

RTSF: An open, schema-validated robot task specification format for vendor-independent description, LLM generation, and multi-platform validation of manipulator tasks

Luigi Gargioni

University of Milano-Bicocca, Viale dell'Innovazione 10, Building U9, Milan, 20126, Italy

ARTICLE INFO

Dataset link: https://github.com/luigigargioni/robot_task_specification_format

Keywords:

Robot task specification
JSON schema
LLM code generation
Digital twin
End-user development

ABSTRACT

Objective: Robot task programming in manufacturing remains tightly coupled to vendor-specific languages, hindering portability and reuse. This paper presents a vendor-neutral, schema-validated format for robot task specification that enables large language model (LLM) generation and multi-robot execution without modifying the task description.

Methods: The Robot Task Specification Format (RTSF) is a JSON format with nine step types covering all six control-flow constructs, governed by a JSON Schema (draft-07). A gap analysis against ten existing formats positions RTSF. Schema-guided LLM generation is evaluated on 30 task descriptions across four domains, with a prompt ablation study. A runtime architecture (recursive parser, PDDL converter, BehaviorTree.CPP converter) enables multi-backend execution. A multi-robot experiment validates six scenarios on four platforms: UR5e, Franka FR3, Kinova Gen3, and Denso Cobotta (physical robot).

Results: No surveyed format combines a task-level abstraction with a formal JSON Schema, vendor-independent execution, complete control-flow coverage, and schema-guided LLM generation. The multi-robot experiment verifies identical action sequences and arguments across over 500 adapter calls on four platforms using unmodified task files. Schema-guided prompting achieves high construct recall (100% corrected validity); an independent benchmark by three external evaluators confirms generalisability. Prompt ablation confirms that disambiguation and field-name guidance rules contribute roughly equally to recall. Scalability benchmark confirms sub-second processing at 5000 steps.

Conclusion: RTSF fills the gap between natural-language task descriptions and heterogeneous execution backends, enabling vendor-independent robot task programming with schema-guided LLM generation and multi-platform validation. The format, examples, scripts, and test suite (107 tests) are released under the MIT licence.

1. Introduction

Collaborative and industrial robots are increasingly deployed across manufacturing, logistics, laboratory, and healthcare settings to execute complex, conditional, multi-agent tasks. Despite this proliferation, robot task programming remains tightly coupled to vendor-specific languages and environments: a task programmed for a Universal Robots arm cannot be directly transferred to a KUKA or ABB system without substantial rework. Beyond portability, the difficulty of task specification acts as a barrier to adoption, particularly for small and medium enterprises (SMEs) where domain experts must rely on integrators to translate their process knowledge into executable robot programs.

Several complementary approaches have been proposed to address this problem. At the lowest abstraction level, the NIST Canonical Robot Command Language (CRCL) [1] provides a vendor-neutral motion

command interface, but operates on joint/Cartesian primitives with no concept of task structure, objects, loops, or error handling. At the highest level, AI planning languages such as PDDL [2] and its hierarchical extension HDDL [3] offer rigorous logical semantics and automated planning, but their text-based syntax is arcane, lacks execution-level constructs (timeouts, retries, HRI confirmation), and is not formally schema-validated, making automated validation and LLM-based generation error-prone [4]. Reactive execution frameworks such as Behaviour Trees [5,6] and ROS-dependent systems (ROSPlan [7], PlanSys2 [8], SkiROS2 [9]) provide modularity and reactivity but couple the task description to a specific library or middleware, producing no shareable, vendor-neutral file. More recently, LLM-based approaches such as Code as Policies [10] and ProgPrompt [11] directly generate Python code from natural language, achieving remarkable flexibility,

E-mail address: luigi.gargioni@unimib.it.

<https://doi.org/10.1016/j.array.2026.101190>

Received 27 July 2026; Received in revised form 25 August 2026; Accepted 26 August 2026

Available online 2 September 2026

2590-0056/© 2026 The Author(s). Published by Elsevier Inc. This is an open access article under the CC BY license (<http://creativecommons.org/licenses/by/4.0/>).

but the generated code is inseparably tied to the robot API used in the prompt, cannot be formally validated, and introduces security risks when executed without sandboxing.

