ChatGPT

Research Plan: Language-to-Mechanics IR Compiler with Auditor and MPC for Peg-in-Hole

1. Project Structure Overview

We propose a modular system that transforms natural language instructions into executable robot control for a peg-in-hole task. The system's architecture is organized into distinct modules, each with clear responsibilities:

- **LLM Interface & IR Compiler:** Receives high-level language commands and uses a large language model (via OpenAI or HuggingFace API) to generate a structured *Intermediate Representation (IR)* capturing the task's contact mechanics constraints and parameters. This stage translates phrases (e.g. "insert the peg gently and slowly") into quantitative constraints (e.g. force limits, speed targets) in the IR.
- **Mechanics IR Format & Parser:** Defines a formal schema for the IR (see Section 3) and parses the LLM's output into this schema, ensuring correct data types and units. For instance, if the LLM outputs "max force 10 Newtons" or "insert quickly (0.2 m/s)", the parser structures that into the IR fields with standardized units (N, m/s).
- **Mechanics Auditor:** A rule-based validator that checks the IR for physical consistency and feasibility before execution. It verifies unit consistency, constraint satisfiability, and contact mechanics realism (e.g. peg fits hole, force limits are sufficient) – akin to an affordance check ensuring the plan isn't dangerous or impossible [1]. If conflicts or impossible conditions are found, the auditor will either adjust parameters (using safe defaults) or request a revised IR from the LLM. This guarantees the robot only attempts feasible, safe actions.
- **MPC Controller (QP Solver):** A Model Predictive Controller that plans low-level motions and forces to achieve the task while respecting the IR constraints. At each control step, it solves a Quadratic Program (QP) to minimize trajectory error and forces, subject to constraints from the IR (e.g. not exceeding the specified force) and dynamic limits. This produces joint torque or velocity commands that drive the robot.
- **Simulation Interface:** Connects the controller to a physics simulator (MuJoCo for primary experiments) to execute the peg-in-hole task. It provides the robot's state to the controller (for MPC feedback) and applies the controller's commands to the simulated robot and environment. We implement the peg and hole as Mujoco bodies with defined geometry, friction, and sensors. An alternative differentiable simulator (Brax or Nimble) can be swapped in if end-to-end gradient analysis is needed.
- **Data Logging & Monitoring:** Continuously records relevant data – robot states, actions, contact forces, IR details, and any auditor interventions. This data is logged to files (e.g. CSV or JSON logs and possibly video frames) for offline analysis of performance. A monitoring process can raise alerts or halt the simulation if unsafe conditions occur (e.g. force exceeds limit despite commands), simulating a fail-safe intervention mechanism.

Each module is designed with clear interfaces: the IR (a structured data object) is the interface between the high-level LLM planner and the low-level controller. This modular decomposition improves system clarity and debuggability – for example, the IR and Auditor serve as an interpretable "checkpoint" that can be inspected or modified before the controller runs. Overall, the pipeline from language to action ensures high-level intent is translated into safe, verified low-level behavior.

2. Environment and Dependencies Configuration

We will implement the project in Python (⩾3.9) with a focus on replicability via a conda environment. Key dependencies include robotics simulation libraries, optimization tools, and ML frameworks:

- **Simulation Engine:** MuJoCo 2.x (e.g. v2.3.1) is our primary physics engine due to its proven stability and realism for contact dynamics [2] [3]. We will use the official MuJoCo Python bindings (`mujoco` pip package) or `mujoco-py` for integration. MuJoCo provides a fast, accurate simulation of rigid-body contacts and soft contact dynamics (needed for smooth peg-in-hole contact modeling). If gradient-based analyses are required, we consider **Brax** (Google's JAX-based differentiable engine) as an alternative for its end-to-end differentiability [2] [3]. For advanced contact gradient testing, **Nimble Physics** (a differentiable fork of DART) is another option, though its developers suggest using MuJoCo/Brax for general robotics due to contact gradient challenges [2].
- **Robot Modeling:** We will use an existing MuJoCo model of a 7-DoF robotic arm (e.g. Franka Emika Panda or UR5) with a cylindrical peg attached as the end-effector. The hole will be a static part in the environment. These models may come from standard libraries (e.g. robosuite or gymnasium assets) or custom MJCF files. The environment will be configured with gravity, contact properties, and sensor outputs (e.g. force/torque at end-effector).
- **Machine Learning Framework:** We plan to use **PyTorch** (for any learning or network inference, if needed for LLM API handling or future fine-tuning of prompts). If using JAX-based Brax, we will incorporate **JAX** and Flax, but the core design does not require writing new neural networks – the LLM is accessed via API.
- **LLM API Access:** For natural language processing, the system relies on external APIs. We will use the OpenAI Python API (for models like GPT-4) for high-quality instruction parsing. The environment will require an `openai` package and an API key configured. Alternatively, for an open-source approach, the Hugging Face Transformers library (with a model like Llama2 or GPT-Neo finetuned for instruction following) can be installed. In both cases, no local training is performed – the model is used in inference-only mode. Prompt engineering will be done to ensure the LLM outputs conform to the IR format (e.g. using few-shot examples in the prompt).
- **Optimization Tools:** The controller uses convex optimization. We will install **CVXPY** (v1.2+) as a high-level modeling library for MPC problems [4]. CVXPY allows us to define QP objectives and constraints in Python, which it will solve with an appropriate solver. For performance, we will use **OSQP** as the QP solver, given its efficiency for control-oriented QPs [5] [6]. The `osqp` Python package will be included, and CVXPY can interface with it as a solver backend [5]. OSQP's ability to handle quadratic cost and linear constraints in real time is crucial. We will also set up **numpy** and potentially **SciPy** for any additional math.
- **Additional Libraries:** For data handling and logging, we include pandas or simpler CSV writing utilities. If visualization is needed, matplotlib or mujoco's built-in viewer can be used to inspect trajectories (though not required for the core plan).

Conda Environment: We will provide an `environment.yml` specifying Python 3.10, `mujoco` (and ensure MuJoCo's binary is installed or use the `mjkey.txt` if needed for older versions), `pytorch`, `cvxpy`, `osqp`, `openai`, `transformers`, etc. Reproducing the environment will be as simple as running `conda env create -f environment.yml`. After installation, we will verify MuJoCo's functionality (rendering, stepping) and the OpenAI API connectivity.

MuJoCo Configuration: MuJoCo may require setting the `MUJOCO_HOME` or similar environment variable if not using the pip package. We will document this. We target MuJoCo's latest release to ensure up-to-date features and open-source license. If using Brax, ensure GPU support via appropriate CUDA and JAX versions in the env.

In summary, the project will rely on widely-used tools in robotics and ML, making it easier for others to replicate. The environment setup will be thoroughly documented, including any quirks (for example, MuJoCo requiring GLFW or specific drivers). By using stable versions of these libraries and containerizing environment config, we ensure the code is portable and reproducible.

3. IR Format Design for Contact Mechanics

The Intermediate Representation (IR) is a structured description of the task and constraints, serving as a bridge between the language instruction and the controller. We design a **schema** for this IR with fields capturing all relevant physical parameters, using consistent units and default values for unspecified details. The IR is essentially a mini domain-specific language for peg-in-hole instructions. Key aspects of the IR design include:

- **Fields and Meanings:** Each IR field represents a specific aspect of the task, with a defined unit and type. The core fields we propose are:
- *action_type* (string): The type of task or skill. For our case this is usually `"peg_in_hole_insertion"` (future extensions might have other actions).
- *peg_dimensions* (object): Geometric properties of the peg. Contains subfields like `length` (m), `radius` (m).
- *hole_dimensions* (object): Properties of the hole, e.g. `radius` (m) and `depth` (m). The **clearance** can be derived from peg vs hole radius (hole_radius - peg_radius).
- *material_properties* (object): Contact parameters such as `friction_coefficient` (dimensionless, default 0.3 for metal-on-metal) and optionally compliance or elasticity (if the peg or environment is not perfectly rigid).
- *speed* (float): The target insertion speed (m/s) or a qualitative speed mapping (e.g. “slow”→0.01 m/s). This primarily affects the approach velocity profile.
- *max_force* (float): Maximum allowable contact force in the insertion direction (Newton). This is critical for “gentle” insertions – the controller will treat this as an upper bound to avoid excessive force.
- *alignment_tolerance* (float): Allowable angular misalignment (degrees or radians) between peg and hole. If the instruction implies high precision (“ensure it’s perfectly aligned”), this tolerance would be small.
- *position_tolerance* (float): Allowable positional error or offset (in meters) for insertion (e.g. lateral play). Defaults to a small value (e.g. 0.5 mm) if not specified.
- *trajectory_strategy* (string/enum): Specifies if any special insertion strategy is used (e.g. `"spiral_search"` or `"straight_in"`). Default is direct linear insertion unless the instruction specifies otherwise.
- *environment* (object, optional): Any environmental assumptions, like `gravity` (m/s^2, default Earth 9.81) or if lubrication is present (which could modify friction). By default none.

- *units*: Although not a field per se, the IR uses SI units internally (meters, seconds, Newtons, kg). The LLM is prompted to output numeric values with units, which the parser will convert to SI. For example, if the LLM says “0.5 cm”, the parser converts to 0.005 m.

- **Default Values:** The IR compiler assigns defaults for fields not mentioned in the instruction, using sensible engineering assumptions. For instance, if friction is not mentioned, assume a moderate μ=0.3. If speed is not specified, default to a safe slow speed (e.g. 0.01 m/s) to avoid high impact. Max force default might be set to a value that the robot can safely exert (e.g. 20 N) if not given. These defaults come from domain knowledge or physical reasoning (e.g. insertion force typically small for a light peg). All defaults are documented so the auditor knows what was assumed.

- **Constraints Representation:** The IR can include explicit constraint statements. For example, `max_force: 10 N` or `speed <= 0.02 m/s`. We use inequality or equality signs in the IR text or assume that certain fields inherently define constraints. In our JSON-like structure, we might just have fields as above, and the controller interprets them as constraints (e.g. it knows *max_force* means an inequality constraint $F \leqslant 10$ N). If complex constraints arise (like "insert within 5 seconds" or "minimize wobble"), these can be captured as fields (`time_limit: 5 s`, etc.) and enforced accordingly. The IR is kept as simple as needed to cover our scenario, avoiding arbitrary code.

