SkillGraph Documentation

AIDF Project. PI Liu.

1 Overview

Many manufacturing tasks, especially product assembly, are still performed by humans because robot automation for such tasks is difficult. Assembly involves complex coordination among multiple robots, precise motion sequences, and adaptation to new setups — challenges that fixed, hard-coded programs cannot easily handle.

This project develops an **AI-based System Integrator** to make robotic assembly more intelligent and flexible. The goal is to create a framework that allows robots to automatically plan and execute multi-robot assembly and disassembly tasks while managing the knowledge and data produced in the process.

The main focus this year is to build **the foundation of this framework**, which includes:

1. An ontology that formally describes robots, objects, environments, tasks, skills, and metrics — we name it as **SkillGraph**. The SkillGraph represents how individual robot skills are related, what resources they require, and what motion data or trajectories are available for execution.
2. **An automatic planner** that generates assembly sequences and assigns skills to robots using information from the ontology.
3. **Integration** of pre-existing modules such as robot APIs and data services to connect the ontology and planner with real or simulated robotic systems.

The implementation begins with the LEGO assembly task, which serves as a controlled testbed to ensure correctness and connectivity between components. The same framework will later be extended to more complex tasks such as NIST box assembly and battery disassembly.2

This effort forms a working skeleton for the AI-based System Integrator and contributes to the AI Data Foundry by providing structured, reusable data from robot assembly tasks.

Currently, our code is implemented in this GitHub-AIDF repository.

2 Skill Graph

This section provides a more detailed description of the skill graph within the overall designed framework. The skill graph serves as a centralized codebase for organizing the necessary robot- and environment-related assumptions and knowledge to enable intelligent integration and execution. The skill graph has five sections:

1. Physical Entities: *Robot*, *Object*, and *Environment*
2. Concepts: *Skill*, including *AtomicSkill* and *MetaSkill*.
3. Executables: *SkillExecutor* and *Algorithm*. The *Algorithm* includes *PlanningAlgorithm*, *ControlAlgorithm*, and *Constraint Satisfaction Generator*.
4. Evaluation Metric: *Metric*, including *Constraint Evaluator*, and *Condition Evaluator*.
5. Date Structures: *SkillParam*, *SkillCondition*, *TaskParam*, and *State* which is constituted by *RobotState*, and *Env State*.

Besides, we also illustrate the input to the SkillGraph, symbolized as *Task*. The skill graph is demonstrated in fig. 1. The code implementation of SkillGraph is at AIDF-SkillGraph.

