error: <Based on the analysis, decide whether
there are errors in the description. Choose
between "Yes" and "No". If you can not
determine, choose "No">
```
Do not say anything else.
~~~~

Output:

~~~~
Relations: The Turntable is at [1500, 2500, 0].
 The ABB Robot IRB6600 is positioned 2.6 meters
 to the right (-2600 on y-axis) and 2.5 meters
back (-2500 on x-axis) from the Turntable, so
it should be at [-1000, -100, 0].

Analysis: Based on the calculations, there's a
mistake in the positions. The positions in the
provided description do not match the
calculated position of the ABB Robot IRB6600
relative to the Turntable. Also, the Welding
Table's position seems unrelated to the
provided scenario description and doesn't match
 any calculated relation from the given
information.

Error: Yes
~~~~

Reassign placement

1st round input:

~~~~
You are given a description of a workstation
wherein a series of objects exist, with their
respective positions mentioned in the form of
coordinates and relative positioning to one
another. The description is as follows:
```
One Welding Table for spot welding, with a
Turntable at [1500, 2500, 0], is equipped with
an ABB Robot IRB6600, positioned 2.6 meters to
the right and 2.5 meters back from the
Turntable.
```
The objects mentioned in the description are:
["Welding Table", "Turntable", "ABB Robot
IRB6600"]
Their known positions are as follows:
```
[
    {
        "name": "Turntable",
        "position": "[1500, 2500, 0]",
        "orientation": "0"
    }
]
```
Their relative positions are as follows:
```
[
    {
        "object 1": "ABB Robot IRB6600",
        "relation": "2.6 meters to the right and
         2.5 meters back",
        "object 2": "Turntable"
    }
]
```
For directions and orientations, consider the
front as the positive direction of the x-axis,
the back as the negative direction of the x-
axis, the left as the positive direction of the
 y-axis, and the right as the negative
direction of the y-axis.
The orientation of an object is what direction
the object faces, including an angle rotated
counterclockwise on the ground in degrees or
towards other objects. For example, rotating
for 0 degrees is to face the front, 90 degrees
is to face the left, 180 degrees is to face the
 back, and 270 degrees is to face the right.
Based on the description and position
information, you should assign coordinates for
every objects. The coordinates are three values
 in millimeters in the form of "[x, y, 0]" with
 no unit after the brackets, with x and y
representing the components of the coordinate
on the x-axis and y-axis respectively.
You should do the following:
~~~~

~~~~
1. Rewrite every relative position into the
increment of one object's coordinate relative
to another object's coordinate.
Write the increment in specific values [+-x, +-
y, 0] instead of placeholders. One relative
position should only be listed once.
If you need to set distances or directions
yourself, the Euclidean distances between
objects should generally be greater than 1
meter, which is 1000 in coordinates, unless
stated otherwise in the description.
2. For each relative position, if one object
has a declared coordinate, calculate the
coordinate of the other object. Then, list the
positional information of each object,
including coordinates and orientations.
You should first perform calculations to get
the coordinates in the analysis part, then come
 up with the final positions.
3. For the remaining objects without
coordinates, assign valid coordinates for them
based on the restrictions of object position,
direction, orientation, and relation.
The coordinate values should generally be from
[-5000, 0] or [0, 5000] intervals unless stated
 otherwise in the description.
The Euclidean distances between objects should
generally be greater than 1 meter, which is
1000 in coordinates, unless stated otherwise in
 the description.
All objects are on the ground with no vertical
relative positions.
To avoid overlapping, coordinates assigned in
this step should be different from existing
objects and each other, with the exception of
Guarding which surrounds the objects and whose
position represents its center which is not
occupied.
You should first perform calculations to get
the coordinates and deduce whether the
coordinates of all objects meet the above
conditions in the analysis part, then come up
with the final positions.
After this step, every object in the
description should have a coordinate.
You should list the positional information,
including coordinates and orientations, of all
objects in Step 3.
You must not skip any step.
You should return in the following format:
```
#Step 1: Rewrite Relative Position#
Analysis: <analysis>
New Relative Positions:
[
    {
        "object 1": "<object name 1>",
        "relation": "[+-x, +-y, 0]",
        "object 2": "<object name 2>"
    },
    ...
]

#Step 2: Calculate Coordinates#
Analysis: <analysis and calculation>
Positions:
[
    {
        "name": "<object name>",
        "position": "[x, y, 0]",
        "orientation": "<orientation>"
    },
    ...
]

#Step 3: Assign Positions#
Analysis: <analysis and calculation>
Positions:
[
    {
        "name": "<object name>",
        "position": "[x, y, 0]",
        "orientation": "<orientation>"
    },
    ...
]
```
Do not say anything else.
~~~~

1st round output (do not calculate losses for training):

~~~~
[
  {
    "name": "Turntable",
    "position": "[1500, 2500, 0]",
    "orientation": "0"
  },
  {
    "name": "ABB Robot IRB6600",
    "position": "[-1100, 0, 0]",
    "orientation": "0"
  },
  {
    "name": "Welding Table",
    "position": "[0, 3500, 0]",
    "orientation": "0"
  }
]
~~~~

2nd round input:

~~~~
Your position allocation contains the following
 error:
```
Relations: The Turntable is at [1500, 2500, 0].
 The ABB Robot IRB6600 is positioned 2.6 meters
 to the right (negative direction of the y-axis
) and 2.5 meters back (negative direction of
the x-axis) from the Turntable, resulting in
its position being [1200, 2500-2600, 0] =
[1200, -100, 0]. The Welding Table location is
not described relative to the Turntable or the
ABB Robot IRB6600, and its given position seems
 to be an absolute one without relational
context from the description provided.