- **Type and Unit Annotations:** Each field's type (int, float, enum) is defined. For instance, *friction_coefficient* is a float $\in [0,\infty)$ (typically [0,1] physically), *max_force* is a non-negative float, *tolerance* fields are non-negative floats. Units are indicated in the field name or documentation (e.g. we may use suffix or a subfield like `"max_force_N": 10` to be explicit; alternatively, since we standardize to SI, we may omit units in keys but keep them in comments or documentation). The IR compiler ensures any numeric has an interpretable unit – if the LLM gives a number without unit, the compiler assumes the default unit for that field (e.g. "5" for force is 5 N).

- **Conflict Resolution Strategy:** In case the LLM output or user instruction contains contradictory information, the IR compiler or auditor must resolve it. For example, an instruction might implicitly say "fast but gentle" – implying high speed and low force, which are conflicting. Our strategy is to prioritize **safety-related constraints (force limits)** over performance (speed). Thus, in a conflict, the compiler might set the conservative values: e.g. *max_force* = low as requested, and *speed* = moderate (downgrade "fast" to a safe speed that still won't overshoot the force limit). The Mechanics Auditor will flag the inconsistency (speed vs force conflict) and can adjust the plan or issue a warning (perhaps triggering a prompt back to the LLM: "The instruction asks for high speed and low force, which is physically challenging; adjust one"). In automated mode, we implement simple rules: never violate a max_force due to a speed request – instead reduce speed. If geometric data conflicts (say one part of instruction says peg diameter 5mm and another says hole diameter 4mm, which is impossible), the auditor will choose a resolution (e.g. use the first mention or the larger hole size) and log a warning. We bias towards *feasible* interpretations – ensuring peg <= hole. In cases where a conflict can't be resolved algorithmically, the system will default to a safe setting (e.g. making the hole bigger to accommodate) and record that assumption.

- **Example IR Output:** Below is an example IR (in YAML/JSON-like format) derived from a user instruction *"Slowly insert the 10mm peg into the hole carefully, without using more than 5N of force."*

```
action_type: "peg_in_hole_insertion"
peg_dimensions: { radius: 0.005, length: 0.10 }
# in meters (5mm radius, 10cm long peg)
hole_dimensions: { radius: 0.0055, depth: 0.05 }     # assume 0.55cm radius
hole (0.5mm clearance)
material_properties: { friction_coefficient: 0.2 }   # assume smooth metal
(if not given)
speed: 0.002       # m/s, interpreted from "slowly"
max_force: 5       # N, from instruction
alignment_tolerance: 2.0      # degrees, default for "carefully" we pick
small misalignment
```

```
position_tolerance: 0.0005    # 0.5 mm, default small value
trajectory_strategy: "straight_in"   # default strategy
```

In this IR, fields not explicitly mentioned (like friction, tolerances) were filled with defaults, but all critical constraints from the instruction are present. The IR is human-readable and could be represented as JSON in code. This IR will be passed to the Mechanics Auditor next. Having a well-defined IR format with units and defaults ensures consistency between runs and allows easy extension to other tasks (by adding new fields or action_types as needed).

4. Mechanics Auditor Implementation Details

The **Mechanics Auditor** is a verification module that checks the IR for any issues and enforces physical realism before the control is executed. It acts as a safety net, much like a type-checker or unit-test for the plan. The auditor performs several layers of checks:

- **Dimensional Consistency & Unit Checks:** The auditor first ensures all numerical values in the IR have proper units and fall into reasonable ranges. It will convert units if needed and verify consistency (e.g. if an acceleration field existed, ensure it's in m/s^2, etc.). It checks that no required field is missing (if so, fill with default and flag it). For example, if the IR specified a force of "5" with no unit, it assumes Newtons but would log that assumption. If any value is outlandish (e.g. a negative friction coefficient or a tolerance greater than the peg size), that is flagged as an error. This step prevents simple errors like unit mix-ups or typos from propagating.

- **Geometry Feasibility (Contact mechanics consistency):** The auditor validates the geometric relationship between peg and hole. **Key rule:** the peg's diameter must be less than the hole's diameter (or at most equal, if a press-fit is intended). If `peg_radius > hole_radius`, the auditor finds an **unsatisfiable constraint** – insertion is physically impossible without deformation. In such a case, it will either (a) adjust the radius if it's a minor discrepancy and likely an error (e.g. make peg_radius equal to hole_radius for a tight fit, or reduce peg_radius by a small epsilon), or (b) abort and request the user/LLM to clarify. Likewise, if *clearance* is extremely small and *friction* is high, the auditor will warn that **jamming** may occur. Jamming and wedging are known problems in peg-in-hole when two-point contacts and friction prevent motion [7]. The auditor uses simple models: for instance, estimate that if clearance < 0.1 mm and friction coefficient > 0.6, the risk of jamming is high – it could then suggest reducing friction (if material can be changed) or ensuring a chamfer on the peg (if our IR could specify that), or simply flag this for the experiment log. It also checks alignment tolerances: if a large misalignment tolerance is given while clearance is very tight, that's inconsistent (tight fit requires small angle error). The auditor would then reduce the tolerance or warn about it. Essentially, all combinations of geometry and mechanical parameters are cross-checked for consistency.

- **Physical Satisfiability of Constraints:** The auditor simulates whether the given constraints can all be satisfied by some trajectory. For example, if the IR asks for an insertion speed of 0.1 m/s but a max force of 1 N, can those coexist? A quick impulse/momentum estimate might say a 0.1 m/s insertion of a certain-mass arm will likely exceed 1 N on impact. The auditor will flag that high speed + low force are conflicting. It might use a heuristic: compute an estimated impact force = (effective mass * velocity^2 / stopping distance). If that estimated force >> max_force, that's a red flag. In our example, it would likely be the case; the auditor could then automatically suggest lowering speed to match the gentle force requirement (or raising the force limit if speed is crucial). This is analogous to how the SayCan system filtered actions by feasibility [1] – here we filter the IR by physics feasibility. Another example: if *max_force* is set below what static friction

requires to overcome sticking, the peg would never move. The auditor can estimate the needed force to overcome static friction: $F_{min} = \mu \cdot N$ (where N is normal force, which in vertical insertion might equal weight of peg or some pre-load). If the IR's max_force is below this F_{min}, the action is infeasible (the peg would stick). The auditor catches this and could adjust max_force up to that threshold (or again, warn/fail).

- **Units and Scale Checks:** It verifies that values make sense scale-wise. E.g., if someone specified a mass or force, is it within the robot's capability? If the peg weight isn't in IR but deduced (say the peg is steel of given dimensions, the auditor can estimate its weight via volume * density). This weight leads to a minimum necessary force to lift or insert vertically. If max_force is below weight, obviously insertion can't succeed (gravity alone exceeds the limit), so auditor flags that. All such physical sanity checks ensure the plan is within the **robot's operating envelope** and the laws of physics.

- **Conflict Resolution & Warnings:** When the auditor finds an issue, it employs a hierarchy of fixes:

- **Auto-correction:** For minor issues, adjust values to safe defaults. E.g. if peg radius was slightly larger than hole, set peg_radius = hole_radius minus a tiny epsilon (making it an extremely tight clearance) and mark this correction.
- **Conservative bias:** If speed vs force conflict, prefer the safer side (lower speed or higher force) as mentioned. The auditor might modify the IR accordingly, adding a comment like "(adjusted by auditor)".

- **Logging and Requesting Revisions:** Significant conflicts that risk failure will be logged as warnings. These warnings are output for the user/developer, and optionally we can feed them back into an LLM prompt to refine the plan. For instance, programmatically: "The auditor found that the peg is larger than the hole. Please correct the dimensions." – send that to the LLM to get a corrected IR. In autonomous mode, one could iterate LLM→IR→Audit until no errors, though for this project we assume one pass with perhaps simple internal fixes. All adjustments are recorded so we can evaluate the necessity of the auditor by how often it intervenes (the *intervention rate* metric).

- **Contact Consistency Check:** Since this is a contact-rich task, the auditor ensures the IR accounts for contact realism. For example, if the IR implies no contact force (max_force = 0) but obviously insertion needs contact, that's unsatisfiable. Or if the IR sets friction to 0 but nothing in instructions suggested a frictionless scenario, it might be a mistake – the auditor might raise friction to a small value (or if it was intentionally frictionless, then okay). If the IR includes a *trajectory_strategy*, the auditor checks if it fits the scenario (e.g. if strategy is "spiral_search" but the hole is not tight or instruction didn't mention search, this might be irrelevant – auditor might ignore or remove it to simplify control).

The **logic structure** of the auditor is implemented as a series of checks (perhaps as functions) that run in sequence. For clarity, we can structure it as:

```
def audit_ir(ir):
    issues = []
    # 1. Unit and basic range checks
    for field, value in ir.items():
        if field in expected_units:
```

```
            value_SI = convert_to_SI(value, field_unit[field])
            ir[field] = value_SI
        if field in non_negative_fields and value < 0:
            issues.append(f"{field} has invalid negative value")
            ir[field] = abs(value)  # or default

    # 2. Geometry checks
    if ir['peg_dimensions']['radius'] > ir['hole_dimensions']['radius']:
        issues.append("Peg is larger than hole; adjusting to fit.")
        ir['hole_dimensions']['radius'] = ir['peg_dimensions']['radius'] +
0.0001  # add tiny clearance

    # 3. Friction vs Clearance vs Force checks (simplified)
    clearance = ir['hole_dimensions']['radius'] - ir['peg_dimensions']
['radius']
    mu = ir['material_properties']['friction_coefficient']
    if clearance < 1e-4 and mu > 0.5:  # very tight and high friction
        issues.append("High friction with very tight clearance may cause
jamming.")

    # 4. Force and motion feasibility
    v = ir['speed']; Fmax = ir['max_force']
    estimated_impact_F = estimate_impact_force(v, peg_mass_estimate)
    if estimated_impact_F > Fmax:
        issues.append("Planned speed likely causes force > max_force;
consider reducing speed.")
    # (No auto-change here, just warn or could auto-reduce v)

    # ... additional checks ...
    return ir, issues
```

In practice, the auditor will produce a list of warnings or corrections. If no major issues, it "approves" the IR and the pipeline continues. If serious issues arise (unsolvable conflicts), it can halt execution prior to controller, outputting the error. This ensures the robot never executes an unsafe or nonsensical command sequence.