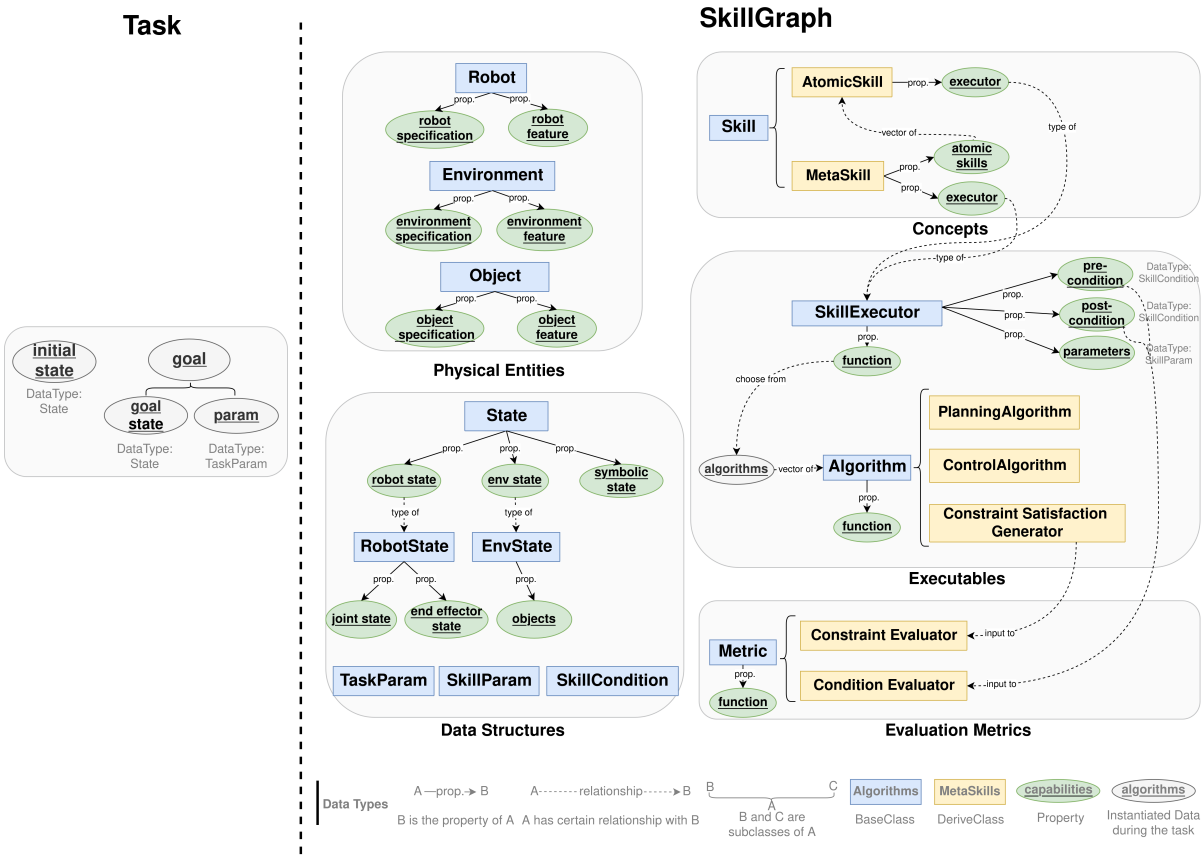

Figure 1: Symbolic Representation of Skill Graph

2.1 Data Types and Relationships in Skill Graph

There are several data types in fig. 1:

1. BaseClass: *Task*, *Robot*, *Object*, *Environment*, *Skill*, *SkillExecutor*, *Algorithm*, *Metric*, *TaskParam*, *SkillParam*, *SkillCondition*, *State*, *RobotState*, and *Env State*
2. DerivedClass: *AtomicSkill* and *MetaSkill* are derived from baseclass *Skill*. *PlanningAlgorithm*, *ControlAlgorithm*, and *Constraint Satisfaction Generator* are derived from baseclass *Algorithm*. *Constraint Evaluator* and *Condition Evaluator* are derived from baseclass *Metric*.
3. Property: Each class might own certain properties, specified by a solid arrow with label "prop." on it. For example, in fig. 1, *atomic skills* and *executor* are properties of derivedclass *MetaSkill*.

Each data may have certain relationship with other data, and the relationships are specified by arrows. If the arrow points from A to B, indicating that A has a certain relationship with B. There are different types of arrows:

1. A $\overset{\text{relationship}}{\dashrightarrow}$ B indicates that A has certain relationship with B. For example, atomic skills is a vector of *AtomicSkill*, and function of *SkillExecutor* can choose from algorithms, which is a vector of *Algorithm*.
2. A $\overset{\text{prop.}}{\longrightarrow}$ B indicates that B is the property of A.

The relationships are represented as labels on the arrows, and the relationships are defined as follows:

1. A *prop.* B: A has a property B.
2. A *choose from* B: A can choose from B.
3. A *vector of* B: A is a vector of B.
4. A *type of* B: A's data type is B.

3 Main Components in SkillGraph

The code of SkillGraph is at The key components in the SkillGraph are:

- **Skill:** Contains derived classes *AtomicSkill* and *MetaSkill*, which define the basic and composite robot skills.
- **SkillExecutor:** Defined by its four properties: *function*, *pre-condition*, *post-condition*, and *parametors*.
- **Algorithm:** Comprising derived classes *PlanningAlgorithm*, *ControlAlgorithm*, and *Constraint Satisfaction Genetor*.
- **Robot:** Defined by its two properties: *robot feature* and *robot specification*.
- **Object:** Defined by its two properties: *object feature* and *object specification*.
- **Environment:** Defined by its three properties: *environment specification*, and *environment feature*.
- **Metric:** Comprising derived class *Constraint Evaluator* and *Condition Evaluator*.

