#7_UR
Autonomous Robot Programming Agent (simulated + physical)

Context

The future of robot programming is agentic. Robots that receive a high-level task description and autonomously generate, test, evaluate, and refine their own programs, without a human writing a single line of code. This is made possible by combining large language models with robot simulation and real-time state feedback.

In this case, students will build an autonomous robot programming agent: a system where an LLM receives a task description, generates robot actions, executes them in simulation, evaluates the outcome, and iterates until the task succeeds before deploying the validated program to a real UR10e. The key is not just generating code, but building the closed-loop reasoning that makes the agent self-correcting.

The agentic reasoning can be implemented in different ways: native LLM tool use, structured output parsing, a state machine with LLM-driven transitions, or any other approach the students prefer.

Challenge

Build an autonomous robot programming agent consisting of four components:

1. System prompt and task interface: Design the prompt template that defines the robot's capabilities, available tools, and how the LLM should reason about tasks. The student defines how tasks are described and how the LLM formulates action plans.

2. Orchestrator loop: A Python script that manages the agent cycle. It sends the task and current state to the LLM, parses the response into executable robot commands, executes them via the provided MCP server on URSim, and feeds results back to the LLM for the next iteration. The parsing and execution logic can be implemented in multiple ways (native tool use, regex parsing, structured JSON output, or a state machine).

3. State serializer: Convert raw RTDE data (100+ numeric signals) into concise, human-readable text that the LLM can reason about. For example: "TCP at [0.30, 0.20, 0.10], 15mm from target, robot stationary."

4. Evaluation function: After each action, determine whether it succeeded or failed. Did the TCP reach the target? Was there a protective stop? This function feeds structured feedback to the LLM so it can self-correct on failure (e.g. "protective stop on joint 2, try a different approach angle").

Demo scope:

- In URSim: the agent plans and executes motion tasks (multi-waypoint trajectories, geometric shapes). Success is measured by comparing the executed TCP path against the intended trajectory.
- On the real UR10e: the same agent runs against the real robot. Pick-and-place can be demonstrated using a virtual world model where the orchestrator tracks object state (e.g. "block_A at [0.3, 0.2, 0.1], status: on table"). Gripper actions can be simulated using digital I/O or a timed wait.

Keywords: Agentic AI, LLM reasoning, autonomous programming, closed-loop control, sim-to-real deployment

Tiers

Bronze: Configure the LLM connection and run the provided agent on the example task (move robot to home position). The agent must complete the task within the maximum iteration limit. Show the full agent log: each observation, LLM reasoning, action taken, and evaluation result.

Silver: Design and implement one custom motion task with a specific evaluation criterion. For example: trace a geometric shape (square, triangle, circle) using waypoints. Write a custom evaluation function that measures how closely the executed TCP path matches the intended trajectory. The agent must complete the task autonomously.

Gold: Implement plan adaptation: when an action fails (e.g. a protective stop), the agent must reason about what went wrong and adjust its approach rather than retrying the same action. Add a summary of past attempts to the LLM context so the agent learns from its mistakes within an episode. Demonstrate recovery from at least one failure mode (e.g. protective stop triggered by an unreachable waypoint).

Diamond: Extend the agent with a virtual world model: the orchestrator tracks named objects as coordinate sets with state (e.g. "block_A at [0.3, 0.2, 0.1], status: on table"). Implement a pick-and-place task where the agent reasons about object locations, plans grasp and place sequences, and updates the world model after each action. Validate on the real UR10e (gripper simulated via digital I/O or timed wait), or demonstrate the full reasoning loop in URSim with the virtual world model.

Tools, methods and materials

LLM backend (free): Students are responsible for sourcing their own LLM access. Free options include GLM-5.1 via NVIDIA NIM (free API key), Z.AI direct, Puter.js (zero-config), and Modal.com. Students may also use any LLM they already have access to. The agent design should not depend on a specific model's tool use capabilities, as different models handle function calling differently.

Robot interface: a pre-built MCP server (provided by UR) that exposes UR10e tools. Students use this as a black box and focus entirely on the agentic reasoning layer.

Agent framework: Python. No specific framework is required. Students may use httpx or any OpenAI-compatible API client for LLM communication.

Development environment: URSim Docker container provided via docker-compose with the MCP server pre-configured. All development and testing happens against URSim on the student's own laptop.

Validation: two UR10e robots are available for deploying and testing the autonomous agent on real hardware.

From UR, the team will receive a docker-compose setup for URSim with the MCP server pre-configured, an orchestrator skeleton with the agentic loop structure, example task definitions, documentation on the available MCP tools, and specifications for the state serializer and evaluation function (describing expected inputs, outputs, and example scenarios, but no implementation). Students build the state serializer, evaluation function, and custom tasks themselves. Two UR10e robots are available for real-hardware testing. UR will be available to discuss details along the way.

View pdf

Back to industry cases