Overall, the Mechanics Auditor ensures **mechanical consistency** of the plan. It fulfills a role similar to a "physics gatekeeper" – much like SayCan's value function filter prevents infeasible actions [1], our auditor prevents physically impossible or dangerous commands from reaching the robot. This added validation layer significantly increases the robustness and safety of the language-to-action pipeline, as confirmed by our evaluation metrics (we expect far fewer constraint violations when the auditor is active). The auditor's checks and any modifications are logged, contributing to an *intervention rate* metric: how often and why the auditor needed to step in.

5. MPC/QP Controller Design for Constrained Contact Control

At the core of the execution is a **Model Predictive Controller (MPC)** that converts the validated IR into real robot motions. We design the MPC as a finite-horizon optimal control problem solved at each time

step as a QP, which naturally handles linear constraints and quadratic costs in real time. The main components of the controller design are:

- **State and Dynamics Model:** We use a simplified dynamic model of the robot peg and environment suitable for MPC. To keep the QP solvable in real-time, we will use a *linear or linearized model*. For instance, if controlling in task-space (end-effector), we can model the peg's motion in the insertion axis with a second-order system (mass-spring-damper, where spring comes from contact stiffness). Alternatively, use a Jacobian linearization of the robot around the current configuration to relate joint torques to end-effector acceleration. Given MuJoCo can simulate full dynamics, the MPC model doesn't need to be perfect; it just provides a predictive trajectory for a short horizon. We assume small deviations, so linearization is valid over the horizon. At each step, the model is updated (re-linearized) at the new state if needed (receding horizon control). This approach yields a convex QP at each step.

- **Optimization Variables:** The QP's decision variables typically include the sequence of control inputs over the horizon (e.g. joint torque commands $\tau_{0\ldots N-1}$ or end-effector force commands) and potentially the states if using direct transcription. However, to keep it QP, we likely formulate it in a **delta control** fashion or utilize the structure where future states are an affine function of current state and control in a linear model. In a common MPC-QP setup, one might treat the next control command as the variable and use a one-step horizon (effectively making it a feedback QP controller). We might implement a short horizon (e.g. N=10 steps at 0.02s each = 0.2s horizon) to anticipate upcoming contact. The variables can also explicitly include *contact forces* at predicted contact times if we enforce contact constraints (this leads to a mixed complementarity formulation, but we can approximate contact as either always active after insertion begins, or ensure smooth engagement by design). We will experiment with including normal force as a variable in the QP (common in contact control formulations that treat contact force as decision variable along with motion, enforcing complementarity constraints – though complementarity is non-convex, we approximate as discussed below).

- **Cost Function:** A quadratic cost is defined to achieve the task while being gentle. For example: $J = \sum_{t=0}^{N-1} [w_p \|x_t - x_{\text{goal}}\|^2 + w_v \|v_t\|^2 + w_u \|\tau_t\|^2 + w_f (F_{n,t} - F_{\text{des}})^2]$. Here x_{goal} is the desired final pose (peg fully inserted), and we penalize position error (to drive insertion), velocity (to damp motions as needed), input effort, and the deviation of contact force from a desired value. If we want to explicitly *minimize contact force*, we could set $F_{\text{des}} = 0$ and give that term a high weight until nearing completion (or minimize overshoot beyond a threshold). However, since we also have constraints on force, we might not need to cost it heavily – constraints will cap it. We will at least include a small weight on normal force to encourage the controller to stay below the cap (soft constraint). The cost weights w_p, w_v, w_u, w_f are tuning parameters. We likely prioritize avoiding force overshoot (for safety) and achieving insertion (so position error) over raw minimal control effort.

- **Constraints in the QP:** This is where we enforce the IR specifications:

- **Force Constraints:** The normal contact force F_n on the peg must stay $\leqslant$ *max_force* from IR. We cannot directly "command" force in position-controlled robots, but through torque and motion we influence it. We handle this by imposing either a constraint on the optimizer's predicted normal force variable or by limiting penetration. One approach is to use a *soft contact model* in the prediction: e.g. assume a linear spring of stiffness K at the contact. Then penetration d relates to normal force $F_n = Kd$. We impose $F_n \leq F_{max}$ as a linear constraint $Kd \leq F_{max}$, which is $d \leq F_{max}/K$. This effectively means the controller won't push deeper than a certain

small penetration that would create a force equal to the limit. This is a **soft-contact approximation** similar to MuJoCo's model that allows slight penetration with a penalty [8]. By dropping hard non-penetration constraints and using a soft spring, we get a convex formulation where a small violation is allowed at a known cost. This aligns with MuJoCo's philosophy that slight penetrations increase realism for soft contacts and avoid the non-differentiability of hard contacts [8]. The softness can be tuned such that normal force constraints become linear and tractable. If the peg hasn't made contact yet, $F_n = 0$ naturally. If contact is made, the model introduces that spring force and the constraint above kicks in.

- **Friction Constraints:** In a fully accurate model, friction yield is a nonlinear (quadratic) constraint ($\|F_t\| \leq \mu F_n$ defining a cone). To keep it in QP, we **linearize the friction cone** into a pyramidal approximation [9] [10]. For example, we can constrain tangential forces by two linear inequalities: $-\mu F_n \leq F_t^x \leq \mu F_n$ and $-\mu F_n \leq F_t^y \leq \mu F_n$ (assuming 2D tangential plane). This forms a friction pyramid – an acceptable convex approximation of Coulomb friction [11] [12]. Linearizing friction cones is standard in MPC formulations to maintain convexity [10]. We will incorporate such constraints if we decide to have the controller explicitly reason about friction forces (it might implicitly handle it via the simulator feedback anyway, but including it can help limit commanded lateral motions that would cause slip).
- **Kinematic Constraints:** We ensure the peg remains approximately aligned: we can constrain angular deviation to within *alignment_tolerance* (e.g. $|\theta_{misalignment}| \leq \theta_{\max}$). If linearized (small angles), this can be a linear constraint on orientation coordinates or simply enforced by commanding orientation to track the hole orientation strongly (we might not need an explicit inequality if cost is high for orientation error).

- **Joint/Actuator Limits:** The QP will also include bounds on joint torques (not to exceed robot's capabilities) and joint positions/velocities if needed (to avoid singularities or joint limits). These are linear constraints or bounds easily handled by OSQP. They ensure the generated trajectory is within hardware limits.

- **Approximation Strategies:** To handle the inherently hybrid nature of contact (before contact vs during contact), we adopt strategies:

- *Soft constraints:* as noted, allow slight penetration with a spring and penalize it rather than a hard non-penetration constraint. Similarly, allow a tiny violation of force limit (slack variable) but heavily penalize it in cost – this ensures feasibility of the QP even in edge cases [13] [14] (the controller won't completely fail if, say, a tiny overshoot is needed; it will just incur a high cost, meaning in practice it will avoid it).
- *Local linearization:* When contact is active, we linearize dynamics around that mode (assuming sticking contact for now). If contact state changes (e.g. peg breaks free or slips), the MPC will re-linearize next step. This is a common approach to deal with mode switching: treat mode as constant over the short horizon, and replan frequently. The high control frequency (e.g. 50 Hz) means we can adapt quickly if mode changes.
- *Horizon management:* Early in the insertion (no contact), the friction and force constraints are irrelevant (no contact force). We can simplify the QP until contact is detected to reduce computation. For instance, we might run a simpler free-space trajectory controller (or MPC without force constraints) until a contact sensor triggers, then switch to the full constrained MPC. However, MuJoCo provides continuous contact forces, so we can continuously run the full QP but before contact the force constraints are inactive (since $F_n = 0$ and $d = 0$ automatically satisfies $Kd \leq F_{max}$). This avoids explicit switching logic.

- *Goal relaxation near completion:* To avoid oscillations or excessive force as the peg fully seats, we may relax constraints slightly when nearly done or incorporate a damping strategy (like once

insertion depth is reached, allow a tiny force to seat it firmly, etc.). This is more of a fine-tuning strategy to improve success and not strictly part of the main design.

- **Execution Interface:** The MPC QP is solved at each control cycle (e.g. every 20 ms). We use OSQP's fast solver which can update and solve QPs with warm-start in a few milliseconds for problem sizes on the order of tens of variables and constraints (which our formulation falls into). The output of the QP is typically the first control action (e.g. a joint torque vector or a desired end-effector velocity/force). We then apply this command to the robot in simulation via MuJoCo's API (for instance, if torque control, set the actuator torques; if velocity control, integrate the commanded velocity over the small timestep). Then we obtain the new state, update the linear model and constraints if needed, and solve the next QP. This feedback loop continues until the peg is inserted or a timeout occurs. The execution interface also collects actual sensor readings: notably the measured contact force – which we use to enforce constraints in real-time (if for some reason the solver didn't predict a spike, a safety monitor can cut input if force > max_force as an emergency stop).

- **Handling Soft Contact in Controller vs Simulator:** We note that MuJoCo itself uses a soft contact model [15] [8]. We will align our controller's assumptions with MuJoCo's settings (e.g. using similar stiffness/damping so that our predicted forces match what MuJoCo computes). MuJoCo's contact solver ensures a smooth (though stiff) contact; our controller will thus see forces rise as penetration increases. By tuning our K (contact stiffness) to a value, we effectively tell the controller how force grows with penetration. We might use MuJoCo's default contact parameters (which yield a convex, smooth contact model [13]) in our prediction as well, for consistency.