To further enhance the structure and organization of the SkillGraph, we define several essential data structures:

- **TaskParam:** Represents the parameters required for task execution, can be specified by user.
- **SkillCondition**: Represents the data structure pre- and post-conditions of skills should use.
- **SkillParam**: Represents the data structure the parameters of *SkillExecutor* should use.
- **RobotState:** Captures the current state of the robot, including its joint state and end effector state.
- **EnvState:** Capture the environment state, including all objects' locations, and status.
- **State:** Captures the current system state, including robot state, environment state, and symbolic state that is interpretable.

In the following sections, we would give a detailed definition of each components and their relationships with other components, and also their implementations in the repository.

3.1 Skill

We define any robot action or robot decision that satisfies the following requirements as a skill: 1) can be executed in a real robot or a sensor, 2) has semantic meaning, and 3) can be planned on. We store those skills in base class *Skill*, and divide them into *AtomicSkill* and *MetaSkill*. Each meta skill is composed by a set of atomic skills, arranged in a sequence or in some partial order. The code implementation of Skill class is at AIDF-skill.

3.1.1 AtomicSkill

AtomicSkill is a derived class from base class *Skill*, and it has one property:

1. executor: type of *Skill Executor*.

Currently, we define five instances of *AtomicSkill*: Pick, Place, Support, Transit, and Detect. Each atomic skill has its own *executor*, whose datatype is *SkillExecutor*. Perception-only actions use the dedicated PerceptionSkill subclass, which stores ROS topic and launch information in a

The code implementation of *AtomicSkill* class is at AtomicSkill

3.1.2 MetaSkill

MetaSkill is another derived class from base class *Skill*, and has 2 properties:

1. executor: type of *Skill Executor*.
2. atomic skills: a vector of type *AtomicSkill* that constitutes current meta skill.

Each instance of *MetaSkill* is comprised of atomic skills, and each meta skill also has its own executor. Meta skills themselves are represented by the MetaSkill class, which tracks the ordered atomic skill list, the robots participating in the sequence, selected objects, and the requested composition mode (temporal, functional, or spatial). String names from the JSON configuration are mapped to the enum values inside Skill::from_string, and helper methods such as `isAtomic` and `isMeta` split the catalogue as needed.

Currently, we have five meta skills: PlaceWithSupport and Handover, PickAndPlace, PickAndPlace-WithSup, PickHandoverAndPlace to group common action sequences and reduce the complexity of planning.

3.2 Skill Executor

Skill Executor is defined as a class that wraps the logic, effect, and requirements of skills execution in the above section. It has 4 properties:

1. function: the underlying execution function of each skill would include instances from *PlanningAlgorithm*, *ControlAlgorithm*, and *Constraint Satisfaction Algorithm*.
2. pre-condition: a *SkillCondition* that measures what requirements the current state must satisfy before executing the skill.
3. post-condition: a *SkillCondition* that measure what the state would become after the execution of certain skill.
4. parameters: a *SkillParam* that specifies the parameters of the skill execution function or the pre/post-conditions.

As mentioned in section 3.1, each skill corresponds to one skill executor stored in class *Skill Executor*.

The execution workflow is defined in SkillExecutor. Each executor keeps copies of its pre- and post-conditions (`TaskParam`), an ordered list of algorithm implementations, and the last planned trajectory. Meta skills reuse the same interface via the MetaSkillExecutor, which chains several atomic executors according to the requested composition type.

3.3 Algorithm

The class *Algorithm* is a class that implements common robotics planning/control/perception algorithms. Base Class *Algorithm* have 4 derive classes: *PlanningAlgorithm*, *ControlAlgorithm*, and *Constraint Satisfaction Generator*. The code implementation is available in the Algorithm base class. It stores a type tag (Planning, Control, Perception) and a display name. As mentioned in section 3.2, each skill executor function can choose its own planning, control, and constraint satisfaction algorithms. There may be multiple algorithms suitable for each skill type. Those algorithms are stored in its corresponding derived classes.