Analysis: Given the positions based on the
description:
- Turntable at [1500, 2500, 0]
- ABB Robot IRB6600 calculated at [1200, -100,
0] which contradicts the given "[ -1100, 0, 0]"
- Welding Table given at "[0, 3500, 0]" without
 relative positioning stated in the description.
~~~~

~~~~
There seem to be inconsistencies in the
provided positions when compared with
calculations based on the provided descriptions
, particularly for the ABB Robot IRB6600.
Additionally, without explicit relational
descriptions for the Welding Table, its
position remains arbitrary without conflict but
 lacks verification.
```
Please write the response again, fixing the
errors.
Only generate the response. Do not say anything
 else.
~~~~

2nd round output (calculate losses for training):

~~~~
#Step 1: Rewrite Relative Position#
Analysis: Given that the direction towards the
right is considered negative on the y-axis and
the direction back is considered negative on
the x-axis, the ABB Robot IRB6600's position
relative to the Turntable can be precisely
defined in terms of coordinate changes. As it
is 2.6 meters to the right, this translates to
-2600 on the y-axis, and being 2.5 meters back
translates to -2500 on the x-axis, considering
the origin at the Turntable's position [1500,
2500, 0].
New Relative Positions:
[
    {
        "object 1": "ABB Robot IRB6600",
        "relation": "[-2500, -2600, 0]",
        "object 2": "Turntable"
    }
]

#Step 2: Calculate Coordinates#
Analysis: Given that the Turntable is located
at [1500, 2500, 0], applying the relative
position to find the ABB Robot IRB6600's
position would involve subtracting 2500 from
the x coordinate and 2600 from the y coordinate
 of the Turntable's position.
Positions:
[
    {
        "name": "Turntable",
        "position": "[1500, 2500, 0]",
        "orientation": "0"
    },
    {
        "name": "ABB Robot IRB6600",
        "position": "[-1000, -100, 0]",
        "orientation": "0"
    }
]

#Step 3: Assign Positions#
Analysis: With both the Turntable and the ABB
Robot IRB6600's positions now known, the
Welding Table's position needs to be set while
ensuring it respects the minimum distance
criterion from other objects. Given its absence
 in the relative position description, a new
position must be chosen. Considering the
spatial layout, placing it at a considerable
distance to avoid collision and maintain
operational efficiency is optimal. A valid
choice is ensuring it's more than 1000 mm (1
meter) away from both the Turntable and the ABB
 Robot.
Positions:
[
    {
        "name": "Turntable",
        "position": "[1500, 2500, 0]",
        "orientation": "0"
    },
    {
        "name": "ABB Robot IRB6600",
        "position": "[-1000, -100, 0]",
        "orientation": "0"
    },
    {
        "name": "Welding Table",
        "position": "[0, 4500, 0]",
        "orientation": "0"
    }
]
~~~~

A.4 Details on `SceneInstruct` Construction

A.4.1 Instruction Construction

During construction of descriptions as instructions in `SceneInstruct`, we devise the following rewriting methods:

- **Object addition/replacement**: Introduce new objects or substitute one or more objects in the description with 2 to 4 objects randomly picked from the object library.
- **Location specification**: Specify the locations of the objects in the description with reasonable coordinate values. We advise the LLM to choose coordinate values greater than -5000 and less than 5000. The distance between objects should be greater than 1 meter.
- **Relation specification**: Specify the relations between two objects mentioned in the description. The relation may include distance, direction, or orientation. We advise the LLM that the distance between objects should be greater than 1 meter and less than 5 meters.
- **Quantity modification**: Adjust the quantity of an object to a value between 3 and 10 if it is not assigned a placement or relations.
- **Fuzzy expressions**: Replace precise numerical relations between two objects with ambiguous or qualitative expressions. This is to get more data where the agent must set values actively, encouraging the agent to infer values independently.
- **Rephrasing**: Restate the description using alternative wording while retaining its meaning.

During each rewriting iteration, a rewriting method is selected probabilistically from a multinomial distribution. The sampling probability of each method is $(\frac{5}{24}, \frac{6}{24}, \frac{6}{24}, \frac{1}{24}, \frac{5}{24}, \frac{1}{24})$, encouraging modifications of objects and layout in the description. Descriptions are sampled from the description pool with equal probability.

A.4.2 Trajectory Collection

We collect the trajectories of `SceneGenAgent` using the constructed instruction dataset. However, collecting trajectories from all parts of `SceneGenAgent` presents significant challenges, particularly because the object retrieval part must be able to process highly arbitrary descriptions, which are difficult for the Evol-Instruct method we adopted to produce (Zeng et al., 2024). To address this, we select the parts of the agent for data collection by observing Llama3.1-70B's performance in each part. Consequently, we select the placement assignment stage for trajectory collection.

A.5 Details on Industrial Scene Generation Settings

A.5.1 Object Library

The object library we use is from the Siemens Tecnomatix Process Simulate tutorial, containing 20 different models of 10 types, including equipment and facilities like robotic arms, work tables, conveyor belts, cabinets, and guardings, for modeling and simulating workstations with different layouts.