- **Example Constraint Setup:** Suppose the peg just touched the hole entrance. The MPC QP now includes: minimize insertion depth error + penalize forces; subject to $d_t \leq d_{max}$ (from $Kd \leq F_{max}$), and $-\mu F_n \leq F_{t,x} \leq \mu F_n$ etc. If the IR said max_force 10 N, and K=10000 N/m, then $d_{max} = F_{max}/K = 0.001$ m = 1 mm. So the QP will not allow the peg to push more than 1 mm into the surface in that 20ms step, limiting force to ~10 N. The next time step, maybe it goes another 1 mm, etc., maintaining force ~10 N steady if needed to insert. This way we achieve a nearly constant force insertion up to the limit, which is a gentle insertion strategy. If friction requires say 5 N to overcome and we allowed 10 N, we have margin; if friction was too high (like requiring 15 N to move but we cap at 10 N), the QP will push to the max d_{max} but progress might stall – that would indicate the plan constraints themselves prevent success (which the auditor should have caught, but if it happens, the controller effectively respects the constraint at the cost of stalling – an outcome we'll record).

By integrating these constraints, the controller *actively enforces* the high-level directives from the IR. For instance, "do not exceed 5 N" is guaranteed by the QP solution (barring model mismatch or solver tolerance). We also ensure the solution remains convex quadratic so that it can be solved quickly and reliably at runtime. Linearizing the friction cone and contact model was essential for that [10]. Any minor infeasibilities (like if the solver needs to break a constraint by 0.1% to find a solution) can be allowed via soft penalties, ensuring the QP always finds a solution and thus the MPC doesn't break down.

In summary, the MPC/QP controller translates the IR's physical limits and goal into real motions by solving constrained optimization problems in real-time. It balances competing objectives (speed vs. force vs. precision) optimally within the given constraints. This approach is more flexible and intelligent than a fixed gain controller – for example, a classical impedance controller might require extensive tuning to both insert quickly and not overshoot force; our MPC can naturally handle the multi-objective

trade-off by adjusting trajectory on the fly within constraints. We expect this controller to achieve high success rates even under varying friction or slight misalignments, as it can adapt each step (e.g. if it senses higher resistance, it will automatically slow down to stay under force limits due to the cost/constraints). The use of QP also gives us a framework to incorporate additional constraints (like joint limits, etc.) with minimal effort, increasing safety.

6. Peg-in-Hole Simulation Experiment Setup

We will conduct experiments in a simulated environment to evaluate the system. The peg-in-hole task is set up in MuJoCo with careful attention to realism and variability:

- **Robot and Objects:** We use a 7-DoF robotic arm model (such as the Franka Panda, which is common in assembly research) mounted on a table. Attached to the robot's wrist is a cylindrical peg. The peg's dimensions can be, for example, 10 cm length and 1 cm diameter (radius 0.005 m), unless specified otherwise in a particular IR. The hole is part of a fixed workpiece (e.g. a block on the table) with a matching round hole. The hole's diameter is slightly larger than the peg's – we will configure a nominal clearance of about **0.2–0.5 mm** (e.g. hole radius 0.0051–0.00525 m when peg radius is 0.0050 m). This clearance range represents a typical fine fit where insertion is possible but requires proper alignment. We can easily adjust these dimensions to test different scenarios (e.g. a tighter fit vs a looser fit). The peg and hole are given collision properties in MuJoCo so that contact forces are computed.

- **Contact Properties:** We set the friction coefficient in MuJoCo for peg-hole contact surfaces. By default, we might use $\mu \approx 0.3$ (like steel on steel). To test robustness, we will randomize friction in some trials (see below). The contact is modeled with MuJoCo's default soft contact parameters (solimp/solref settings) which provide a stiffness and damping – ensuring finite contact forces. We align our controller's assumptions with these parameters. No glue or welding effects are used – the peg can freely slide or get stuck only due to friction and geometry.

- **Initial Conditions:** At the start of each trial, the peg is positioned above the hole with an *initial pose offset*. We intentionally introduce small random misalignments to simulate real-world uncertainty: e.g. up to a few millimeters lateral offset and a few degrees tilt relative to perfect alignment. This requires the controller to correct alignment during insertion. We ensure the peg starts just above the hole entry (no initial penetration). The robot is holding the peg steadily at this start pose.

- **Environment Variations:** We incorporate **domain randomization** in simulation parameters to ensure our system is not overfit to a single scenario [16] [17]. Specifically:

- *Friction randomization:* For each trial, we randomly sample the friction coefficient between, say, 0.1 and 0.5 (covering very smooth to somewhat sticky). This tests if the controller can adapt to different resistance levels using the same IR constraints (the IR might assume nominal 0.3, but auditor/controller should handle if actual is a bit different).
- *Clearance variation:* We also vary the hole diameter slightly per trial (e.g. ±0.1 mm from nominal). This can cause some trials to be tighter or looser fits. The IR will generally assume the nominal clearance, but the controller will experience slight differences – a good test for robustness.
- *Peg material compliance:* We could vary contact stiffness (MuJoCo solimp) to simulate different material stiffness (rubbery vs metal contact). However, this is less visible to our high-level system as long as friction is accounted; we might keep stiffness constant for simplicity.

- *Sensor noise:* We can add a small noise to force/torque sensor readings to simulate imperfect measurements (MuJoCo allows adding noise to sensors). Our controller is fairly robust to small noise due to optimal feedback, but we include it for realism.

- These randomizations follow the approach of prior work that used domain randomization to cover a range of insertion scenarios [16]. We will run multiple trials with different random seeds to evaluate average performance.

- **Control Frequency:** The controller (MPC) will operate at a fixed frequency. We anticipate using **50 Hz** (i.e. 20 ms timestep) for the QP update, which balances reactiveness and computational load. MuJoCo's internal physics simulation can run at a higher rate (we might set the simulation timestep to 0.5–1 ms for stability of contact simulation, which is typical; MuJoCo can sub-step between controller updates). The robot's actuators will be considered as either direct torque control or low-level position controllers updated by MPC outputs. Using 50 Hz control is reasonable for modern arms and allows time for OSQP to solve the QP (which typically takes <5ms for small QPs). We'll monitor solver timing to ensure real-time performance is feasible; if needed, we can drop to 20 Hz, but likely 50 Hz is fine given OSQP's efficiency.

- **Episode Execution:** Each simulation trial (episode) will proceed until either the peg is successfully inserted or a timeout occurs. We define *success* as the peg's tip reaching the bottom of the hole (or the peg fully seated with a certain depth) without exceeding force limits. We define a reasonable timeout (e.g. 10 seconds simulated time) which is far more than needed if things go well (in real life insertion might take <5 seconds). A timeout might indicate a failure to insert (perhaps due to misalignment or jam). The simulation will be reset after each episode.

- **Sensors and Data Collection:** We instrument the simulation with sensors: the robot's joint encoders (to get positions for state feedback), a force/torque sensor at the peg (MuJoCo can simulate an F/T sensor on the wrist to measure contact forces), and possibly a "proximity" sensor to know when contact starts (or we can infer contact when force becomes non-zero). These help the controller and also allow logging of the actual force vs time, position vs time, etc. We will log key events: contact start, insertion complete, any time the force goes above the IR limit (this would be a violation event we want to measure). The data logging module will capture time series of pose, force, and the commands for analysis.

- **Example scenario configuration:** For clarity, one test scenario could be: Peg radius 5 mm, Hole radius 5.1 mm (clearance 0.1 mm), friction 0.3, initial lateral offset 0.2 mm, initial angle error 1°, max_force in IR 10 N, speed target 0.01 m/s. We expect the peg to make contact, experience friction but within the 10 N limit, and be guided in. In another scenario, friction might be 0.5 and clearance 0.0 (a tight fit). Our system should still succeed by perhaps oscillating or using near-max force, or it might fail if it can't overcome static friction at the limit – those outcomes are insightful to measure.

- **Simulation Platform:** We primarily use MuJoCo standalone. However, for easier integration, we might use the OpenAI Gym/Gymnasium interface or Robosuite's `InsertPeg` environment if available [18], customizing it to inject our controller. Alternatively, writing a custom simulation loop with MuJoCo's Python API for full control is straightforward and likely what we will do (to easily implement our own controller and logging).

By designing the simulation experiment with slight randomness and realistic physics, we ensure the evaluation will reflect the system's true strengths and limitations. The peg-in-hole task is known to be sensitive to alignment and force control, providing a stringent test of our language-conditioned

approach. Our setup will allow us to systematically vary conditions (material, clearance) and see how the system copes, which will be reported in the results.

7. Evaluation Metrics and Data Recording

We will evaluate the system on multiple metrics that capture both success in task execution and adherence to specified constraints. The following quantitative metrics are defined:

- **Success Rate:** The primary metric is the percentage of trials in which the peg is successfully inserted into the hole within the allowed time (and without human intervention). A trial is considered a success if the peg reaches the target depth with orientation alignment within tolerance, *and* it remains there (no bounce-out). Formally, success = 1 if final insertion depth $\geqslant$ (hole depth − small epsilon) and final misalignment $\leqslant$ tolerance. We will report success rate over, say, 50 randomized trials for each tested method. A higher success rate indicates the system's overall effectiveness at completing the task under varied conditions.

- **Peak Contact Force:** We measure the maximum contact force experienced by the peg in each trial (from the force/torque sensor). This is important for evaluating safety and gentleness of insertion. The IR specifies a max_force; ideally the peak force should remain at or below that. We will compute the average and worst-case peak forces across trials. Lower peak forces are better (especially if below any damage threshold for hardware or parts). We expect our MPC controller with force constraints to maintain low peaks, whereas a baseline without such control might exhibit higher spikes. This metric can be directly compared to the IR's commanded limit – any exceedance is noted as a violation (see next metric). For example, if IR max_force=10N and in one trial a transient 12N occurred, that trial had a 20% overshoot; we will analyze such cases.

- **Constraint Violation Rate:** This metric tracks how often the system violates the specified constraints (as given by IR). We define it as the fraction of trials (or time-steps) in which any constraint was broken. Constraints include force limit, alignment tolerance, etc. There are a couple of ways to calculate this:

- **Per-trial violation:** How many trials had *any* violation event? (e.g. in 2 of 50 trials, force exceeded the limit at some point -> 4% trials had a violation). We can further detail the type of violation (force, etc.).
- **Rate over time:** What percentage of control steps saw a violation? But ideally, a well-behaved controller should violate rarely or only momentarily. Likely the per-trial count is more interpretable (and we expect either zero or one distinct violation events per trial if any, typically).

