RobotManager & PLC Integration – Full Technical Manual

This document is a focused, full technical explanation of the core orchestration logic in your system:

the Python side that parses the DSL, constructs runtime structures, and drives either the ROS/MoveIt

coordinator or the real PLC/ADS-based cell.

It is centered around three main components:

1. RobotHTTPClient – thin HTTP gateway to FastAPI (ros_http_bridge2).

2. PLCClient – ROS2 action/service client wrapper to talk to the Beckhoff/Prometheus PLC.

3. RobotManager – the central brain that parses the DSL tree, validates it, stores all data (trays, units,

screws, poses, parameters, workflow), and executes the workflow step-by-step.

Below we document each class and the overall lifecycle in detail so you can fully understand and explain it.

1. RobotHTTPClient – HTTP Gateway to ros_http_bridge2

RobotHTTPClient is a small helper class whose only responsibility is to send HTTP POST requests to your
FastAPI server (ros_http_bridge2) running at BASE_URL (by default http://localhost:8000). It provides
specialized methods for specific endpoints and a generic method for arbitrary actions.

1.1 Base URL and Purpose

BASE_URL = "http://localhost:8000"

Every method in RobotHTTPClient constructs a URL of the form f"{self.base_url}/endpoint" and then sends JSON
payloads with requests.post. This class hides the details of HTTP and provides clean Python methods that
RobotManager can call.

1.2 Methods

• send_init_parameters(self, params)

- POST /init_parameters with body {"params": params}.
- Used at startup to push all DSL-derived parameters to the ROS/C++ side.
- Returns a boolean: res.json().get("success", False).
- If the HTTP call fails (connection error, timeout, etc.) it prints an error and returns False.

• confirm_workflow(self, task_description)

- POST /confirm_workflow with body {"task_description": ...}.
- No timeout – this is intentional so the FastAPI side can block until a user answers Y/N.
- Returns boolean "success" field from JSON.
- Used by RobotManager for manual steps when control_type == "manual" in simulation mode.

• add_tray2(self, tray_name: str, object_name: str)

- POST /add_tray2 with {"tray_name": ..., "object_name": ...}.
- Used to spawn a tray with a logical object (e.g., AOCS_Tray + specific component).

• index_action(self, endpoint: str, index: int)

- Generic helper for endpoints that expect {"index": }, like /pick_tray, /position_tray, etc.
- RobotManager calls this with human-friendly action names, already converted to snake_case.
- Returns decoded JSON or {} on error.

• generic_params_action(self, action: str, params_list)

- Generic helper for actions that expect {"params": [...]}, where "action" is given in CamelCase.

- It converts CamelCase → snake_case (e.g., "RechargeSequence" → "recharge_sequence") and POSTs to that URL.

• internal_screw_by_num(self, index: int, screw_num: int)

- Specialized helper for /internal_screw_by_num with payload {"index": int(index), "ScrewNum": int(screw_num)}.

- Used by RobotManager when executing InternalScrewUnitHole for a particular unit + hole.

• execute_action(self, action, data)

- Very general helper for POSTing to "/" with JSON body "data".

- It does CamelCase → snake_case conversion internally and posts to that endpoint.

2. PLCClient – PLC / ADS / ROS2 Integration

PLCClient hides the complexity of interacting with the Beckhoff/Prometheus PLC via ROS2 action and service calls.

It assumes the prometheus_req_interfaces and prometheus_req_py packages are installed and available.

It is only used when RobotManager.mode == "real". In simulation mode, RobotManager never creates a PLCClient.

2.1 Lazy ROS Imports and Initialization

PLCCLient._load_ros_dependencies() dynamically imports:

- rclpy
- ActionClient
- prometheus_req_interfaces.action.CallFunctionBlock
- prometheus_req_interfaces.srv.SetScrewBayState
- prometheus_req_interfaces.msg.ScrewSlot, Offset
- std_msgs.msg.Empty
- prometheus_req_py.ADS.utils.msgType

These imports are cached in PLCClient._ros_imports. The class keeps a static flag _rclpy_initialized so

that rclpy.init() is only called once, even if multiple PLCClient instances are created.