A.5.2 Evaluation Settings

Evaluation Method. We manually inspect the scenes and compare them against their descriptions to determine their quality during evaluation. Due to the absence of testing tools and the non-uniqueness of correct generation in scene generation tasks, we can only measure the generation quality by visually inspecting the scenes. Previous studies on scene generation (Yang et al., 2024b; Hu et al., 2024; Zhou et al., 2024b,a) utilize text-to-image CLIP scores (Radford et al., 2021) to assess scene quality automatically. However, this metric is insufficient to evaluate a scene's quantitative accuracy, which is a key focus of our assessment. Although we explored the use of multimodal LLMs, such as GPT-4o, to verify whether scene images align with the quantitative requirements in the descriptions, adding grid lines in the scenes to aid in distance identification, current models struggle to extract precise quantitative information from images, making them unsuitable for evaluation. We anticipate that advancements in testing tools and multimodal LLMs may enable automated evaluation.

Collision Criterion. We require the Euclidean distance between two objects at least 1 meter to avoid collision. This criterion is introduced by engineers with expertise in industrial scene modeling and incorporated into our work to align with established real-world practices, adequate for preventing collisions. If additional factors like volume and projection area were considered, they would serve as supplementary distance constraints. In this case, the agent would adhere to these constraints, analogous to the one-meter requirement in the placement assignment stage. Specifically, this would involve converting relations to coordinate differences, allocating placement to the objects, and verifying the placement to ensure the scene's coherence.

Evaluation of Open-Source LLMs. We evaluate open-source LLMs by applying them during the placement assignment stage with Llama3.1-70B managing the remaining operations. This setting is due to the way we construct the `SceneInstruct` dataset. We refer to Llama3.1-70B's capabilities to select the parts of the agent for data collection, which is the placement assignment stage, as discussed in Section 3.2.1 and Appendix A.4.2. However, smaller models struggle in operations beyond placement assignment, making it unable to evaluate their performance. To address this, we use Llama3.1-70B to handle operations other than placement assignment for all open models to enable the evaluation of smaller models and provide evidence that fine-tuning with `SceneInstruct` improves the performance of these models in `SceneGenAgent`.

A.6 Training Hyper-Parameters

For models fine-tuned with `SceneInstruct` for the results in Table 2, their training hyper-parameters are provided in Table 5.

A.7 Prompts

A.7.1 `SceneGenAgent`

Object Retrieval

~~~
You are given a description of a workstation,
which is used to build a scene in Process
Simulate (PS).
The original description is as follows:
```
{{ prompt }}
```
~~~

Table 5: Hyper-parameters for model training with `SceneInstruct`.

Model	Peak Learning Rate	Batch Size	LoRA Rank	LoRA alpha	#Epochs
Qwen2.5-Coder-7B	1e-5	64	64	128	2
Llama3.1-8B	1e-5	256	64	128	2
GLM4-9B	1e-5	64	64	128	8
Codestral-22B	1e-4	256	64	128	8
CodeLlama-34B	1e-4	256	64	128	8
Llama3.1-70B	1e-4	256	64	128	8

~~~
You should follow the steps below:
1. First, find all objects mentioned in the
description and should appear in the
workstation according to the description.
Objects with or without specific positions
should all be listed. If the description
mentions multiple instances of the same object,
 list the object repeatedly for each instance.
For example, if "ten tables" are mentioned,
then include "table" ten times. Note that a "
station", a "workstation", or a "scene" is not
an object.
2. Then, fix the names of objects that are not
correct. The objects that may appear in the
description are from the permission list: [Kuka
 Robot KR125, Kuka Robot KR350, ABB Robot
IRB6600, YASKAWA Robot ma01800, Welding Table,
Turntable, Cabinet, ValveStand, Conveyor,
Guarding]. For every object you find, change it
 so that it meets all the requirements:
   - All objects found in the previous step is
   independent of each other. Do not merge them
    into one object.
   - If the object is not from the permission
   list, including upper/lower cases, replace
   it with an object in the permission list
   that satisfies the restrictions in the
   description.
   - If there are objects with general types,
   replace them with specific objects from the
   list.
   - The object type must satisfy the
   restrictions in the description. For example,
    the description may state that some objects
    must be the same/different.
If the description requires you to add objects
or there are no objects in the scene, add some
objects that meet the description restrictions.
Only fix the objects that are already found and
 do not add additional objects.
Return all objects after correction, including
ones that are correct and ones that are changed
 to be correct.
3. Finally, you should modify the original
description:
   - Rewrite the description like an industrial
    modeling engineer would in accurate
   wordings.
   - Include all objects found in the last step,
    including upper/lower cases.
   - All objects found in the previous step is
   independent of each other. Do not merge them
    into one object.
   - Keep the positional, directional, and
   orientational information of objects.
   - Remove the color, shape, length, height,
   and function attributes of objects.
Return the rewritten description with one
single sentence in one line and keep all
positional information.

You should return your analysis, objects and
description in the format below:
```
#Step 1: Find all objects#
Analysis: <analysis>
Objects: ["ObjA", "ObjB", ...]

#Step 2: Fix object names#
Analysis: <analysis>
Objects: ["ObjC", "ObjD", ...]