3.3.1 PlanningAlgorithm

PlanningAlgorithm, located in here in the code, is in charge of planning robot trajectories. Currently, task-specific logic is provided in separate, domain-specific modules such as LEGO manipulation in lego_algorithms.cpp.

For example, we have two LEGO-specific planning algorithms, *LegoGraspGenerator* here and *LegoPlan* here, which implement the grasp pose generation and motion primitives for manipulating (pick, place, and etc) LEGO.

3.3.2 ControlAlgorithm

ControlAlgorithm, implemented here in the code, is in charge of tracking the planned robot trajectories.

3.4 Robot

Class *Robot* describes the physical robot and its characteristics. The class has two properties:

1. robot specification: a json file that includes the detailed specification of each robot.

2. robot feature: currently is robot capabilities, a list of instances of *Skill* that the robot can perform.

Currently, we define one instance of *Robot*: GP4. Both robots have the same capabilities of Pick, Place, Support, and Transit.

The code implementation is available at the Robot class.

3.5 Object

The Object struct models each manipulable workpiece. It stores the semantic type, current state (static, attached, supported, handover), owning robot, the parent link, the full pose, an optional attachment pose, and a geometric description (box, sphere, cylinder, or mesh with dimensions).

3.6 Environment

Class *Environment* is the abstract entity that describe the physical assembly cell and its characteristics. The class has three properties:

1. environment specification: a json file that specifies the details of the environment.
2. environment feature: describes environment features, might including object library that has a vector of *Object* that specifies all objects that can be interacted with in the environment and backend that specifies a middleware or simulator that handles the control of the environment.

Currently, we define two instances of *environment*: EnvLego for Lego manipulation, and EnvNist for NIST test case assembly.

The implementation is available here here.

3.7 Metric

Class *Metric* defines any quantitative measure used to verify any task, object, or robot requirements that may constrain skill execution. Currently, our base class *Metric* has 2 derived classes: *Constraint Evaluator*, and *Condition Evaluator*.

Metrics are declared in metrics.hpp. The base `Metric` stores the evaluator name, and the derived `ConditionEvaluator` and `ConstraintEvaluator` expose boolean checks that plug into `TaskParam`.

3.7.1 Constraint Evaluator

Constraint Evaluator is in charge of evaluating whether an instance of *Constraint Satisfaction Generator* satisfies the constraints of the given task defined by its *TaskParam*.

3.7.2 Condition Evaluator

Condition Evaluator is in charge of evaluating whether the precondition or postcondition of a given *SkillExecutor* is satisfied.

3.8 TaskParam, SkillParam, SkillCondition

These data structures are self-definable.

3.9 State

The *State* model lives in data_structure.hpp. A *State* bundles a vector of *RobotState* entries, a deep-copied *EnvState*, and the number of assembly steps already completed; Equality and hashing rely on the content of each component, so the planner can store states inside associative containers.

3.10 RobotState

The ata structure *RobotState* defined in data_structure.hpp has several properties:

1. robot id: Specifies the robot id.

2. joint state: Specifies robot joints state.
3. end effector state: Specifies the state of the end-effector.
4. attributes: open-ended mapping for extra annotations.

3.11 EnvState

The data structure Envstate consists of

1. objects: Specifies the state of objects, including their locations, directions, and other status.

EnvState owns deep clones of every tracked *Object*, ensuring that copies and assignments do not alias the same pointer. Equality tests compare the content of each object, and the *to_string* helper produces a human-readable dump. The global hash specialization lets planner data structures use *EnvState* as a key.

3.12 RobotTrajectory

The planner expresses motion plans as RobotTrajectory sequences that record the robot id, intermediate *RobotState* waypoints, timestamps, action ids, and accumulated cost. Multi-robot solutions are stored as *MRTrajectory* (a vector of per-robot trajectories).