We will likely present the percentage of trials with at least one violation of each type: - *Force violations:* any time measured force > max_force (with maybe a small tolerance if it's noise). - *Tolerance violations:* if at insertion completion the alignment or position errors exceed the allowed tolerances in IR. - *Other violations:* e.g. exceeding time limit (means failure essentially).

A low violation rate indicates the system respected the constraints it was given (which is a sign of the effectiveness of the IR+Auditor+MPC combination). We anticipate that with the Mechanics Auditor enabled and MPC controlling forces, the violation rate will be **near zero** for force and alignment

constraints (since those are actively enforced). This will be contrasted with baselines like “no auditor” or “no MPC” which likely have higher violation occurrences.

- **Intervention Rate:** This measures how often the Mechanics Auditor (or any supervisory element) had to intervene or modify the plan. Concretely, we can measure per trial:
- Did the auditor make any adjustments to the IR (yes/no)?
- Did the auditor abort or request re-plan (yes/no)?
- Optionally, if we had an outer loop to reprompt the LLM on failure, each reprompt count could be an intervention.

We will define intervention rate as the fraction of trials where the auditor detected and fixed a problem in the IR. A lower intervention rate in final system runs might indicate the LLM is producing better IRs or our default assumptions sufficed. However, some interventions (like minor unit conversions) will happen every time – we might not count those routine ones. We specifically focus on major interventions (like altering a conflicting parameter or flagging an issue). For instance, if out of 50 trials the auditor adjusted the plan in 10 of them (maybe due to very high friction draws), that’s a 20% intervention rate. This metric is more qualitative as it depends on the LLM output quality; but it helps understand to what extent the auditor is needed or how often it catches issues.

Additionally, if an *online* intervention occurs (for safety, say halting the robot if something unexpected happens), that would also count. In our simulation, one possible intervention is if force somehow goes above limit, the system could stop the trial. We will note if such emergency stops occurred (we intend them not to, but if they do, that is a violation and an intervention by a hypothetical safety monitor).

- **Efficiency Metrics (Secondary):** We will also record things like *insertion time* (how long it took to complete the task in each trial) and *trajectory smoothness*. While not explicitly requested, these give insight: a faster insertion that still meets constraints is better from a productivity standpoint. We can average the times for successful insertions. Also, the number of oscillations or adjustments might be qualitatively noted (did the robot insert in one go or did it have to back out and try again? Our controller might not explicitly do retry motions unless we design that, but it might slow down or pause if force is high, effectively waiting until alignment improves).

- **Ablation Comparison:** For each metric, we will compare the full system to baseline ablations (Section 8). For example, success rate of full system vs success rate without MPC, etc. We expect the full system to have the highest success and lowest violations. These comparisons will validate each component’s contribution.

Data Recording: Throughout each trial, we log data at each time step: - Robot state (joint angles, end-effector pose). - Command outputs (e.g. the QP solution such as desired force or torque). - Contact forces (normal and tangential). - IR and auditor info (the IR used, any adjustments). - Flags for any constraint active or violated at that step.

After trials, we aggregate these logs to compute metrics. For success rate and violation checks, we can parse the logs with a simple script to detect events (e.g. check the force time-series against limit). Peak force is simply max of the force time-series for each trial. Tolerance final values are taken from final step states.

We will present these results likely in tabular or graphical form in the paper. For example, a table might list success %, average insertion time, average peak force, and violation % for each method. Additionally, we may plot force-vs-time for representative trials to illustrate how our controller keeps

forces below the threshold (especially highlighting a case with high friction where it slowed down to maintain force < limit, etc.).

We ensure that all logged data is time-stamped and labeled by trial so that we can trace back any anomalies to what happened. The logging is also crucial for debugging during development (e.g. if something goes wrong, the logs and perhaps recorded video from MuJoCo can help pinpoint whether it was a planning issue or controller issue).

In summary, our evaluation metrics cover **task success** and **constraint satisfaction**, aligning with the system's dual goals of performing the task effectively and respecting physical limits. By systematically measuring these, we can validate that incorporating the IR and auditor indeed yields safer and more reliable performance (e.g. significantly fewer force violations [19] [20] than a naive approach that might slam the peg causing large forces). All metric computations and logs will be included with the code for transparency and reproducibility.

8. Ablation and Baseline Experiments

To demonstrate the necessity and benefit of each component of our system, we will conduct ablation studies and baseline comparisons. We specifically design three primary ablation conditions in contrast to the **full system** (which includes the LLM-generated IR, the Mechanics Auditor, and the MPC controller):

1. **No IR (Direct Control Baseline):** In this scenario, we bypass the language-to-IR process. Instead of using an IR with explicit constraints, the robot is commanded with a fixed default strategy for peg insertion. Essentially, the system ignores any nuanced instruction – it just performs a naive insertion. For implementation, we can use a simple controller (e.g. a position controller or open-loop trajectory) that moves the peg straight down into the hole at a standard speed and maybe stops when a certain force is felt. This is akin to how a robot would operate without language guidance or with a very coarse instruction. The purpose is to see how well an uninformed approach does. We expect that without an IR specifying, say, force limits, this baseline might drive the peg in quickly and possibly cause high forces or jamming if misalignment occurs (since it won't adjust parameters per scenario). It might have a lower success rate, especially in high-friction or tight-clearance instances, and higher peak forces. This baseline helps highlight if the high-level language (IR) input provides value (e.g., if the instruction "gently" indeed led to safer outcomes). It can also be considered a **human default policy** baseline – what if we didn't have the LLM and just programmed a generic insertion? We will measure metrics for this baseline (likely it will have higher constraint violations and maybe slightly faster insertions at the cost of success).

2. **No Auditor:** Here, the IR is used but we disable the Mechanics Auditor checks/adjustments. That means whatever the LLM outputs (or the default IR) goes straight to the controller, even if it's inconsistent or suboptimal. This tests the importance of validating the plan. In practice, if the LLM output had a mistake (e.g. it set an unrealistically low max_force or gave values in wrong units), the controller will try to follow those literally. We foresee several outcomes: (a) If the LLM IR was reasonable, we might not see much difference in performance from the full system (ideally the auditor is a safety net). (b) If there was an IR issue, the "no auditor" run could suffer failures. For example, suppose the LLM said max_force 2 N for a task that actually needs ~5 N (maybe "very gentle" insertion). With the auditor off, the controller will obey 2 N limit and likely stall (failure to insert) or take extremely long, resulting in more timeouts or low success. Or if a unit was misinterpreted (say LLM said "10" meaning 10 N but we took it as 10 (dimensionless) so

it became 10 m/s speed accidentally), that could cause huge error – the auditor would have caught unit issues. So "no auditor" baseline could highlight any catastrophic outcomes that our auditor prevented. We will quantify differences: success rate drop or violation rate increase when auditor is removed. This ablation underscores the auditor's role as a safeguard.

3. **No MPC (Simpler Controller):** In this ablation, we use the IR (and still run the auditor on it), but we do **not use the advanced MPC controller** to enforce constraints. Instead, we substitute a simpler control method for execution. One such method is a traditional *impedance or force-control scheme* without optimal predictive control. For example, we could implement a fixed stiffness Cartesian impedance controller: command the end-effector towards the hole with a certain stiffness and damping, and perhaps a simple force limit logic (like "if force exceeds X, pause or back off"). Many industrial solutions do this – essentially a hand-tuned controller. We will tune it reasonably (to give it a fair chance). The key difference is that this controller won't solve an optimization each step; it will react myopically (and often cannot perfectly respect constraints – e.g., a PD controller might overshoot a force limit before backing off). We expect that without MPC's predictive constraint handling, this baseline might exhibit higher peak forces and possibly more oscillation. It might still succeed in many cases (impedance control is known to work for peg-in-hole with compliance devices [19] [20]), but the quality of insertion may be worse (e.g., slower settling or minor violations). We will compare metrics: does our MPC achieve lower force spikes and faster insertion than a well-tuned impedance controller? This is a meaningful baseline since it represents the "classic" approach without advanced optimization.

Additionally, we can consider combinations or partial ablations: - **No IR + MPC:** (i.e., use MPC but with generic preset constraints rather than LLM-derived ones). This would test if having the language-conditioned parameters helps. For instance, we could run the MPC with a fixed max_force of, say, 10 N in all cases, versus our system which might set it lower if the instruction said "gentle". If the instruction is actually important (e.g. one scenario instructs gentle, another instructs fast), the fixed baseline wouldn't adapt. This baseline might perform decently on average, but it wouldn't be following the user's intent on a case-by-case basis. We likely will implicitly cover this by scenarios in which the instruction varies. The full system should adapt to each, whereas a baseline (no IR) would treat them all the same.

- **No IR + No Auditor + No MPC (fully manual):** This essentially collapses to a simple open-loop approach, similar to baseline 1 but even without any checking. However, baseline 1 was essentially that already. So we don't need a separate case.

- **Oracle IR vs LLM IR:** Another insightful experiment is to provide a *perfect IR* manually (as if an expert wrote it) to the system and see performance, then compare to LLM-generated IR performance. This isolates if any remaining failures are due to suboptimal IR from the LLM. If performance is similar, the LLM IR is as good as an expert's; if not, there's room to improve the prompt or LLM. We might do this qualitatively: e.g. if LLM gave a weird parameter, try a corrected IR.

For each ablation, we will run the same set of trials (with same random seeds/conditions) as the full system for a fair comparison. The results will be tabulated or plotted side by side. We anticipate findings such as: - The full system (IR + Auditor + MPC) has the highest success (nearly 100% in simulation if tuned well) and minimal constraint violations. - The no IR baseline likely has lower success especially in difficult scenarios (perhaps it succeeds in easy low-friction cases but fails if friction high or misalignment bigger because it might not adjust strategy). Also possibly it might inject larger forces (since it doesn't know to be gentle). - The no Auditor variant might show a similar success rate *if* the LLM output was usually sane, but any odd outlier could lead to a failure that the full system would have caught. Even if success doesn't drop much, the key difference might be that it violates some constraints

or performs in a less safe manner occasionally. We will highlight any such events. - The no MPC (with a fixed compliance controller) likely has success not far from the full system in nominal cases (since peg-in-hole can be done with a well-tuned compliance control [19] [20]), but we expect higher forces or a longer time to insert on average. For instance, the robot might wiggle or press harder than needed before finding the hole, whereas MPC might align better before insertion. Also, under tight tolerances, MPC can optimize approach angle, whereas a fixed controller might get stuck. Any such differences will be noted (e.g. maybe no MPC fails in the tightest clearance scenario whereas MPC succeeds, demonstrating its advantage in precision control).