#Step 3: Rewrite description#
Analysis: <analysis>
New Description: <new description>
```
Do not say anything else.
~~~

Layout Extraction

~~~
You are given a description of a workstation,
which is used to build a scene in Process
Simulate (PS).
The description includes several objects and
their positions. The description is as follows:
```
{{ prompt }}
```
The objects mentioned in the description are: {
objects}
You should perform the following steps:
1. First, based on the provided object list,
list all objects that are mentioned in the
description and should appear in the scene
according to the description. Objects with or
without specific positions should all be listed.

If the description mentions multiple instances
of the same object, list the object name
repeatedly for each instance and assign numbers
 in the object names to tell them apart.
You should return a list of objects in the
following format:
```
Objects:
["ObjA", "ObjB 1", "ObjB 2", ...]
~~~

~~~~
```
In the example, "ObjB 1" and "Obj 2" are two
objects of the same kind. You should assign
numbers to tell them apart.
You should not return unspecified object names.
 Instead, infer them to object names in the
list.

2. Then, find all positional information of
objects and reference points directly provided
in the description. The positional information
of objects include coordinates and orientations.

The coordinates are in the form of [x, y, 0],
which should contain three values in
millimeters in the form of "[x, y, 0]" with no
unit after the brackets.
The coordinates of different objects must be
different to avoid overlapping with Guarding as
 an exception, which surrounds the objects
inside it and whose position represents its
center which is not occupied.
For directions and orientations, consider the
front as the positive direction of the x-axis,
the back as the negative direction of the x-
axis, the left as the positive direction of the
 y-axis, and the right as the negative
direction of the y-axis.
The orientation of an object is what direction
the object faces, including an angle rotated
counterclockwise on the ground in degrees or
towards other objects. For example, rotating
for 0 degrees is to face the front, 90 degrees
is to face the left, 180 degrees is to face the
 back, and 270 degrees is to face the right.
The default orientation is to rotate for 0
degrees which makes the object face the
positive direction of the x-axis.
Only list coordinates directly provided by the
description. Do not calculate coordinates by
yourself.
You should return a list of positions that
appear in the description:
```json
Positions:
[
    {
        "name": "<name>",
        "position": "<absolute position,
        including coordinate [x, y, 0]>",
        "orientation": "<orientation>"
    },
    ...
]
```
Use JSON format. <name> should either be from
the object list or be an reference point.
If no coordinates are provided, return an empty
 list.

3. Find every relative positions that are
mentioned in the description between objects or
 reference points. The relative positions may
include distances, directions, and orientations
, and should include specific distances,
directions, and orientations instead of
ambiguous descriptions if they are provided.
The distances between objects or reference
points should be outside brackets and in meters.
 Each pair of objects or reference points
should be listed at most once.
You should return a list of relative positions
that appear in the description.
```json
Relative Positions:
[
    {
        "object 1": "<name 1>",
        "relation": "<object 1 relative to
        object 2>",
        "object 2": "<name 2>"
    },
    ...
]
```
Use JSON format. Both <name 1> and <name 2>
should be either from the object list or an
reference point.
If no relative positions are found, return an
empty list.

You should first write your analysis of the
description, then return the position
information, in the following format:
```
#Step 1: Identify Objects#
Analysis: <analysis>
Objects:
[...]

#Step 2: Absolute Positions#
Analysis: <analysis>
Positions:
[
    ...
]

#Step 3: Relative Positions#
Analysis: <analysis>
Relative Positions:
[
    ...
]
```
Do not say anything else.
~~~~

Placement Assignment

~~~~
You are given a description of a workstation
wherein a series of objects exist, with their
respective positions mentioned in the form of
coordinates and relative positioning to one
another. The description is as follows:
```
{{ prompt }}
```
The objects mentioned in the description are:
{{ objects }}
Their known positions are as follows:
```
{{ positions }}
```
Their relative positions are as follows:
```
{{ relations }}
```
For directions and orientations, consider the
~~~~

~~~
front as the positive direction of the x-axis,
the back as the negative direction of the x-
axis, the left as the positive direction of the
 y-axis, and the right as the negative
direction of the y-axis.
The orientation of an object is what direction
the object faces, including an angle rotated
counterclockwise on the ground in degrees or
towards other objects. For example, rotating
for 0 degrees is to face the front, 90 degrees
is to face the left, 180 degrees is to face the
 back, and 270 degrees is to face the right.
Based on the description and position
information, you should assign coordinates for
every objects. The coordinates are three values
 in millimeters in the form of "[x, y, 0]" with
 no unit after the brackets, with x and y
representing the components of the coordinate
on the x-axis and y-axis respectively.
You should do the following:
1. Rewrite every relative position into the
increment of one object's coordinate relative
to another object's coordinate.
Write the increment in specific values [+-x, +-
y, 0] instead of placeholders. One relative
position should only be listed once.
If you need to set distances or directions
yourself, the Euclidean distances between
objects should generally be greater than 1
meter, which is 1000 in coordinates, unless
stated otherwise in the description.
2. For each relative position, if one object
has a declared coordinate, calculate the
coordinate of the other object. Then, list the
positional information of each object,
including coordinates and orientations.
You should first perform calculations to get
the coordinates in the analysis part, then come
 up with the final positions.
3. For the remaining objects without
coordinates, assign valid coordinates for them
based on the restrictions of object position,
direction, orientation, and relation.
The coordinate values should generally be from
[-5000, 0] or [0, 5000] intervals unless stated
 otherwise in the description.
The Euclidean distances between objects should
generally be greater than 1 meter, which is
1000 in coordinates, unless stated otherwise in
 the description.
All objects are on the ground with no vertical
relative positions.
To avoid overlapping, coordinates assigned in
this step should be different from existing
objects and each other, with the exception of
Guarding which surrounds the objects and whose
position represents its center which is not
occupied.
You should first perform calculations to get
the coordinates and deduce whether the
coordinates of all objects meet the above
conditions in the analysis part, then come up
with the final positions.
After this step, every object in the
description should have a coordinate.
You should list the positional information,
including coordinates and orientations, of all
objects in Step 3.
You must not skip any step.
You should return in the following format:
```
#Step 1: Rewrite Relative Position#
Analysis: <analysis>
New Relative Positions:
[
    {
        "object 1": "<object name 1>",
        "relation": "[+-x, +-y, 0]",
        "object 2": "<object name 2>"
    },
    ...
]