4 Planner

4.1 Domain Agnostic Task Planner

The core planner lives in AssemblyPlanner. Its implementation (planner.cpp) performs a best-first search over *skillgraph::State* objects using a priority queue ordered by cumulative cost. The planner pulls the initial state from the SkillGraph, expands successors through *feasible_u* and *get_next_state*, and stops as soon as *at_target* reports success. The helper *get_path* routine backtracks the predecessor maps to recover both the state sequence and the corresponding skills once a solution is found.

The key factor that allows our planner to be domain independent is the availability of MetaSkills mapped to each assembly step, as well as the domain-agnostic interface of SkillGraph that provides the set of feasible skills for a given state.

After task planning, the robot trajectories can be generated with a single-agent motion planner. This allows each task to be sequentially executed, even for multiple robots. We elaborate how we integrate failure detection and safe asynchronous execution below.

4.2 Temporal Plan Graph for Execution

We use Temporal Plan Graph, also known as a TPG, to schedule asynchronous, multi-robot execution on the real robot. The implementation is provided in tpg.h. Specifically, the TPG captures all dependencies between different robots' tasks and motions and generates a partial-order graph, which can significantly reduce unnecessary wait time and is robust to execution delay and uncertainty. The TPG uses a graph-based data structure, where nodes are robot poses or skills, and edges represent precedence constraints between consecutive nodes for the same robot and different nodes for different robots.

The TPG formulation provides an easy way to execute a multi-robot plan. The TPG is hosted on a central server communicating with each robot's execution thread. Each pose node corresponds to an action moving the robot to its configuration, while each skill node corresponds to an action executing a predefined robot skill. An action can be executed safely if there are no incoming edges from unexecuted nodes. Although each TPG node has an associated timestamp, it is only used for TPG construction and ignored during execution.

If an action is safe to execute based on the TPG, the central server sends it to the robot's action queue. Each robot maintains its own controller-sensing loop and actuates the robot according to upcoming commands and its state estimation. The state estimation is also shared with the TPG, which then updates the TPG when a node is being executed or completed. If the perception algorithm detects an execution failure, this is treated as an additional delay to the completion of the current node. This

ensures that other robots wait for the human operator to fix assembly before marking the current node as complete. Newly completed nodes can enable the central server to enqueue new nodes if their outgoing edges were previously preventing unsafe actions. During execution, the TPG can be interpreted as a control law that maintains the safe scheduling of individual robot actions.

5 Third-Party Libraries

5.1 Visualization: lego-failure-web

Purpose: Streams the execution state into a browser-based dashboard so operators can inspect skills, predicted failures, and 3D reconstructions of each assembly step.

Code highlights:

- backend/failure_summary_node.py subscribes to perception and planner ROS topics, fuses them into a concise failure summary, and republishes the status for the web UI.
- backend/render_assembly_steps.py converts LEGO instructions into GLB/PNG assets via Blender so the frontend can display the target structure.
- frontend/src/App.tsx (React + Tailwind + `@google/model-viewer`) renders live telemetry, error messages, and the generated 3D models.
- setup.sh builds the assets, launches the ROS backend, and starts the development server for local testing.

5.2 Perception Skill: Detect Pick/Place

Purpose: Provides the perception stack for the DetectPick/DetectPlace skills and for gap monitoring around the assembled structure.

Code highlights:

- scripts/pick_place_classify_node.py loads a DINOv2 + SVM classifier that labels whether a gripper has a brick in hand and publishes the result to */robot/pick_place_classify*.
- Datasets, model checkpoints, and helper scripts live under *tools/lego-failure/models* and *tools/lego-failure/scripts*, providing the training and evaluation utilities used by our perception skills.

5.3 Perception Skill: DINOv2 NIST Detection

Purpose: A global perception skill for NIST box detection using an external stationary camera to estimate the rough 3D position of the box in the robot's workspace.