What is missing is an open format that combines task-level abstraction (named actions, workspace objects) with a formal schema, vendor-independent execution, the full set of execution-relevant control-flow constructs, and the ability to be generated by LLMs through schema-guided prompting. This paper addresses the gap with the Robot Task Specification Format (RTSF), a lightweight, human-readable JSON format that fills the role of a portable interchange layer between natural language task descriptions and heterogeneous execution backends.

From a standards and interfaces perspective, RTSF contributes a formally specified, schema-validated domain-specific language with a well-defined adapter interface that decouples task specification from execution, an LLM-based code generation pipeline, and a multi-platform testing framework, addressing requirements specification, software quality through formal validation, and verification and validation of cyber-physical systems.

RTSF is designed around a single principle: *one file, multiple backends*. A task described in a single `.json` file can be: validated offline against a JSON Schema; executed on any robot via an adapter (i.e., Python, Node.js, etc.); compiled into PDDL for AI planning; converted into BehaviorTree.CPP XML for reactive execution; or bound to a digital twin for pre-deployment testing. The design aims to make RTSF accessible to domain experts who can author and share tasks without robot-specific knowledge using a natural language interface and puzzle-like visual editor [12,13].

This paper makes four contributions:

- C1** A formally specified, vendor-neutral robot task description format (RTSF v1.0) covering nine step types and all six control-flow constructs relevant to manufacturing and service robots, governed by a normative JSON Schema (draft-07).
- C2** Evidence that schema-guided LLM prompting achieves high construct recall on 30 natural-language task descriptions across four domains, with a prompt ablation study decomposing the effect of three prompt rule categories (approximately –14 pp from full to minimal).
- C3** A runtime architecture enabling a single RTSF file to drive mock, simulated, and planned execution without modification to the task description; a scalability benchmark on synthetic tasks from 10 to 5000 steps confirms sub-second end-to-end processing time.
- C4** A multi-robot, multi-platform experiment across six structurally distinct scenarios (pick-inspect-place, CNC machine tending, parcel sorting, parallel cross-dock, pharmacist-supervised drug preparation, dual-arm assembly) running on four robot platforms (UR5e 6-DOF, Franka Emika FR3 7-DOF, Kinova Gen3 7-DOF, Denso Cobotta 6-DOF). The experiment verifies that the parser-to-adapter interface correctly isolates task logic from execution backend, achieving 100% action-sequence and argument match across 548 adapter calls using the same unmodified task files, and characterises cross-platform timing across software stub, physics-based Gazebo simulation, controller-stub ROS 2 execution, and physical robot execution.

These contributions test three hypotheses, each derived from a limitation identified above and each falsifiable by one of the experiments reported below.

H1 (portability). If vendor-specific knowledge is confined to an adapter layer behind a fixed method interface, a single unmodified task specification will produce equivalent action sequences across robots of different manufacturers, kinematics, and protocols. The hypothesis follows from the API-coupling of LLM-generated policies [10,11] and the framework-coupling of BT.CPP leaf nodes and PlanSys2 plans [5,8]. It predicts that divergence across platforms is confined to timing,

and is falsified by any scenario-platform pair whose call sequence or arguments differ, or that requires editing the task file (Section 6).

H2 (schema-guided generability). If the target representation is constrained by a formal schema supplied to the model, LLM generation will yield specifications whose structural defects are detectable and repairable before execution. The hypothesis responds to the absence of schema validation in free-text planning notations [2,3] and of pre-execution validation in code-generation pipelines [10,14]. Since the mechanism claimed is detection and repair rather than error-free generation, it predicts high corrected validity alongside imperfect first-pass validity, and a refutable asymmetry: construct completeness is not schema-enforceable and should therefore not improve under the correction loop (Sections 5.2 and 5.2.1).