- **Ablation on LLM quality:** Not a formal separate baseline, but we may also discuss how the system behaves with different LLMs (GPT-4 vs GPT-3.5 or an open model). If the simpler model produced less accurate IR, that effectively becomes like a "degraded IR" scenario. The auditor might have to intervene more, or the performance might suffer slightly. This highlights the importance of using a strong LLM for reliable parsing. We might include this in analysis if relevant (e.g. "with GPT-3.5, we observed auditor interventions in 30% of cases versus 10% with GPT-4, showing how advanced language understanding reduces errors").

- **Human teleoperator or heuristic baseline:** If relevant, one could compare to a human-designed strategy or teleoperation performance, but since this is simulation, we likely stick to automated baselines.

Our ablation study essentially demonstrates the contribution of each component: - LLM-based IR (for adapting to high-level instructions), - Mechanics Auditor (for ensuring feasibility and safety), - MPC controller (for enforcing constraints optimally).

By removing each, we expect performance to drop or some aspect to worsen, thus justifying our design. These results will be crucial in the paper's evaluation section to answer questions like "do we really need an auditor?" or "what do we gain from the MPC vs a simpler controller?".

We will present these findings likely in a dedicated subsection, perhaps with a bar chart for success rates and violation rates for each ablation. For example, a chart might show success: Full ~95-100%, No IR ~70%, No Auditor ~90% (with a note that failures were due to bad IR in a few cases), No MPC ~85%. And a separate chart for peak force where Full has lowest. Such visualizations will make the benefits clear.

In conclusion, the ablation experiments form a **controlled comparison** that will validate each part of our system. They transform a vague claim ("MPC with constraints is better") into concrete evidence (e.g., "peak force was 50% lower with MPC vs no MPC, preventing jamming in 5 cases that the PID controller failed [19] [20]"). These experiments strengthen the paper by showing the system's robustness is not magic but comes from the synergy of the IR, auditing, and MPC pieces.

9. Project Execution Timeline

Developing this integrated system involves multiple components (LLM integration, simulation setup, controller design, etc.), so a phased timeline will ensure steady progress. Below we outline a tentative schedule with milestones, assuming a project start in Month 1 (we can adjust actual dates as needed). We also identify potential risks at each stage and contingency plans to mitigate them:

- **Month 1: Literature & Environment Setup** – *Milestone:* Complete background research and software installation. We will thoroughly review related work (e.g. prior LLM robotics approaches,

peg-in-hole research) to inform our design. Concurrently, set up the development environment: install MuJoCo and verify we can simulate a basic scene, install LLM API access and test a simple prompt. **Risk:** MuJoCo or the robot model might have issues (e.g. licensing or compilation problems). *Mitigation:* Use the open-source MuJoCo 2.3 which requires no license, and test on multiple machines/OS. Another risk is slow API access to LLM. Mitigation: cache LLM outputs during development or use smaller local models for quick iteration, reserving GPT-4 calls for final runs.

- **Month 2: LLM Prompt & IR Schema Development** – *Milestone:* Define the IR format (Section 3) in detail and implement the IR compiler. During this phase, we will craft prompt templates for the LLM so it outputs in a structured way (possibly JSON). For example, develop few-shot examples: "Instruction: 'insert quickly' -> IR: {...}". Implement a parser that reads the LLM's text and populates a Python IR object. **Risk:** The LLM might not reliably follow format or might produce inconsistent info. *Mitigation:* Use few-shot learning and instructions like "Respond in JSON only with these fields...". If still unreliable, consider using a more structured approach (like a smaller semantic parser or GPT function calling). We will test on various instructions to refine the prompt. Another risk: The IR schema might need adjustments (we may realize new fields or different structure is better). That's fine – this month allows refining the design before locking it in.

- **Month 3: Mechanics Auditor Implementation** – *Milestone:* Have a working Auditor module that can process an IR and identify issues. We will implement the checks described (units, geometry, etc.) one by one and create unit tests for each (e.g., feed an IR with an impossible peg/hole size and see that the auditor flags it). We will artificially create some wrong IRs to ensure the auditor catches them. **Risk:** It might be hard to quantitatively decide some thresholds (e.g. what counts as "too tight" for jamming). *Mitigation:* Start with conservative thresholds based on literature (like if clearance=0 and friction>0.4, definitely jam potential). We can adjust based on simulation experiments later. Another risk is over-correcting (auditor changes IR in a way that deviates from user intent too much). We will document any automatic changes and perhaps limit how drastic they can be. For example, if something is highly inconsistent, maybe better to flag and request new IR rather than guess a fix. Our fallback for irreconcilable IR is to log an error (or in a real system, ask the user). Since we have the luxury of simulation, an error can just abort that trial and count as failure (with intention that in real usage, you'd iterate with user).

- **Month 4: Initial Integration & Simple Controller** – *Milestone:* Achieve first end-to-end run in simulation, possibly with a simplified controller (e.g. open-loop or PD control) before the full MPC. We will connect the pieces: take a test instruction, produce IR, audit it, then just do a simple insertion (like constant velocity push) using the IR parameters (speed, etc.), in MuJoCo. This is mainly to verify the simulation and IR values have the intended effect (e.g. if IR says speed 0.01 m/s, does our robot actually move at that speed; if max_force=5N, we could implement a crude check to stop when >5N to see what happens). **Risk:** We might see failures or issues in this step, such as the robot not inserting due to too strict parameters from IR. That's expected; it will highlight where adjustments needed. *Mitigation:* Use this as a debugging opportunity. Possibly tune defaults or auditor rules if, say, the IR was too conservative always stopping the robot. Another risk: hooking up MuJoCo properly to our Python control loop (ensuring real-time stepping with our controller in the loop). *Mitigation:* Use MuJoCo's scripting or the `mujoco` python API with a set of known patterns (the MuJoCo documentation and examples can guide event loops). If performance is an issue (e.g. Python loop can't keep up), we can reduce sim complexity or use bigger timesteps initially.

- **Month 5: MPC Controller Development** – *Milestone:* Implement the quadratic program formulation and get it running with CVXPY/OSQP in a test setting (e.g. controlling a 1D mass-spring system first, then integrate with the robot). We will start by deriving the linear model for the robot's insertion motion. Possibly fix the orientation and consider motion mostly in the insertion axis for initial simplicity. Formulate the QP with force constraints (simulate a spring contact as described). We will test the QP on known scenarios: for example, simulate a scenario in code where the peg is at the hole entrance, then solve QP to push it just enough to generate, say, 5N force and no more. Verify OSQP returns a solution quickly and it matches expectations. Then integrate this QP solve into the MuJoCo loop (replacing the simple controller from Month 4). **Risk:** Writing the QP correctly and getting it to solve every timestep is complex. We might encounter solver convergence issues or modeling mistakes (e.g. sign errors in constraints). *Mitigation:* We will test in offline simulations first. Also OSQP is reliable for convex problems, but if we set it up wrong we might inadvertently make something non-convex. We'll keep constraints linear and perhaps avoid integer mode variables (no complementarity directly). Tuning weights is another risk – if costs are not balanced, the controller might prioritize wrong things (like it might nearly violate force to reduce position error if weights are skewed). *Mitigation:* Start with intuitive weights and then systematically adjust after seeing performance. We can use a few scenarios to tune (e.g. a high friction case to ensure it respects force, a low friction case to ensure it still goes fast enough).

- **Month 6: Integrated System Testing and Refinement** – *Milestone:* Full system (LLM -> IR -> Auditor -> MPC -> MuJoCo) runs on a variety of test instructions successfully. We will extensively test instructions like "insert quickly", "insert slowly with max 5N", "insert the tight peg carefully", etc., and observe behavior. This month will likely involve debugging the interplay: for example, maybe the auditor made max_force 5N but the actual needed was 6N, resulting in a stall – how to handle? We might refine the auditor to consider dynamic adjustment (or allow slight overshoot via the controller slack). We also ensure that the system can handle when instructions are absent some info (the defaults should work). **Risk:** The LLM might produce inconsistent outputs for some phrasing. *Mitigation:* If we see this, we can harden the prompt or do a pre-processing (maybe parse the text ourselves for numbers as double-check). Another risk is time synchronization – the controller must run faster than real-time simulation. If OSQP or Python loop is too slow at 50Hz, we might need to reduce frequency or optimize (e.g. by using OSQP's warm-start and not using CVXPY overhead each time, instead use OSQP's update API). We will profile and if needed, drop horizon length or frequency. If truly necessary, writing the QP solve more directly (without CVXPY) is a backup, but OSQP's Python API can update matrices quickly. Given typical sizes, we expect it okay, but it's a risk to watch. Plan B would be to let MuJoCo run slower-than-real-time so that it doesn't matter (since it's simulation).

- **Month 7: Extensive Simulation Experiments** – *Milestone:* Run the full battery of trials with the system to collect data for evaluation. By now, the system should be stable, so we set up automated runs: e.g. 50 random initial conditions times a set of different instruction types (or fix instruction and randomize environment factors as described). We log all metrics. We also run the baselines (no IR, no auditor, no MPC) by disabling parts of code accordingly. This will generate raw data for metrics. **Risk:** A certain combination might still cause failure (e.g. extreme misalignment beyond our correction capability). *Mitigation:* That will be part of results (the system's limitations). If failures are common in some corner of the space that was supposed to work, we may decide to refine the controller or auditor a bit more here. For instance, if we see that with very high friction, success plummets, we could consider if the auditor should have recommended a different approach (like perhaps it should instruct a wiggle strategy if friction too high – though implementing that is complex and maybe out of scope). We'll likely document that as future work rather than solve it last-minute. Another risk is randomness causing variance

– we must ensure enough trials or fixed seeds for fair baseline comparison. We plan experiments such that each method sees the same set of random conditions for a meaningful comparison (e.g. use a fixed random seed list for scenario generation). That way differences are due to method, not luck.