#Step 2: Calculate Coordinates#
Analysis: <analysis and calculation>
Positions:
[
    {
        "name": "<object name>",
        "position": "[x, y, 0]",
        "orientation": "<orientation>"
    },
    ...
]

#Step 3: Assign Positions#
Analysis: <analysis and calculation>
Positions:
[
    {
        "name": "<object name>",
        "position": "[x, y, 0]",
        "orientation": "<orientation>"
    },
    ...
]
```
Do not say anything else.
~~~

Placement Verification

~~~
You are given a description of a workstation,
which is used to build a scene in Process
Simulate (PS).
The description includes several objects and
their positions. The description is as follows:
```
{{ prompt }}
```
For directions and orientations, consider the
front as the positive direction of the x-axis,
the back as the negative direction of the x-
axis, the left as the positive direction of the
 y-axis, and the right as the negative
direction of the y-axis.
The orientation of an object is what direction
the object faces, including an angle rotated
counterclockwise on the ground in degrees or
towards other objects. For example, rotating
for 0 degrees is to face the front, 90 degrees
is to face the left, 180 degrees is to face the
 back, and 270 degrees is to face the right.
The default orientation is to rotate for 0
degrees which makes the object face the
positive direction of the x-axis.
Based on the description, we allocate positions
 for the objects. The allocated positions of
~~~

~~~
objects are as follows:
```
{{ placements }}
```
However, the positions may represent certain
errors which need to be identified.
You should carry out the following actions to
check whether there are conflicts in object
positions:
1. Calculate each object's positional
arrangement. In case objects' positions are
described relatively, deduce the corresponding
coordinates. All objects exist in a 2D plane
with z-coordinate being 0.
2. Identify any existing errors among the
positions of the objects:
   - Constraints: The description may provide
   additional positional constraints which the
   objects should not violate. The front is the
    positive direction of the x-axis and the
   left is the positive direction of the y-axis.

   - Conflicts: The position description may
   contain inconsistencies among the objects.
   The position of an object may be calculated
   in many ways, and if the results obtained
   from different calculations are not
   consistent, there are conflicts in the
   description.
   - Overlap: The Euclidean distances between
   objects should be greater than 1 meter,
   which is 1000 in coordinates, or the objects
    will overlap with each other. Note that one
    exception is the Guarding, which surrounds
   the objects inside it and whose position
   represents its center which is not occupied.
    Also note that objects can overlap with
   each other if the description requires them
   to do so, e.g. their coordinates are
   explicitly specified.
3. Report whether there are positional errors.
If you cannot deduce whether there are any
errors - for example, when precise positions
cannot be computed, please report that there
are no errors.
You should return in the following format:
```
Relations: <Calculate the coordinates and
positional relations>
Analysis: <Based on the calculated positions,
deduce whether there are errors>
Error: <Based on the analysis, decide whether
there are errors in the description. Choose
between ”Yes” and ”No”. If you can not
determine, choose ”No”>
```
Do not say anything else.
~~~

Placement Error Feedback

~~~
Your position allocation contains the following
 error:
```
{{ feedback }}
```
Please write the response again, fixing the
errors.
Only generate the response. Do not say anything
 else.
~~~

Code Generation

~~~
You are given a description of a workstation,
which is used to build a scene in Process
Simulate (PS). You should write C# code with
specific packages to build this scene.
The description is as follows:
```
{{ prompt }}
```
The objects that should be added into the scene
 are as follows: {{ objects }}
Their positions and orientations are as follows
:
```
{{ placements }}
```
You should write the complete code in the
following format:
```csharp
string rootDir = TxApplication.
SystemRootDirectory;
string weldingLibPath = Path.Combine(rootDir, ”
Welding”);
string[] weldingModels = Directory.
GetDirectories(weldingLibPath, "*.cojt”,
SearchOption.TopDirectoryOnly);

/* create model list */

foreach (string model in weldingModels)
{
    DirectoryInfo directoryInfo = new
    DirectoryInfo(model);

    /* load models */
}

Random rand = new Random();
TxPhysicalRoot txPhysicalRoot = TxApplication.
ActiveDocument.PhysicalRoot;

/* add objects into the scene */

TxApplication.RefreshDisplay();
```
where the parts surrounded by ”/* */” are the
parts you should fill.

For the ”create model list” part, you should
define Lists of DirectoryInfo, which are used
to store models in the ”add models” part. For
example, if you want a list to store robots:
```csharp
List<DirectoryInfo> robotModels = new List<
DirectoryInfo>();
```
You can name the model lists whatever you like
in a similar form.

For the ”load models” part, you should check
whether the current `directoryInfo` belongs to
the types of models that are needed in the
scene one by one and add it to corresponding
model list.
You should use the following methods to load
~~~

~~~
the objects:
{{ guidance for loading object models }}

For the "add objects into the scene" part, you
should add the objects into the scene and set
its positions. For each object in the scene,
pick the model from the model lists, put it
into the scene, and set its coordinate and
orientation.
To pick an object `obj1` from the list `
objModels`, place it at [`x`, `y`, 0], and
rotate it for `degree` degrees:
```csharp
DirectoryInfo objModel1 = objModels[rand.Next
(0, objModels.Count)];
string obj1Name = Path.
GetFileNameWithoutExtension(objModel1.Name) + "
_" + DateTime.Now.ToString("yyyy-MM-dd-HH-mm-ss
");
TxInsertComponentCreationData txInsertDataObj1
= new TxInsertComponentCreationData(obj1Name,
objModel1.FullName);
ITxComponent txComponentObject1 =
txPhysicalRoot.InsertComponent(txInsertDataObj1
);