H3 (locus of generation difficulty). If the schema carries the structural constraints, the residual generation difficulty lies in the prompt rather than in the format; the alternative, that the step-type taxonomy is itself hard to target, would undermine the design. It predicts that removing prompt rule categories degrades construct recall while leaving schema conformance largely intact (Section 5.3), and Sections 5.2.1 and 5.4 report the evidence on the residual confound, that the prompt rules were authored by the format designer.

The remainder of the paper is organised as follows. Section 2 surveys related work and positions RTSF in the landscape of existing formats. Sections 3 and 4 present the format design and system architecture, respectively. Section 5 reports the LLM evaluation and prompt ablation study, while Section 6 presents the multi-robot, multi-platform validation. Section 7 discusses implications, limitations, and timing constraints, and Section 8 concludes.

2. Related work

2.1. Task and motion planning languages

The Planning Domain Definition Language (PDDL) [2,15] is the standard for AI task planning competitions. A PDDL domain defines types, predicates, and action schemas with first-order logic preconditions and effects; a problem file specifies the initial state and goal. Planners such as Fast Downward [16] and POPF [17] search for an action sequence achieving the goal. HDDL [3] extends PDDL with Hierarchical Task Networks (HTN), enabling recursive task decomposition through ordered method definitions. Both languages offer rigorous semantics and decades of solver support [4], but they are purely declarative: execution-level constructs (timeouts, retries, HRI confirmation, parallel multi-robot branches, event triggers) are absent. The free-text syntax is not schema-validated, so LLM-generated PDDL is frequently syntactically invalid, and generated plans are flat action sequences that discard the original task structure.

2.2. Reactive execution frameworks

Behaviour Trees (BTs) [5] originated in game AI and have become widely adopted in industrial and research robotics [6], including production navigation systems such as Nav2 [18]. BT.CPP (C++) and `py_trees` (Python) provide control-flow nodes (Sequence, Fallback, Parallel, Decorator) whose reactive ticking enables continuous monitoring and recovery. BTs naturally model error handling (Fallback/Retry) and parallel execution, and RTSF provides a converter to BT.CPP XML. The format, though, is library-specific and not interoperable across implementations; it does not capture high-level task semantics (workspace objects, agent roles, safety envelopes), and its leaf nodes require C++ implementations, making standalone task authoring impossible without a programming environment.

SkiROS2 [9] extends BTs with a Web Ontology Language (OWL) world model and automatic PDDL domain generation, providing rich knowledge integration and reactive multi-robot orchestration. Its reliance on the full ROS 2 stack and Python skill classes, though, makes it framework-dependent and inaccessible to non-programmers, and no standalone portable task file is produced.

2.3. ROS-based planning systems

ROSPlan [7] (ROS 1) and its successor PlanSys2 [8] (ROS 2) integrate PDDL-based planning into the Robot Operating System (ROS), enabling automated plan generation and BT-based execution with re-planning on failure. Both inherit all PDDL limitations for task description and require a full ROS installation, making them inaccessible outside the ROS ecosystem. The ROS 2 Industrial initiative extends ROS 2 [19] with vendor-specific driver packages and standardised interfaces, but does not address task-level portability or vendor-neutral task description.

2.4. Standardised robot interfaces and industrial standards

The NIST Canonical Robot Command Language (CRCL) [1] provides a vendor-neutral motion command protocol (XML-based) with an XSD schema. CRCL decouples task-level software from vendor controllers at the motion level, and has been implemented for FANUC, ABB, UR, and KUKA. Its scope, though, is strictly the motion abstraction level: the CRCL schema has no concept of objects, locations, conditions, loops, or error handling. CRCL is thus an appropriate *execution target* for RTSF adapters, not a competing task-level format.

ISA-88 (IEC 61512) [20] defines a hierarchical procedural model (Procedure → Unit Procedure → Operation → Phase) for batch process control in the pharmaceutical and chemical industries. RTSF's hierarchy (task → subtask → action step) mirrors this model, adapting it for robot manipulators in discrete manufacturing. RTSF extends ISA-88 by adding parallelism, multi-robot synchronisation, event-driven triggers, and a formal JSON Schema, constructs absent from ISA-88 due to its process-industry focus.