The constructor then:

- Creates a ROS2 node called "dsl_plc_client".
- Creates an ActionClient for CallFunctionBlock on "CallFunctionBlock".
- Creates publishers for "errorCheckAck" and "offset".
- Creates a client for SetScrewBayState on "setScrewBayState".
- Configures an ATS IP (ats_ip, default 10.10.10.100) for requesting screw/tray correction offsets.

2.2 BLOCK_PARAM_SCHEMA – Mapping DSL Params to PLC Goal Fields

BLOCK_PARAM_SCHEMA defines how high-level parameters map to the CallFunctionBlock.Goal fields for specific PLC

function blocks. For example:

- "loadTray": [("bool_param1", "load", "bool")]

Means the PLC block "loadTray" expects a bool field "bool_param1" that is provided from DSL as key "load".

- "screwTight":

[("float_param1", "offset_x", "float"),

("float_param2", "offset_y", "float"),

("float_param3", "offset_z", "float"),
("int_param1", "target", "int"),
("int_param2", "focal_plane", "int"),
("int_param3", "recipe_id", "int"),
("bool_param1", "inside_area", "bool")]
This is a richer PLC function block that accepts offsets, target index, camera focal plane, recipe id, etc.

2.3 call_block() – The Main PLC Entry Point

call_block(self, block_name: str, params: Optional[Dict[str, Any]] = None) -> Dict[str, Any]:
- Resolves the block_name using resolve_block_name() against the allowed BUILDING_BLOCKS.
- Stores params in self._active_context for later use in feedback/picture handling.
- If the resolved block is "setScrewBayState", calls _call_set_screw_bay_state().
- Otherwise builds a CallFunctionBlock.Goal using _build_goal() and sends it via _send_goal().
- Returns a small Python dict: {"success": bool, "state": ..., "msg": ...}.

2.4 _build_goal() and Type Coercion

_build_goal(self, block_name, params):
- Creates a CallFunctionBlock.Goal() message.
- Sets goal.function_block_name = block_name.
- Looks up the schema in BLOCK_PARAM_SCHEMA.
- For each field in the schema (e.g., ("float_param1", "offset_x", "float")), it:
* Checks that the required parameter ("offset_x") exists in params.
* Coerces it to the correct type via _coerce_int(), _coerce_float(), or _coerce_bool().
* Attaches it to the goal as goal.float_param1, goal.int_param1, etc.
This ensures that DSL/RobotManager can provide flexible Python values (strings, ints, floats, bools) and they will
be safely converted to the types expected by the PLC.

2.5 _send_goal() – Action Client Lifecycle

_send_goal(self, goal):
- Waits for the CallFunctionBlock action server to be ready (wait_for_server(5 seconds)).
- Sends the goal asynchronously with feedback_callback=_feedback_callback.
- Spins rclpy until the send_future is complete, then checks if the goal was accepted.
- Waits for the result via get_result_async() and spin_until_future_complete().
- Returns a dict with keys: success, state, msg based on the action result message fields.
Feedback is handled by _feedback_callback(), which can:

- Auto-acknowledge error checks by publishing an Empty() on errorCheckAck.

- Handle picture requests (ASK_PICTURE_SCREW, ASK_PICTURE_VCHECK) by calling _handle_picture_request().

2.6 Picture Offset Handling (calculate_picture_offset)

When the PLC requests a picture-based correction, _handle_picture_request() calls calculate_picture_offset(), which:

- Builds a parameter dict: {"calibrationPlane": int, "roiId": int, "findScrew": bool}.

- Chooses command GetScrewCorrection or GetTrayCorrection depending on msg_type.

- Sends an HTTPS GET request to https://ATS_IP/Command with the parameters (TLS verification disabled).

- Parses the JSON (or string representation) response.