double transXValue1 = x;
double transYValue1 = y;
double rotValue1 = degree * Math.PI / 180.0;
TxTransformation txTransTransXYRotZ = new
TxTransformation(new TxVector(transXValue1,
transYValue1, 0.0), new TxVector(0.0, 0.0,
rotValue1), TxTransformation.TxRotationType.
RPY_ZYX);
ITxLocatableObject obj1 = (ITxLocatableObject)
txComponentObject1;
obj1.AbsoluteLocation *= txTransTransXYRotZ;
```

Now, you should generate the complete code to
build this scene. Only generate the complete
code. Your code will be run directly in the
production environment, so don't omit anything.
Please strictly follow the given code snippets
to load and place objects. Do not generate a
class or a function, instead directly generate
the function body. Do not define or call custom
 classes or functions by yourself.
Please use .NET Framework 4.6.2 or below, or
the code will not run.
Do not say anything else.
~~~

A.7.2 Ablation Study

Analyze Layout without SceneGenAgent

~~~
You are given a description of a workstation,
which is used to build a scene in Process
Simulate (PS).
The original description is as follows:
```
{{ prompt }}
```
You should follow the steps below:
1. First, find all objects mentioned in the
description and should appear in the
workstation according to the description.
The objects that may appear in the description
are from the permission list: [Kuka Robot KR125
, Kuka Robot KR350, ABB Robot IRB6600, YASKAWA
Robot ma01800, Welding Table, Turntable,
Cabinet, ValveStand, Conveyor, Guarding]
If any object is not from the permission list
or is from a general type, replace it with an
object in the permission list that satisfies
the restrictions in the description, including
upper/lower cases.
If the description mentions multiple instances
of the same object, list the object repeatedly
for each instance, list the object name
repeatedly for each instance and assign numbers
 in the object names to tell them apart. Note
that a "station", a "workstation", or a "scene"
 is not an object.
You should return a list of objects in the
following format:
```
Objects:
["ObjA", "ObjB 1", "ObjB 2", ...]
```
In the example, All objects in the list are
chosen from the permission list and should
appear in the scene according to the
description. "ObjB 1" and "Obj 2" are two
objects of the same kind. You should assign
numbers to tell them apart.
2. Then, for each object in the object list
generated in the previous step, set its
placement in the scene according to the
description. The placement includes coordinates
 and an orientations.
The coordinates are in the form of [x, y, 0],
which should contain three values in
millimeters in the form of "[x, y, 0]" with no
unit after the brackets, with x and y
representing the components of the coordinate
on the x-axis and y-axis respectively.
To avoid overlapping, coordinates assigned in
this step should be different from existing
objects and each other, with the exception of
Guarding which surrounds the objects and whose
position represents its center which is not
occupied.
The coordinate values should generally be from
[-5000, 0] or [0, 5000] intervals unless stated
 otherwise in the description.
The Euclidean distances between objects should
generally be greater than 1 meter, which is
1000 in coordinates, unless stated otherwise in
 the description.
All objects are on the ground with no vertical
relative positions.
The orientation of an object is what direction
the object faces, i.e. an angle rotated
counterclockwise on the ground in degrees or
towards other objects. For example, rotating
for 0 degrees is to face the front, 90 degrees
is to face the left, 180 degrees is to face the
 back, and 270 degrees is to face the right.
The default orientation is to rotate for 0
degrees which makes the object face the
positive direction of the x-axis.
For directions and orientations, consider the
front as the positive direction of the x-axis,
the back as the negative direction of the x-
axis, the left as the positive direction of the
 y-axis, and the right as the negative
direction of the y-axis.
~~~

~~~~
You should first perform calculations to get
the coordinates and orientations and deduce
whether the placement of all objects satisfies
the description in the analysis part, then come
 up with the final placement.
You should return a list of placements that
appear in the description:
```
Placements:
[
    {
        "name": "<name>",
        "position": "<coordinate [x, y, 0]>",
        "orientation": "<orientation>"
    },
    ...
]
Use JSON format. <name> should be from the
object list.
```

You should return your analysis, objects and
placements in the format below:
```
#Step 1: Identify Objects#
Analysis: <analysis>
Objects:
[...]

#Step 2: Assign Placements#
Analysis: <analysis>
Placements:
[
    ...
]
```
Do not say anything else.
~~~~

Code Generation without Object Placement API Guidance

~~~~
You are given a description of a workstation,
which is used to build a scene in Process
Simulate (PS). You should write C# code with
Tecnomatix to build this scene.
The description is as follows:
```
{{ prompt }}
```
The objects that should be added into the scene
 are as follows: {{ objects }}
Their positions and orientations are as follows
:
```
{{ placements }}
```
You should write the complete code in the
following format:
```csharp
string rootDir = TxApplication.
SystemRootDirectory;
string weldingLibPath = Path.Combine(rootDir, "
Welding");
string[] weldingModels = Directory.
GetDirectories(weldingLibPath, "*.cojt",
SearchOption.TopDirectoryOnly);

/* create model list */

foreach (string model in weldingModels)
{
    DirectoryInfo directoryInfo = new
    DirectoryInfo(model);

    /* load models */
}

Random rand = new Random();
TxPhysicalRoot txPhysicalRoot = TxApplication.
ActiveDocument.PhysicalRoot;