2.5. LLM-based task generation

Code as Policies [10] demonstrates that large language models can generate executable Python robot code from natural language by providing the robot API as in-context examples. ProgPrompt [11] adds assertion statements for precondition checking and recovery. Inner Monologue [14] uses LLM-generated language feedback to close the perception-action loop. SayCan [21] grounds LLM-generated plans in robotic affordances by scoring actions against learned world models. RT-2 [22] co-trains vision-language models on robotic trajectory data to produce vision-language-action models that directly output motor commands. VoxPoser [23] uses LLMs to generate 3D value maps for manipulation, bridging language and spatial reasoning. A shared limitation across these approaches is that the generated code is inseparably tied to the robot API or model used in the prompt, cannot be formally validated without execution, offers no schema-enforced safety guarantees, and requires re-prompting or retraining for each new robot platform. RTSF instead constrains generation with a formal schema, sacrificing some flexibility in exchange for portability and validation guarantees.

The reported results of these systems clarify what has been established and why the present evaluation measures something different. SayCan [21] is the most systematically quantified: over 101 instructions to a mobile manipulator in an office kitchen, PaLM-SayCan selects the correct skill sequence in 84% of cases and executes it in 74%, falling to 81% and 60% in a previously unseen kitchen. Code as Policies [10] reports a 39.8% solve rate on HumanEval alongside demonstrations on several real platforms, but reports its robotic evaluation per task family rather than as an aggregate. ProgPrompt [11] evaluates on VirtualHome with GPT-3 using success rate, goal-condition recall, and plan executability, the last measuring the fraction of generated actions the environment can execute and the closest published analogue to the validity metric used here. Inner Monologue [14] and VoxPoser [23] report task-level success on simulated and real manipulation suites, and RT-2 [22] gains on emergent-skill evaluations over its RT-1 baseline. All of

these measure task execution success within a fixed embodiment, which is appropriate for a policy but unmeasurable for a format prescribing no motion primitives; what transfers instead is executability, which ProgPrompt introduces precisely because generated code frequently is not executable and which RTSF makes statically decidable. SayCan's asymmetric cross-environment drop points the same way: symbolic descriptions degrade far less across contexts than their physical realisation, which is why this evaluation is organised around structural equivalence across platforms rather than end-to-end success on one.

Two of these claims have since been qualified, in both cases in a direction that supports the present argument. The first concerns the sufficiency of code generation alone: a neuro-symbolic reformulation of Code-as-Policies inserting symbolic verification and interactive validation into the generation loop reports a 46.2% relative improvement in task success over the Code-as-Policies baseline and over 86.8% executability in dynamic, partially observable settings [24]. That LLMs can generate executable robot code therefore stands; the implicit corollary that generation alone suffices does not. The second concerns pre-execution validation, largely absent in 2022–2023 and now an explicit target: linear-temporal-logic verification of plans before execution [25] and hybrid pipelines routing LLM proposals through a symbolic planner and validating the generated code by compilation and simulation, at 71%–75% task completion [26], both start from the premise that generated plans should be checked before a robot acts; recent reviews of LLM-based task planning [27] and of vision-language-action models [28] confirm the trajectory, the latter identifying cross-embodiment transfer as open. What differs is where the check sits: these systems verify an instance, whereas RTSF constrains the representation so a class of defects cannot be expressed at all, which is cheaper and decidable but strictly weaker, since a schema cannot detect that a conformant plan is the wrong plan. Combining the two is a concrete direction for future work.

2.6. Comparison

Table 1 summarises eleven formats, RTSF and the ten existing formats surveyed above, across seven dimensions relevant to robot task specification. RTSF is the only format that satisfies all seven: it operates at the task level, carries a formal JSON Schema, is vendor-neutral, covers all six control-flow constructs, supports schema-guided LLM generation, and ships with converters to both PDDL and BT XML under an MIT licence.

3. RTSF format design

3.1. Design goals

RTSF v1.0 was designed around five goals that jointly address the gap identified in Section 2.