Code highlights :

- Dinov2Matcher Class: The core class in 'run_dino.py' that encapsulates the entire detection pipeline, handling model initialization, feature extraction, and real-time processing. It manages the DINOv2 ViT-B/14 model, processes reference images, and performs live feature matching against camera feeds.
- calculate_heatmap: Creates colored similarity heatmaps by computing cosine similarity between reference and target features, applying threshold filtering (default: 0.6), and using K-means clustering to identify detection clusters for precise component localization.
- pixel_to_3d(): Converts 2D pixel coordinates to 3D camera coordinates using depth information from RealSense cameras, applying pinhole camera model with pre-calibrated intrinsic parameters for accurate 3D reconstruction.
- lego_edge_matcher(): Implements edge detection using a separate DINOv2 matcher instance with different reference patches, providing complementary detection capabilities for comprehensive NIST box component identification.

- main(): Coordinates the full detection pipeline by processing live camera feeds, performing dual-stage detection (center and edge), applying clustering algorithms, and publishing robot control commands via ROS topics.

5.4 Perception Skill: Lego Offset Detection

Purpose: A fine-grained perception skill is developed for Lego stud detection using an in-hand camera mounted on a Gen-3 robotic finger.

Code highlights :

- pub_detect_offset: Start-point for the inference time detection module
- YOLOv8 model: A YOLOv8 model is trained on a custom lego dataset to perform real-time stud segmentation with high spatial accuracy.
- detect_offset: From the predicted masks, a circle-fitting algorithm estimates stud centers and radii. Post-processing methods then optimize these parameters such as outlier-filtering, smoothing and making the fitted circles lie on two perpendicular tangents, enforcing the orthogonal arrangement expected in Lego stud patterns. The relative pose between the end-effector tool and the Lego block is computed in 2D pixel space, considering only the x- and y-coordinates and the orientation derived from the refined stud positions.
- This integrated perception pipeline enables precise alignment for downstream manipulation, supporting contact-rich, sub-centimeter accuracy tasks in robotic assembly.

6 Video Skill Extraction

Purpose: A video-based skill extraction pipeline enabling zero-shot skill sequence extraction from a video of human building legos.

Code highlights

- Task Enum: The Task enum defines key robot actions in LEGO assembly, including PICK, PLACE_CENTER, PLACE_SHIFTED, and DEMO_OVER. Each has a readable display_name and short description, ensuring consistent, interpretable action types for model prompts and task generation.
- Brick Enum: The Brick enum lists all LEGO brick types used in the assembly, such as RED_2x4, BLUE_2x2, and YELLOW_1x3. Each entry includes a readable display_name and a description specifying color and size, ensuring clear identification in both visual and structured model outputs.
- class LegoCountResponse: The LegoCountResponse Pydantic model defines the schema for the model's scene-understanding output. It stores the list of detected bricks and their counts, enabling consistent parsing and downstream use in the assembly pipeline.
- createClass_lego_assembly(): This function dynamically creates a Pydantic class that defines the structure of a LEGO assembly sequence. It generates fields for each step (task, brick type, arm used, and timestamp) to guide Gemini's structured JSON responses.
- initial_scene_understanding(): This function analyzes the first and last frames of a LEGO assembly video using Gemini to detect and count visible bricks, especially the base ones. It outputs a parsed count summary and saves an initial setup file (env_setup_gemini.json).
- main: The main() function coordinates the full pipeline—initializing Gemini, analyzing the video, creating the task schema, prompting the model, and saving the final structured plan (assembly_task_gemini.json).

7 WebGUI

The WebGUI for using the SkillGraph and executing skills are shown in fig. 2.