- **Month 8: Data Analysis and Results Compilation** – *Milestone:* Calculate all evaluation metrics and produce tables/plots for the paper. Using the logs from Month 7, we will compute success rates, etc. This will involve writing analysis scripts (possibly using pandas or simply parsing log files). We also intend to create illustrative plots: e.g. a graph of force over time for a gentle vs aggressive instruction to show difference, or a comparison of two controllers. We will also count how many times the auditor intervened and in what way, summarizing those cases. **Risk:** Metrics might initially look underwhelming (e.g. maybe success isn't near 100%). *Mitigation:* Understand why – if there is a specific failure mode we missed, decide if we have time to address it (small tweak) or just report it. At this stage, we avoid major changes unless absolutely needed, focusing instead on clearly explaining results. Another risk is that baseline differences might be smaller than expected (implying our method is not significantly better). If that happens, we will honestly report it but also analyze why. It might be because baseline was already pretty good in sim; however, we can then emphasize other benefits like constraint satisfaction or the ability to handle different instructions (which baseline can't do). We will make sure to highlight any nuance (for example, baseline might have similar success but did so by using higher forces, whereas our method kept forces low as intended, which matters for safety).

- **Month 9: Paper Writing (Drafting)** – *Milestone:* A complete draft of the paper is written, including Introduction, Related Work, Methods, Experiments, and initial Discussion/Conclusion. We will structure the paper as outlined in Section 10. We plan to start writing earlier (some sections like Related Work can be drafted in Month 1-2 alongside research, Methods can be written in Month 5-6 as we implement). By Month 9, we integrate the results and polish the narrative. **Risk:** Writing often takes longer than expected, especially ensuring clarity and academic tone in English (since this is an academic style paper). *Mitigation:* Allocate sufficient time and possibly get internal reviews (peers or advisor feedback). We have to be mindful of deadlines if this is for a conference or similar. If submission deadline is earlier, adjust timeline accordingly. Another risk is missing some citations – we should ensure by this time all key references are included (we've maintained a list of those, see Section 10). If some results are surprising, we'll add discussion to explain them.

- **Month 10: Paper Revision and Submission** – *Milestone:* Finalize the paper and submit to a conference or journal (or deliver as required). This includes refining figures, reference checks, proofreading, and conforming to formatting guidelines (like IEEE or ACM templates, etc.). **Risk:** Last-minute issues like word count or figure formatting could arise. *Mitigation:* Buffer time for that, perhaps finish main content by early in month so we have a couple weeks for revisions. Another risk is if experiments had to be rerun or additional ones done after reviewers' feedback (if this is a revision of R1 as hinted). If so, we incorporate them as needed.

Throughout the timeline, we maintain a version control (e.g. Git) for code and a shared document for writing to track changes. If any phase falls behind (risk: maybe MPC took 2 months instead of 1 because of complexities), we have a few strategies: - Overlap some tasks (e.g. writing can start while experiments are still running). - Reduce scope if necessary: for instance, if full 3D MPC is too hard, perhaps constrain to mainly 1D insertion with orientation handled separately. But we believe the plan is feasible as is. - Use simpler alternatives temporarily to get results (e.g., if MPC proving too slow, we could discretize a few candidate forces and do a smaller QP or even PID with auditor adjustments to demonstrate concept – but that'd be last resort as it undermines the novelty).

Given the careful planning, we expect to hit the key milestones. The risk that LLM integration poses (since it's unpredictable) is mitigated by focusing on relatively straightforward outputs. Also, working in simulation avoids many real-world uncertainties (sensors noise, robot hardware faults) – so we can iterate faster and not worry about physical damage (the max_force limits ensure even in sim we don't get numerically unstable huge forces).

In conclusion, by following this timeline, we aim to have a completed and evaluated system by the end of Month 9, leaving time for thorough writing and any contingency. This phased approach ensures that critical components (like the controller) are developed on time and that integration issues are discovered early (Month 4-6) rather than at the last minute. We also allocate significant time for experiments and analysis, which are vital for a high-quality submission.

10. Paper Writing Structure and References (APA Style)

We plan to write the research paper in a formal academic style, either in English or bilingual as required. The structure of the paper will be as follows (with approximate sections):

- **Abstract:** A concise summary of the problem (language to action for contact-rich tasks), our approach (LLM to IR compiler, Mechanics Auditor, MPC controller) and key results (e.g. "We achieve X% success in peg-in-hole with constraints satisfied, improving success by Y% over baselines").

- **1. Introduction:** Introduce the context of combining large language models with robot manipulation, and highlight the challenge of ensuring physical consistency (contact mechanics) when following human instructions. We will mention how previous works have either not addressed low-level physical constraints or require heavy learning. Then we present our solution outline and contributions. For instance: *"This paper proposes a novel framework that compiles natural language instructions into an intermediate representation (IR) of contact mechanics, verifies it with a 'Mechanics Auditor' for physics consistency, and executes it using a constrained model-predictive controller. To our knowledge, this is the first integration of LLMs with real-time MPC for robotic assembly tasks, bridging high-level intent and low-level force control."* We will also mention the significance: enabling non-expert users to instruct robots in plain language for delicate tasks. Key references to cite here include recent LLM+robotics works (to show what's been done and what gap we fill) – e.g. **Ahn et al. (2022)** who used LLMs for high-level planning with value function grounding [21], **Liang et al. (2023)** Code-as-Policies for LLM generating robot code [22] [23], and perhaps the survey by **Bian et al. (2026)** to emphasize increasing interest in LLM-based task planning [24] [25]. We will position our work relative to these: unlike SayCan which picks from predefined skills, we *generate parameterized skills and ensure their correctness*; unlike Code-as-Policies which focuses on code generation, we introduce an explicit physical IR and auditor.

- **2. Related Work:** This section will delve deeper into categories of prior work:

- *LLMs in Robotics:* Cite works like Ahn et al. (2022) "SayCan" [21], Huang et al. (2022) "Language Models as Zero-Shot Planners" (Huang et al., 2022), **Vemprala et al. (2023)** on ChatGPT for Robotics (showing using prompting to get plans, but without formal verification) [26]. Also mention vision-language models like **Driess et al. (2023)** PaLM-E if relevant, and others such as **Shah et al. (2023)** LM-Nav for navigation. Emphasize what they achieve and their limitations (e.g. lack of hard constraint enforcement, or need for value function). We will cite these appropriately in APA style.

- *Intermediate Representations & Skill Specification:* Discuss the idea of using structured representations or programs as an output of LLMs. E.g., Code-as-Policies (Liang et al., 2023) which treats LLM output as a program policy [22] [27], and other works that attempted graphs or formal languages. We'll mention how **Tang et al. (2025)** (SEAM) used a context-free grammar to structure VLM outputs (Tang et al., 2025) – though that was vision-language, it shows benefit of a defined vocabulary. Our IR is domain-specific for mechanics.
- *Peg-in-hole and force control:* Summarize classical approaches to peg-in-hole assembly. Cite **Whitney (1982)** as the foundational work on passive compliance devices (RCC) for peg insertion [28] and perhaps **Mason (1981)** on compliant motion. Then cite some recent works: e.g. **Beltran-Hernandez et al. (2020)** who used deep RL with variable compliance for peg-in-hole (showing modern learning approach) [29] [30], and **Zhang et al. (2019)** on active compliance disassembly [31] [28]. This positions how our method differs: we use model-based control (MPC) with an LLM-driven spec, instead of learning from scratch. Also mention how constraint-based controllers (e.g. QP controllers) have been used in walking and manipulation for contact – we could reference something like **Bouget et al. (2015)** or others, but an easier reference is the concept that friction cones are linearized for QP in legged locomotion [10], which we also do. Possibly cite **Caron (2020)** for friction pyramids.
- *Optimization in contact-rich control:* Cite works like **Posa et al. (2014)** on contact implicit optimization, or recent **Le Cleac'h et al. (2024)** contact-implicit MPC [32] [33] to show what state-of-art MPC for contact can do. Those are advanced; we do a simpler approach but it's informed by these (we mention using soft constraints rather than full complementarity as per Todorov's smooth contact model [8]). Also mention tools like **OSQP** (Stellato et al., 2020) as enabling fast QP solving for robotics, which we leverage.

- We will ensure all references are cited in-text in APA style (Author, Year). For example: *"Prior work by Ahn et al. (2022) combined LLMs with learned value functions to choose feasible skills [1], while Liang et al. (2023) generated robot code from prompts [22]. However, these approaches did not enforce low-level physics constraints. Our work addresses this gap…".*

- **3. Methodology:** This will be the core technical section, likely split into subsections:

- 3.1 **System Overview:** Describe the pipeline (as in section 1 of this plan). A figure of the system architecture will be included (with blocks for LLM, IR, Auditor, Controller, Simulation). This helps reader see how components connect.
- 3.2 **Intermediate Representation:** Explain the IR format, fields, and an example. (From Section 3 of plan). Possibly present Table 1 listing IR fields, types, default values for clarity. This acts as a formal spec of how language translates to numbers.
- 3.3 **Mechanics Auditor:** Detail the auditor's function, maybe as an algorithm box. Include examples of checks (we might put a small listing of pseudocode or bullet list of checks). Emphasize how it ensures feasibility (citing any relevant concept like SayCan again for analogous role).
- 3.4 **Constrained MPC Controller:** Provide the formulation of the MPC. We will include the main equations: the cost function (in math form) and constraints (e.g. $F_n \leq F_{\max}$, friction pyramid inequalities). Possibly reference MuJoCo's contact model or friction linearization as justification [10]. We'll keep it at a level suitable for a robotics audience, referencing QP controllers used in e.g. locomotion (since they might be familiar). Also mention computational aspects (OSQP solving at 50Hz etc.).

- 3.5 **Integration:** Briefly mention how it all ties together in real-time: e.g., the frequency of the loop, how the auditor is run once per instruction, etc. This might also include any implementation details like using OpenAI API, but those could also be in experiments section.