The format operates at the *task abstraction level*: it names actions (pick, place, inspect) and workspace objects without encoding joint angles or Cartesian poses, which is the level at which domain experts think about tasks and at which LLMs produce the most accurate output from natural language. The notation is JSON with a formal JSON Schema (draft-07), which provides universal parser support, direct in-memory representation across all major programming languages, and a machine-parseable specification enabling automated validation and schema-guided LLM generation. YAML was rejected for implicit type coercions, XML for verbosity, and custom DSLs for LLM generation difficulty.

No field in RTSF encodes robot-brand-specific primitives: execution is mediated through a Python adapter interface, so the same task file runs on any robot with a conforming adapter. The format also provides *full control flow*: real manufacturing tasks require loops (batch repetition), conditionals (quality gates), parallel branches (dual-arm, robot+human), error handling (retry on grasp failure), human-in-the-loop confirmation (per ISO/TS 15066 [29]), event triggers (WMS

Table 1

Comparison of robot task specification formats across seven dimensions. L = loop, C = conditional, P = parallel, E = error handler, H = human-in-loop, M = multi-robot sync. ✓ = full support; ~ = partial; × = not supported.

Format	Abstraction level	Notation	Formal schema	Vendor-neutral	L	C	P	E	H	M	LLM-gen.	Licence
PDDL 2.1 [2]	Goal	Custom text	×	✓	×	×	×	×	×	×	×	Open (ICAPS)
HDDL [3]	Goal + Task	Custom text	×	✓	~	×	×	×	×	×	×	Open (ICAPS)
BT.CPP [5]	Task	XML + C++	Partial	✓	✓	✓	✓	✓	×	×	×	MIT
CRCL [1]	Motion	XML	XSD	✓	×	×	×	×	×	×	×	Public domain
ROSPlan [7]	Goal + Task	PDDL + YAML	×	×	(ROS 1)	×	×	×	~	×	×	BSD-3
PlanSys2 [8]	Goal + Task	PDDL + ROS 2	×	×	(ROS 2)	✓	✓	✓	✓	×	~	Apache-2.0
SkiROS2 [9]	Task + Goal	Python + OWL	OWL ontology	×	(ROS 2)	✓	✓	✓	✓	×	✓	BSD-3
ISA-88 [20]	Task (process)	Vendor XML	Vendor XSD	×	~	~	×	~	✓	×	×	IEC (prop.)
Code-as-Policy [10]	Task	Python	×	×	✓	✓	✓	✓	✓	✓	✓	N/A
ProgPrompt [11]	Task	Python	×	×	✓	✓	✓	✓	✓	✓	✓	N/A
RTSF v1.0	Task	JSON	JSON Schema draft-07	✓	✓	✓	✓	✓	✓	✓	✓	MIT

signals), and multi-robot synchronisation, all as first-class schema constructs, not workarounds. Finally, the JSON Schema is embedded in the system prompt with a self-correction loop that feeds validation errors back to the LLM, transforming generation from unconstrained text synthesis into schema-constrained structured completion.

3.2. Document structure

Every RTSF file is a JSON object with two top-level fields: `rtsf_version` and `task`. The task object groups seven orthogonal sections:

- **metadata**: author, creation date, robot model, version, and licence.
- **safety**: top-level machine-readable safety envelope: `max_speed_ms`, `max_force_n`, `max_payload_kg`, `human_proximity_zone_m`, `on_proximity_detected`, and `emergency_stop`. Placing safety constraints at the top level lets the parser enforce them before executing the first step, and lets a digital twin extract safety envelopes without parsing the full program logic.
- **workspace**: declarative description of objects (with type and `initial_location`), locations (typed as `home`, `source`, `destination`, `processing`, `storage`, `discard`, or `generic`), and agents (typed as `cobot`, `industrial_robot`, `mobile_robot`, `human`, or `virtual`).
- **actions_library**: STRIPS-style action schemas with typed parameters, preconditions, and effects, enabling automatic PDDL domain generation.
- **subtasks**: named, parameterised reusable sub-procedures with explicit parameter types (`object_ref`, `location_ref`, `string`, `integer`, `float`, `boolean`).
- **program**: the imperative step sequence (see Section 3.3).
- **termination**: optional success/failure condition expressions.