- Checks DataValid field. If False, returns no correction.

- Otherwise extracts TranslationX, TranslationY, Rotation and returns them.

The offset data is then published as an Offset message via self._offset_pub and logged.

2.7 _call_set_screw_bay_state()

This method is used for PLC side management of screw bays.

It expects params["slots"] to be a list of either dicts with keys

[max_idx_x, max_idx_y, next_idx_x, next_idx_y] or 4-tuples of values.

It:

- Waits for the setScrewBayState service.

- Fills a request with the proper ScrewSlot messages.

- Calls the service asynchronously and spins until completion.

- Logs and returns a small dict summarizing success.

3. RobotManager – Central Orchestrator

RobotManager is the core orchestrator. It is constructed with a Lark parse tree (parsed_tree) of the DSL and:

- Parses all DSL sections (mode, locations, trays, tray_step_poses, main_poses, parameters, assembly).
- Stores them into internal structures (trays, unit_lookup, tray_step_poses, main_poses, named_poses, parameters).
- Validates trays and workflow symbols.
- Sends an InitParameters payload to the ROS/HTTP bridge in simulation mode.
- Creates a PLCClient in real mode.
- Executes the workflow step-by-step, with optional manual confirmations.

3.1 Constructor (__init__)

__init__(self, parsed_tree):

- Creates a RobotHTTPClient instance.
- Sets mode = "simulation" by default, plc_client = None.
- Initializes many lists/dicts used either in legacy form or for clarity:

tip_offset, tray_angles, tray_down_steps, operator_steps, rotation_steps, new_tray_steps,

vector3, vector2, int_list, tray_step_poses, current_tray_step, location_order,

location_name_to_index, main_poses, named_poses, parameters.

- Calls self.build_workflow(parsed_tree) to populate:

self.workflow, self.trays, self.locations.

- Constructs a params dict containing:

tray_step_poses, named_poses, main_poses, trays, and a number of parameter arrays

(tip_offset, tray_angles, tray_down_steps, operator_steps, rotation_steps, new_tray_steps,

tray_heights, origin_to_bottom, new_dummy, tray_init_offset, tray_pose_operator, tray_final_pose).

If mode == "simulation":

- Prints that it is sending InitParameters via HTTP.
- Calls http_client.send_init_parameters(params). If result is False, raises RuntimeError.
- Computes a special main pose "Recharge" from the "Table" main pose and the tip_offset, mirroring your old C++ logic:

Recharge.x = Table.x - 0.55160 + tip[0]

Recharge.y = Table.y - 0.17248 + tip[1]

Recharge.z = Table.z + 0.82025 + 0.051 + tip[2]

(roll, pitch, yaw = 0).

- Saves this new Recharge in main_poses.
- Prints DSL runtime parameters, then calls send_init_parameters(params) again (so ROS/C++ get the updated Recharge).

If mode == "real":

- Skips HTTP init and instead constructs a PLCClient (self.plc_client = PLCClient(logger=self._plc_log)).

3.2 build_workflow() – Parsing the DSL Tree

build_workflow(self, tree):

This is the heart of the DSL parsing logic. It walks the Lark parse tree and:

- Reads mode_definition: sets self.mode to "simulation" or "real".

- Reads locations_definition: builds locations dict and maps location names to indices.

- Reads trays_definition: for each tray:

* Reads tray name.

* Accumulates attributes (tray_line, units_def, screws_def, height_def, initial_pose, operator_pose, final_pose).

* For units_def: calls _parse_unit_tree() to extract per-unit data (including screw blocks, pose_index).

* For screws_def: builds quantity/type pairs.

- Reads tray_step_poses_definition: builds a simple list of 6D poses, each being [x, y, z, r, p, y].

- Reads main_poses_definition: builds main_poses mapping pose name → 6D values.

- Reads parameters_definition:

* For each named_pose_entry: populates named_poses.

* For vector3, vector2, vector6: collects floats in parameters[param_name].

* For int_list or float_list: collects integer or float lists in parameters[param_name].