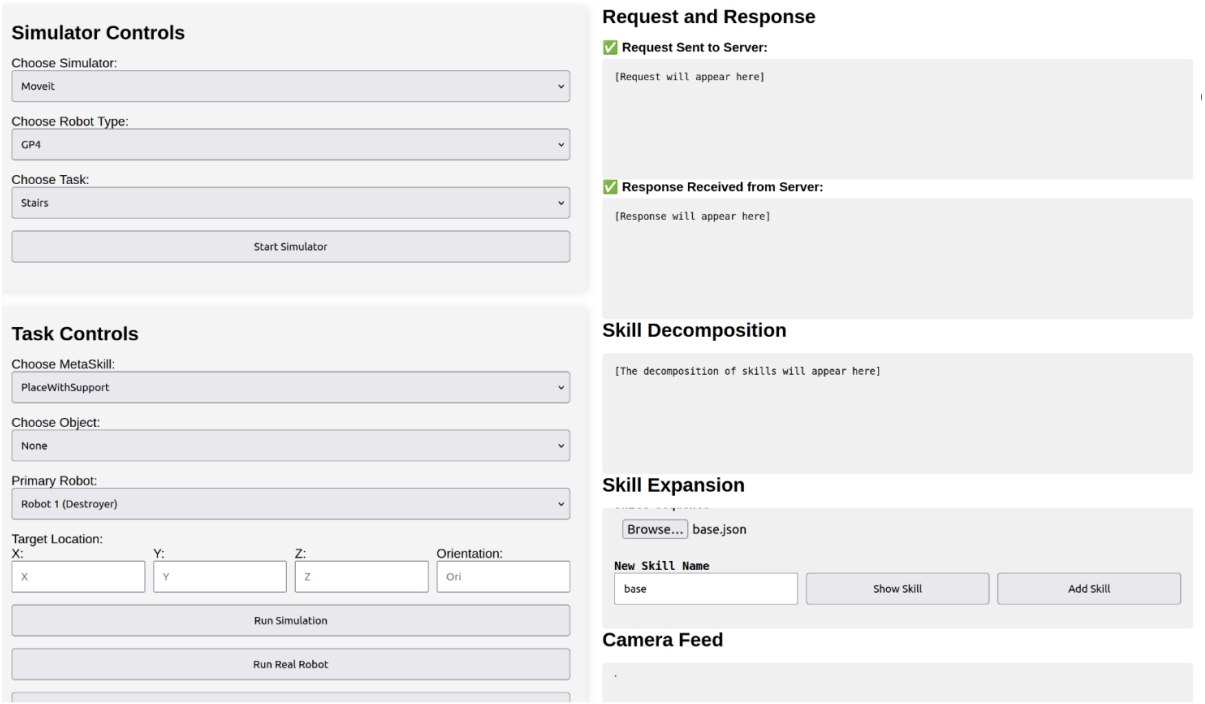

Figure 2: WebGUI

7.1 How to run WebGUI

The webpage API interacts with the simulators through communication with the server. To start the frontend of the webpage, run "run_frontend.sh", and visit localhost:8000/api.html for the web visualization. To start the backend of the webpage (that connects to the C++ skillgraph implementation), run "run_backend.sh".

7.2 How to interact with the simulator?

After you opened up the webpage and the server, choose your preferred simulator, robot type, and task under "Simulator Controls", and then click on [Start Simulator]. If the "Response Received from Server" shows the message is correctly delivered, you should see you simulator start to run(make sure you have your simulator correctly installed and the environment approriately activated!).

After the simulation has been started, choose skill, object, main robot, and target locations, and then click on [Run Target Task]. If your chosen set is feasible, you would see the robot moving as expected; otherwise, you would see an error pop out.

7.3 How to modify the commands?

Given that everyone's execution environment differs, we encourage you to check and modify the command "command = xxx TODO: Update the real execution commands" in server.py in order for successful interaction.

8 Implementation Details

In this section, we present the structure of our code and explain how each component from the previous sections is implemented within it.

8.1 Code Structure and Component Implementation

/AIDF
|— exe/ (includes executables for running the planner and visualization)

|— tools/ (include third-party libraries)
|— config/ (includes skillgraph configurations)
|— planner/ (includes planner definition and implementations)
|— skillgraph/
| |— api/ (class and data structure definitions)
| | |— Utils/
| | |— skillgraph.hpp (implement the structure of the skill graph)
| | |— algorithms.hpp (implement *SkillExecutor* and *Algorithm*)
| | |— backend.hpp (implement the backend definitions in *Environment*)
| | |— environment.hpp (implement *Environment*)
| | |— metrics.hpp (implement *Metric*)
| | |— objects.hpp (implement *Object*)
| | |— robots.hpp (implement *Robot*)
| | |— skills.hpp (implement *Skill*)
| | |— tasks.hpp (implement *TaskParam*)
| | |— data_structure.hpp (implement *Data Structures*)
| |— src/
| | |— corresponding cpp files...
| |— include/
| | |— lego/ (lego-specific implementations)
| | |— Instance Implementation.hpp files