- **4. Experiments:** This corresponds to sections 6-8 of our plan. It describes the simulation setup and results.

- 4.1 **Experimental Setup:** Explain the peg-in-hole simulation: robot, peg/hole dimensions, friction randomization, control frequency. (From Section 6). Possibly a figure of the simulation or a picture of the MuJoCo scene [34] could be included. Also define the evaluation metrics here formally (success criteria, etc., as in Section 7). This ensures readers know how we measure improvement. We'll reference any standard benchmarks if any (peg-in-hole is common but not a standardized benchmark like say sliding a block).
- 4.2 **Results:** Present the performance of our full approach. E.g., "The system achieved 96% success rate under random misalignments up to 5° and friction 0.1–0.5. The instructed force limits were respected in 100% of trials, with peak forces staying 10–20% below the set thresholds on average." We will include a table or plot of metrics. Also, perhaps a qualitative example: "When instructed to insert 'quickly', the system completed insertion in 2.1s with a peak force of 15N; when instructed 'gently', it took 3.5s but peak force was only 8N, demonstrating compliance with the different requirements." We can illustrate this with a force-vs-time graph for those two cases. We'll also note how often auditor intervened, e.g., "In 2 out of 50 trials, the auditor adjusted the IR (both times because the LLM set an unrealistically low force); those trials still succeeded after adjustment, whereas without adjustment they would have failed to insert." This highlights the auditor's effectiveness.
- 4.3 **Baseline Comparisons:** Present ablation results. Perhaps a bar chart showing success rate of Full vs No IR vs No Auditor vs No MPC (and any other relevant baseline). And similarly for peak force or violations. We will describe: *"Without the IR (using a fixed policy), success dropped to 70% since in many cases the peg jammed due to no adaptation to friction. Without the auditor, one scenario with conflicting constraints led to failure (success 98% vs 100% with auditor) and an average of 2 constraint violations per 10 trials were observed. Without MPC, using a stiff PID controller, success was 85% and peak forces were 2-3x higher, occasionally exceeding limits (in 20% of trials). These results confirm that each component – the IR guidance, auditor, and MPC – contributes to performance and safety."* We will ensure to cite any relevant work for baselines if applicable (for instance, compliance control methods or RL methods that got certain success rates, though direct comparison may be tricky since conditions differ). We might compare qualitatively to Beltran-Hernandez 2020's success if they reported (they did some real experiments too).

- If space permits, we might also include a short *"Ablation analysis"* subsection discussing one or two specific cases in detail (like an illustrative failure when auditor is off).

- **5. Discussion:** Here we interpret the results, mention limitations, and possible extensions. For example, discuss how well the system might transfer to real hardware (issues like sim-to-real gap, which our method might mitigate somewhat because it's model-based and has explicit safety constraints – but friction uncertainty etc. is always an issue). Also mention limitations: our LLM is not learning new skills, it's limited by its prompt and cannot handle instructions beyond the IR scope. We could discuss how one could extend the IR to other tasks or incorporate vision (if the hole location was unknown, etc., we assumed known pose). We'll also address that our approach requires a good physics model; if the model is wrong, the auditor or controller might be misled (though auditor uses fairly general rules and controller is robust within reason). Another point: the current system is single-turn (the instruction is given once). In future, one could imagine a dialogue where if the auditor flags an issue, the system asks the user for clarification – that would improve user experience. Safety-wise, we can mention how the approach ensures physical safety by design, which is an advantage when moving to real robots (the robot should never exert more force than specified, protecting both the robot and environment). We'll also relate back to the literature: e.g. "While deep RL approaches (Beltran-

Hernandez 2020) can learn insertion, they often require thousands of trials and don't allow easy constraint specification; our method achieves the task in a zero-shot fashion with constraints, thanks to the LLM and model-based controller, albeit currently only in simulation." This sort of contrast can be made.

- **6. Conclusion:** A brief recap of what we achieved and the key takeaways. E.g., *"We demonstrated a novel compiler from language to robot actions that incorporates physical reasoning. On a peg-in-hole task, the system followed human instructions (like enforcing low force) and achieved high success without manual tuning. This illustrates a pathway to instruct robots in natural language for delicate tasks with built-in safety verifications."* Mention future work: testing on a real robot, extending to more complex assembly, combining vision for finding the hole, or using the framework for other contact tasks (like sliding, stacking). Possibly mention exploring fine-tuning the LLM for even better structured output, or integrating learning to improve the IR parameters over time.

- **References:** We will include a list of references in APA format. Key references will include those discussed above. For example:

References (APA style):

- Ahn, M., Brohan, A., Brown, N., Chebotar, Y., et al. (2022). *Do As I Can, Not As I Say: Grounding Language in Robotic Affordances.* arXiv:2204.01691. (Introduced SayCan, using value functions to filter LLM actions)
- Beltran-Hernandez, C. C., Petit, D., Ramirez-Alpizar, I. G., & Harada, K. (2020). *Variable compliance control for robotic peg-in-hole assembly: A deep-reinforcement-learning approach*. Applied Sciences, 10(19), 6923. (Peg-in-hole via RL with compliance, for baseline comparison)
- Bian, S., Zhang, Y., Tian, G., Miao, Z., Wu, E. Q., Yang, S. X., & Hua, C. (2026). *Large language model-based task planning for service robots: A review*. Biomimetic Intelligence and Robotics, 100274. (Survey of LLMs in robotics, provides context and taxonomy)
- Diamond, S., & Boyd, S. (2016). *CVXPY: A Python-embedded modeling language for convex optimization*. Journal of Machine Learning Research, 17(83), 1–5. (CVXPY software used for our MPC/QP implementation)
- Freeman, C. D., Frey, E., Raichuk, A., Girgin, S., Mordatch, I., & Bachem, O. (2021). *Brax – A differentiable physics engine for large scale rigid body simulation*. arXiv:2106.13281. (Brax engine, considered for differentiability)
- Liang, J., Huang, W., Xia, F., Xu, P., Hausman, K., Ichter, B., Florence, P., & Zeng, A. (2023). *Code as Policies: Language Model Programs for Embodied Control*. In Proceedings of Robotics: Science and Systems (RSS). (LLM generates Python code for robot; we cite to compare our IR approach)
- Stellato, B., Banjac, G., Goulart, P., Bemporad, A., & Boyd, S. (2020). *OSQP: An operator splitting solver for quadratic programs*. Mathematical Programming Computation, 12(4), 637–672. (Optimizer used for MPC QP)
- Todorov, E., Erez, T., & Tassa, Y. (2012). *MuJoCo: A physics engine for model-based control*. In 2012 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS) (pp. 5026-5033). (MuJoCo simulator reference)
- Vemprala, S., Bonatti, R., Bucker, A., & Kapoor, A. (2023). *ChatGPT for Robotics: Design Principles and Model Abilities*. arXiv:2303.17071. (Explores prompting ChatGPT for robotics tasks; shows interest in conversational robotics, we cite as related LLM usage)
- Werling, K., Exarchos, I., Omens, D., & Liu, C. K. (2021). *A fast and feature-complete differentiable physics engine for articulated rigid bodies with contact constraints*. arXiv:2103.16021. (Nimble Physics engine, differentiable contacts – relevant to mention differentiable simulators)

(Note: The above references include the key ones discussed; in the actual paper, we will include all citations used throughout the text in APA format, sorted alphabetically by author.)

We will ensure the citation style is consistent (e.g. using parentheses with author last names and year for in-text, and full details in the References section). Important references like Ahn et al. 2022 and Liang et al. 2023 will be highlighted early as they are central comparisons. We will also likely cite additional references in related work as needed (the above list is representative, not exhaustive).

Finally, we will cross-check each reference for accuracy (titles, venues, years) and ensure all are actually cited in the text to avoid dangling references. The writing will maintain a formal tone, use past tense for describing our experiments, and present tense for established knowledge. We will avoid colloquialisms, and ensure clarity even with the mix of concepts (we'll carefully define any notation or jargon we introduce).

By following this structure and citing relevant prior art, the paper will clearly convey our methodology, situate it in the context of existing research, and demonstrate its validity through comprehensive experiments. The APA-style references provide the scholarly grounding and allow interested readers to find those works for deeper reading. This completes the research plan, covering the entire pipeline from concept to evaluation and documentation.

---

1 21 26 Agentic LLM-based robotic systems for real-world applications: a review on their agenticness and ethics - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC12402697/

2 3 What is Nimble? — Nimble Physics 0.4.0 documentation
https://nimblephysics.org/docs/intro.html

4 [PDF] Differentiable Convex Optimization Layers - Stanford University
https://stanford.edu/~boyd/papers/pdf/diff_cvxpy.pdf

5 Citing OSQP
https://osqp.org/citing/

6 [PDF] OSQP: an operator splitting solver for quadratic programs
https://dspace.mit.edu/bitstream/handle/1721.1/131868/12532_2020_179_ReferencePDF.pdf?sequence=1&isAllowed=y

7 28 31 Peg–hole disassembly using active compliance - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC6731726/

8 13 14 15 Computation - MuJoCo Documentation
https://mujoco.readthedocs.io/en/stable/computation/index.html

9 10 11 12 Friction cones
https://scaron.info/robotics/friction-cones.html

16 17 19 20 29 30 Variable Compliance Control for Robotic Peg-in-Hole Assembly: A Deep-Reinforcement-Learning Approach
https://www.mdpi.com/2076-3417/10/19/6923

18 Robotics Gym & Experiments - Tiny struggles
https://www.tinystruggles.com/posts/robotics_gyms_and_experiments/

22 23 27 [2209.07753] Code as Policies: Language Model Programs for Embodied Control
https://arxiv.org/abs/2209.07753

24 25 Large language model-based task planning for service robots: A review - ScienceDirect
https://www.sciencedirect.com/science/article/pii/S2667379726000021

32 33 msl.stanford.edu
https://msl.stanford.edu/papers/le_cleach_fast_2024.pdf

34 Snapshot of peg-in-hole scenario for MuJoCo - ResearchGate
https://www.researchgate.net/figure/Snapshot-of-peg-in-hole-scenario-for-MuJoCo-the-peg-is-pressed-down-to-insert-the-peg_fig9_366888254