- Reads assembly_definition:

* For each command: reads control_type (manual/auto) and action node.

* Converts action grammar name to CamelCase (via to_camel_case).

* Extracts named arguments via _named_args_to_dict().

* If no named arguments, falls back to a list of positional parameters using _literal_from_node().

* Appends a workflow entry: {"control_type": ..., "action": ..., "params": ...}.

At the end:

- If mode == "simulation":

* Calls _validate_trays(trays) to enforce tray/unit structure with a Pydantic TrayModel.

* Calls _validate_workflow_symbols(workflow) to check that units, trays, named poses, and parameters referenced

in the workflow are defined and consistent.

- If mode == "real":

* Initializes tray_models, unit_lookup, tray_names, unit_names but does not validate via Pydantic (different use-case).

The method returns workflow, trays, locations, which are stored in self.workflow, self.trays, self.locations.

3.3 Unit and Screw Block Parsing

_parse_unit_tree(self, unit_node):

- Iterates over "unit_field" child nodes.
- Each unit_field has a key token (e.g., "name", "pose_index", "screws") and a value node.
- Uses _parse_unit_value() to handle ints, strings, screw_block, int_list, float_list, or vector6.
- Returns a dict with the full unit description (e.g., {"name": "MTQ12", "pose_index": 0, "screws": {...}}).

_parse_screw_block(self, block_node):

- Walks over fields (manual_field, auto_field, positions_field).
- manual_field: extracts a list of manual_indices.
- auto_field: extracts a list of auto_indices.
- positions_field: collects all vector6 children as screw positions.
- Returns a dict: {"manual_indices": [...], "auto_indices": [...], "positions": [[x,y,z,r,p,y], ...]}.

3.4 Validation: _validate_trays() and _validate_workflow_symbols()

_validate_trays(self, trays):

- Uses TrayModel (from dsl_models) to validate each tray payload.

- Builds tray_models (name → TrayModel instance).

- Builds unit_lookup mapping unit_name → {tray, pose_index, allowed_screws, definition}.

- allowed_screws is computed from the union of manual_indices and auto_indices defined in the screws metadata.

- Collects errors if a unit is duplicated, missing pose_index, or tray validation fails.

- If any errors exist, raises DSLValidationError with a list of textual diagnostics.

After a successful validation:

- self.tray_names = set of tray names.

- self.unit_names = set of unit names.

_validate_workflow_symbols(self, workflow):

- Builds a set of known symbols combining tray_names, unit_names, named_poses keys, and parameter keys.

- For actions defined in ACTION_PARAM_SCHEMAS (e.g., AddTray, InternalScrewUnitHole, CallBlock):

* Ensures that required parameters exist and are of the expected type (int or str).

- For InternalScrewUnitHole, also checks:

* unit is known.

* hole index is inside the unit's allowed_screws set.

- For list-based params: warns if string values don't match any known tray/unit/named pose/parameter.

- If any hard errors are found (missing required parameters or invalid screw hole), raises DSLValidationError.

3.5 execute_task() – Executing a Single Workflow Step

execute_task(self, task, module_name="my_python_pkg.functions"):

Input: a workflow command dict of the form:

{ "control_type": "manual" or "auto",

"action": "SomeAction",

"params": either a dict or a list }

1) It prepares:

- named_params: if params is a dict, uses it directly;

if params is a list, tries to interpret it as [key, value, key, value,...] via _list_to_named_params().

- positional_params: if list→dictionary conversion fails, then the whole list is used as positional_params.

2) Manual confirmation:

- If task["control_type"] == "manual" and mode == "simulation", it sends a confirm_workflow request:

"Execute '' with parameters ?"

- If the user says no, the action is skipped.

3) Real mode:

- If mode == "real", it calls _execute_real_task(action, named_params, positional_params) and returns.

4) Simulation mode – action routing:

- Special case: AddTray

* Requires named "tray" and "object"; calls http_client.add_tray2(tray_name, object_name).