9 Future Work: A Multi-Layer Framework for Continual Improvement

While the current SkillGraph implementation establishes a robust ontology for planning and execution, it remains a static system. The next phase of development will transition the SkillGraph into a dynamic, "closed-loop" framework. By leveraging the introspectable nature of the ontology—-specifically **SkillParam**, **SkillExecutor**, and **MetaSkill**-—we propose implementing four hierarchical feedback loops that allow the system to adapt at the parametric, policy, planning, and generative levels.

9.1 Parametric Adaptation: Closing the Physical Loop

Real-world deployments inevitably suffer from calibration drift and mechanical wear, leading to systematic biases that static plans cannot address. To resolve this, we will implement Parametric Adaptation (Loop 2) to automatically tune **SkillParam** data structures.

- **Mechanism**: We will introduce a "Parametric Corrector" module that subscribes to quantitative error data logged by the **Metric** class (e.g., offsets $\Delta x, \Delta y, \Delta\theta$ captured by the Lego Offset Detection skill).
- **Adaptation**: Instead of requiring manual recalibration, the system will maintain a running estimate of the systematic bias. Before a **SkillExecutor** runs, it will query this corrector to adjust the target parameters dynamically:

$$Pose_{corrected} = Pose_{target} - Bias_{estimated}$$

- **Result**: This ensures high-fidelity execution alignment, effectively "healing" calibration drift over time without high-level replanning.

9.2 Policy Adaptation: Contextual Algorithm Selection

The current **SkillExecutor** design allows for a vector of multiple **Algorithm** implementations (e.g., Planner_RRTConnect vs. Planner_PRMStar). However, the selection of which algorithm to use is currently static. Future work will implement Policy Adaptation (Loop 3) to solve this as a Contextual Multi-Armed Bandit (MAB) problem.

- **Mechanism**: We will implement an **AlgorithmSelectionPolicy** that selects a specific **Algorithm** from the available vector based on the current **State** (context).
- **Learning**:The policy π will be updated based on execution rewards (success rate, execution time) logged by the **Metric** system.

$$\pi(State) \rightarrow Algorithm_{optimal}$$

- **Result**: The system will learn to swap strategies autonomously—for example, utilizing compliant controllers for contact-rich tasks while switching to stiff position control for free-space motion.

9.3 Planner Adaptation: Risk-Aware Cost Learning

The current **Assembly Planner** utilizes a best-first search based on cumulative costs. Currently, these costs are static and optimistic, ignoring the "cost" of risk. We plan to implement Planner Adaptation (Loop 4) to inject real-world failure rates into the symbolic planning process.

- **Mechanism**: A **FailureCostUpdater** will aggregate historical failure logs associated with specific **AtomicSkill** and **State** pairs to build an empirical risk model $r(State, AtomicSkill)$.
- **Adaptation**: The planner's cost function will be modified to include a weighted risk term, penalizing actions that frequently fail in specific contexts:

$$c'(State, Skill) = C_{static}(State, Skill) + w \cdot r(State, Skill)$$

- **Result**: The planner will autonomously generate "risk-aware" plans, avoiding brittle state-skill transitions that are theoretically efficient but practically unreliable.

9.4 Generative Adaptation: Learning from Human Intervention

Finally, to address unanticipated failures ("unknown unknowns") that cannot be solved by existing algorithms, we will implement Generative Adaptation (Loop 5). We will leverage the **Temporal Plan Graph (TPG)** pause mechanism—currently used for manual recovery—as a data collection event for Learning from Intervention (LfI).

- **Mechanism**: When the TPG pauses for human recovery, the system will record the intervention (e.g., teleoperated joint states or action sequences).
- **Adaptation**: An offline process will distill this demonstration into a new **MetaSkill** (a vector of **AtomicSkills**).
- **Result**: This new skill (e.g., **MetaSkill_Recovery_01**) will be added to the library and linked to the failure state, allowing the planner to autonomously invoke this recovery strategy in future occurrences.