/* add objects into the scene */

TxApplication.RefreshDisplay();
```
where the parts surrounded by "/* */" are the
parts you should fill.

For the "create model list" part, you should
define Lists of DirectoryInfo, which are used
to store models in the "add models" part. For
example, if you want a list to store robots:
```csharp
List<DirectoryInfo> robotModels = new List<
DirectoryInfo>();
```
You can name the model lists whatever you like
in a similar form.

For the "load models" part, you should check
whether the current `directoryInfo` belongs to
the types of models that are needed in the
scene one by one and add it to corresponding
model list.
You should use the following methods to load
the objects:
{{ guidance for loading object models }}

For the "add objects into the scene" part, you
should add the objects into the scene and set
its positions. For each object in the scene,
pick the model from the model lists, put it
into the scene, and set its coordinate and
orientation.

Now, you should generate the complete code to
build this scene. Only generate the complete
code. Your code will be run directly in the
production environment, so don't omit anything.
Please strictly follow the given code snippets
to load and place objects. Do not generate a
class or a function, instead directly generate
the function body. Do not define or call custom
 classes or functions by yourself.
Please use .NET Framework 4.6.2 or below, or
the code will not run.
Do not say anything else.
~~~~

A.8 Use Of AI Assistants

In this work, we use AI assistants to polish our writing.

A.9 Benchmark

We list all the details of the industrial scene generation benchmark in Table 6, Table 7, and Table 8.

Table 6: Benchmark details for descriptions with ID from 1 to 20.

ID	Description	Category
1	Create a welding station consisting of 3 robotic arms and a welding table. Arrange the 3 robotic arms in a straight line at equal intervals in front of the welding table.	Geometric Arrangement
2	Can you design a virtual robotic welding cell setup? I'd like it to have a robot that picks up a part from two tables (at [1500, 0, 0], [-1500, 0, 0]) and moves part to a conveyor belt about 1.5m away. The conveyor should run parallel to the two tables.	Geometric Arrangement
3	Create a layout featuring 6 robots positioned alongside two conveyor belts that are lined up end-to-end. Arrange 3 robots on each side of the conveyor belts in a straight line, with 1.5-meter intervals between each arm.	Geometric Arrangement
4	Create a layout with 9 robots arranged in a 3x3 matrix configuration. Position the arms with a 3-meter interval between each one.	Geometric Arrangement
5	How can we create a processing station where the robotic arm, work table, and conveyor belt are positioned at right angles to one another? Additionally, we need to enclose the area with protective guarding.	Geometric Arrangement
6	Please set up a three-row layout. Place 3 work tables in the first row, position 3 robots in the second row, and install 2 conveyor belts in the third row.	Geometric Arrangement
7	We need to set up an L-shaped scene. On one side, line up a series of tables in a row,spaced out by about 2 meter. On the other, put a row of cabinets, spaced out by about 1 meter.	Geometric Arrangement
8	Provide me a U-shaped simulation layout. Position a row of worktables along the base. On one side, arrange four equally spaced cabinets. On the opposite side, install a series of conveyor belts. Place an ABB robot at the center of this layout.	Geometric Arrangement
9	Picture a circular welding setup with a table in the middle and six robots around it. The robots can all reach the table easily, and there's a safety fence surrounding the whole thing.	Geometric Arrangement
10	Create a welding workstation with a central work table. Position 4 KUKA robots on each side of the table, about 2 meters away.	Geometric Arrangement
11	Create a welding workstation with a robot positioned at the center. Put 3 work tables and 1 conveyor belt around the robot, each placed in a different direction.	Geometric Arrangement
12	Create a scenario with a Kuka Robot, a welding table, guarding, and a cabinet. Pick random types from the TX root 3D library. Space these components evenly in the scenario.	Geometric Arrangement
13	Set up a U-shaped layout with 10 tables for the initial simulation scene.	Geometric Arrangement
14	Position a Kuka robot in front of a table.	Geometric Arrangement
15	Give me two conveyors, arranged in parallel.	Geometric Arrangement
16	I need a simulation model for welding processes. Place an ABB robot at the center of the scene with a welding table 2 meter in front of it. Surround the entire scene with protective fencing. Additionally, position a KUKA robot arm at coordinates [1000, 2000, 0] for subsequent processing.	Positional Details
17	I want a simulation scenario with the following layout: Position a welding table as the central element. Place a Kuka Robot KR125 2 meters in front of the table. To one side, at a distance of 4 meters from both the robot and table, arrange three cabinets in a row. Set the spacing between the first and second cabinet at 2.1 meters, and between the second and third cabinet at 1.3 meters.	Positional Details
18	Create a simulation model with this layout: Position three ABB robots. Place the first robot at coordinates [2000, 2000, 0]. Align the other two robots along the y-axis of the first robot, with a 3-meter interval between each. Then, at a distance of 5 meters along the x-axis from the first robot, position a ValveStand next to a cabinet.	Positional Details
19	I need a welding station with two KUKA robots for loading and unloading. The loading robot is at [2000, 2000, 0] and the unloading robot at [-2000, -2000, 0]. A conveyor belt transfers materials between loading and unloading positions. Additionally, two welding tables are situated near the loading and unloading robots.	Positional Details
20	Please help me create a welding workstation including 1 Kuka robot and 1 ABB Robot, separately located 2m aside from a welding table in the center of the workstation.	Positional Details

Table 7: Benchmark details for descriptions with ID from 21 to 33.