- Index-based actions: PickTray, PositionTray, OperatorPositionTray, RechargeSequence,

InternalScrewingSequence, ExternalScrewingSequence, PlaceTray.

* Extracts an index either from named "index", from "location" name (mapped via location_name_to_index),

or from the first positional param.

* Calls index_action("snake_endpoint", index).

* Logs tray_step_pose[index] if available.

* Maintains self.current_tray_step:

- After PositionTray and InternalScrewingSequence success: self.current_tray_step = index.

- After PlaceTray success: self.current_tray_step = None.

- InternalScrewUnitHole:

* Requires a unit name and a hole number.

* Looks up the unit in unit_lookup.

* Confirms the hole is in the allowed_screws set.

* Gets tray_index from the unit's pose_index.

* If current_tray_step != tray_index:

- Automatically triggers a PositionTray call to align the tray.

- Updates current_tray_step on success.

* Logs tray_step_pose[tray_index] and calls http_client.internal_screw_by_num(tray_index, hole).

- Fallback actions:

* For any other action:

- If params is a list: uses it as fallback_params.

- If params is a dict: wraps it in a single-element list [params].

- Calls http_client.generic_params_action(action, fallback_params).

3.6 execute_workflow() – Running the Whole Program

execute_workflow(self):

- Creates a queue.Queue.

- Pushes all tasks from self.workflow into the queue.

- While the queue is not empty:

* Pops one task.

* Calls self.execute_task(task).

* Marks task as done.

- After all tasks are processed, prints "Workflow completed successfully."

This is a simple sequential execution loop, but because each task may internally block on manual confirmation

or HTTP/ROS feedback, it can still represent a complex interactive/robotic sequence.

4. Real Mode vs Simulation Mode – Behavior Summary

RobotManager has two operational modes, controlled by a "mode_definition" in the DSL or the default:

1) Simulation mode (default):

- Creates no PLCClient.

- Fully validates trays and workflow via Pydantic and DSLValidationError.

- Sends the full InitParameters payload to ROS via HTTP.

- Executes all workflow actions using RobotHTTPClient / FastAPI / ROS / C++ coordinator.

2) Real mode (mode: "real") in DSL:

- Skips InitParameters HTTP calls.

- Constructs a PLCClient.

- execute_task() routes only CallBlock actions to PLC (_execute_real_task). Other actions are currently skipped.

- PLCClient.call_block(...) uses the building blocks in BUILDING_BLOCKS and the BLOCK_PARAM_SCHEMA to map

DSL parameters onto PLC function block inputs.

This separation ensures that real hardware operations are only triggered when explicitly requested in the DSL and

that simulation remains safe and fully ROS/MoveIt-driven.

5. Mental Model: How Everything Fits Together

To truly understand RobotManager, keep this mental picture:

- The DSL describes WHAT to do: trays, units, screws, poses, and a sequence of actions.

- RobotManager parses that, validates it, and becomes the runtime representation of the DSL.

- In simulation mode:

* RobotManager is a conductor: each workflow action is translated into HTTP calls to ros_http_bridge2,

which in turn triggers ROS actions/services and the C++ coordinator to move the robots in simulation.

- In real mode:

* RobotManager forwards CallBlock actions to PLCClient, which communicates with the PLC over ROS actions/services,

while other DSL actions can be gradually implemented as real PLC function blocks.

- The combination of unit_lookup, tray_step_poses, and InternalScrewUnitHole logic means:

* You reason about "unit MTQ12, hole 5" at the DSL level.

* RobotManager automatically finds the tray, the pose_index, ensures the correct tray step is aligned,

logs the pose, and sends the correct tray_index + ScrewNum to the HTTP/ROS layer.

With this document, you should be able to:

- Explain to others the responsibility of each class and method.

- Navigate RobotManager's code with confidence.

- Understand how DSL changes propagate to robot actions.

- Extend or debug behaviors (e.g., new actions, new PLC blocks) without getting lost.