ID	Description	Category
21	Hey, could you set up a scene for me? I need an industrial welding setup with multiple models. I want a KUKA Robot KR125, a YASKAWA Robot, and 2 turntables. Can you put the KUKA at [2000, 0, 0], the YASKAWA at [0, 2000, 0], and one table at [-500, -500, 0], with another 2m behind it?	Positional Details
22	Set up a welding station with a regular table, a welding table, two KUKA robots, a value stand, and safety guarding. Place one robot 1500mm from the regular table and the other 2000mm from the welding table.	Positional Details
23	Put a pair of robots at the front of a new scene. Create a cabinet next to a turntable behind the scene.	Positional Details
24	An ABB Robot is at [2000,2000]. A Table is at [25,20,0]. A Conveyor is at [-1234mm,-2001mm,0].	Positional Details
25	Create a scene with multiple robots and cabinets and 3 worktables. Position 3 worktables with a 2.5-meter interval between each. Ensure that every table is closely accompanied by two robots and one cabinet.	Object Quantity
26	We'll need a table and five robots to set up an assembly line. Use three identical model robots, and two different models for the remaining two.	Object Quantity
27	let's set up the scene with 5 ABB robots, position 3 welding tables next to 3 of the robots. Add 2 conveyor belts near the other robots. On one side of the scene, include 2 cabinets and one standard table.	Object Quantity
28	Could you please generate a scene of a welding workstation with the following equipment: Kuka Robot KR350, ABB robot, guarding, and valve stands? Please apply a random number of each item in the scene. One important requirement is to position one of the Kuka Robots near one of the ABB robots, with a distance between them ranging from 4000 to 5000 millimeters.	Object Quantity
29	I need to create a 3D model for a robotic welding cell. It's got a welding table in the middle and two Kuka KR125 robots around it, 3 abb robots 4 meters away. Can you help me set this up in PS.	Object Quantity
30	The conveyor is positioned at coordinates [0,0,0]. 6 robots are uniformly distributed in front of the conveyor.	Object Quantity
31	Design a complex welding station incorporating four robotic units: two ABB robots and two KUKA robots. Position the ABB robots at coordinates [2000,0,0] and [-2000,0,0], tasking them with product part loading and unloading operations. Place the KUKA robots at [0, 2000, 0] and [0, -2000,0], assigning them to perform collaborative spot welding tasks; add a guarding fence around the station, install a centrally positioned welding table and two conveyor systems to transport parts from the table. Additionally, place two equipment cabinets adjacent to the robots.	Composite Description
32	These robotic arms are positioned in a triangular formation within the welding workstation, allowing them to work collaboratively on complex welding tasks. Each machine is equipped with advanced sensors and controllers for precise movement and coordination. The Kuka Robot KR350, located at [0, 0, 0], handles heavy-duty welding, while the ABB Robot IRB6600 at [2500, 2500, 0] assists with high-speed operations. The YASKAWA Robot MA01800 at [0, 2500, 0] focuses on intricate welding tasks. Together, these machines ensure high efficiency, accuracy, and flexibility for various welding operations.	Composite Description
33	The proposed layout consists of a process simulation model featuring an U-shaped guarding structure, centrally located at the coordinates [1500, 1500, 0]. Inside the guarding, a square welding table and a cabinet are positioned, maintaining a 1.5-meter distance between them to ensure proper workspace and accessibility. The guarding serves to protect this working area, while allowing for efficient movement within the enclosed space. Adjacent to the front side of the guarding, there is a bar-type conveyor system. Positioned alongside the conveyor is a valve stand and two robots, strategically positioned to enhance production flow. The overall design ensures a logical and organized layout, optimizing both safety and functionality within the workspace.	Composite Description

Table 8: Benchmark details for descriptions with ID from 34 to 40.

ID	**Description**	**Category**
34	A welding workstation needs to be set up with three robotic manipulators and additional equipment. The main robots are: Kuka Robot KR350 (Machine No. 1) ABB Robot IRB6600 (Machine No. 2) YASKAWA Robot MA01800 (Machine No. 3) These robots are arranged in the following positions: Machine No. 1: [0, 0, 0] Machine No. 2: [2000,1500, 0] Machine No. 3: [-2000, 2500, 0] To complete the workstation, we need to add more equipment: A conveyor belt is placed at [250, -1250, 0] to move materials. A control cabinet is set up at [-200, -3250, 0] to manage the system. A welding table is positioned at [0, 1500, 0] for the robots to work on. A safety fence surrounds the entire area. This setup creates a complete welding workstation with robots, material handling, and safety features.	Composite Description
35	I need to create a simulation scenario with the following steps: 1. Load a welding table 2. Generate two robots beside it 3. Protect the area with safety fencing 4. Use ABB IRB6600 models for both robots 5. Place two conveyor belts and three cabinets inside the fence 6. Replace one of the robots with a KUKA model 7. Remove one cabinet and position the remaining two next to the robots 8. Ensure the welding table is at the center of the scene, then finalize the layout design	Composite Description
36	Could you give an example of a real-world welding scenario in industry?	Fuzzy Description
37	Build a welding workstation scene with 5 tables, and could you randomly add some typical items based on what you know about welding?	Fuzzy Description
38	Please add robots, tables, and conveyors to create a welding scene. You may randomly determine the types, quantities, and spatial arrangements of these elements.	Fuzzy Description
39	Hi, can you set up a welding scene for me? I need an initial environment for a welding use case, but without any robots.	Fuzzy Description
40	Could you create an industrial welding workstation scene using the existing assets from our library?	Fuzzy Description