The separation of `workspace`, `actions_library`, and `program` mirrors the classical AI planning distinction between domain (types, predicates, action schemas) and problem (objects, initial state, goal), enabling a direct PDDL mapping while preserving the imperative execution order absent from purely declarative formats.

3.3. Step types

RTSF v1.0 defines nine step types covering all six control-flow classes. Table 2 summarises them; Listing 1 shows a representative nested example.

The action step is the atomic execution unit: it names a capability from `actions_library`, an agent, typed arguments, and an optional `timeout_ms/on_timeout` policy (abort or retry). The loop step covers three sub-types: `count` (fixed repetition), `while` (condition-gated), and `for_each` (over a declared collection). The

conditional step branches on four condition types: `sensor_check`, `state_check`, `counter_check`, and `message_check`. The parallel step runs branches with `wait_all`, `wait_any`, or `fire_and_forget` synchronisation.

The `human_action` step models human-in-the-loop actions as a first-class construct, reflecting the requirement in collaborative robot deployments under ISO/TS 15066 [29] that operator confirmation gates be explicitly modelled. A dedicated step type enables static schema validation that instruction and confirmation required fields are always present, supports end-user authoring of task programs, and allows the BT converter to generate appropriate acknowledgement nodes. Similarly, `multi_robot_sync` models barrier synchronisation between distinct agents, complementing the parallel step which governs concurrent branches of a single robot; Listing 2 shows a barrier example from a dual-robot assembly task.

Subtask parameters use a `$parameter_name` variable reference syntax (inspired by shell scripting and Ansible) that is unambiguous in JSON, parseable with a single regular expression, and readily generated by LLMs. The scope of `$variables` is limited to subtask bodies to keep the format declarative and side-effect-free at the top level.

3.4. Validation

RTSF validation is two-level. *Structural validation* checks each task against the normative JSON Schema (Appendix) in `schema/rtsf_schema_v1.0.json` using `jsonschema` (draft-07). The schema uses `additionalProperties: false` on all objects, so LLM-invented field names are rejected immediately rather than silently ignored. *Semantic validation*, implemented in `rtsf/validator.py`, performs three classes of cross-reference checks: subtask IDs referenced in `subtask_call` steps must be declared in subtasks; agent IDs referenced in `multi_robot_sync` must appear in `workspace.agents`; agent IDs referenced in parallel branches must be declared agents. The complete test suite comprises 107 tests (21 adversarial invalid-schema cases, 16 Gazebo adapter dry-run cases, and 70 positive cases), all passing deterministically.

4. System architecture

The RTSF system architecture implements the *one file, multiple backends* principle through four components: a recursive parser, two format converters, and a robot adapter interface. Fig. 1 provides an overview of the full pipeline from natural-language input to robot execution.

4.1. Parser and execution model

RTSFParser (in `rtsf/parser.py`) loads a validated RTSF JSON file and recursively traverses `program.steps`. Each step is dispatched to the appropriate handler: action steps invoke the corresponding adapter method; loop, conditional, parallel, error_handler, and subtask_call steps recursively process nested

Table 2

RTSF v1.0 step types, key fields, control-flow class, and representative use.

Step type	Key fields	Class	Example use
action	action, agent, args, timeout_ms, on_timeout	Primitive	Pick part from tray
loop	loop_type (count/while/for_each), body	Control	Repeat batch of n items
conditional	condition (sensor_check/state_check...), then, else	Control	Route on vision quality check
parallel	branches, wait_mode (wait_all/wait_any/fire_and_forget)	Control	Robot moves; human confirms
subtask_call	subtask (ID ref), args (\$param bindings)	Modularity	Invoke pick-inspect-place
human_action	instruction, confirmation_required, agent	HRI	Pharmacist approves vial label
event_trigger	event, source, timeout_ms, on_trigger	Reactive	Wait for WMS pick-order signal
error_handler	body, on_error, max_retries	Fault tol.	Retry grasp; recover to home
multi_robot_sync	agents (ID list), sync_type (barrier/pipeline)	Multi-robot	Synchronise UR5e and AGV handoff

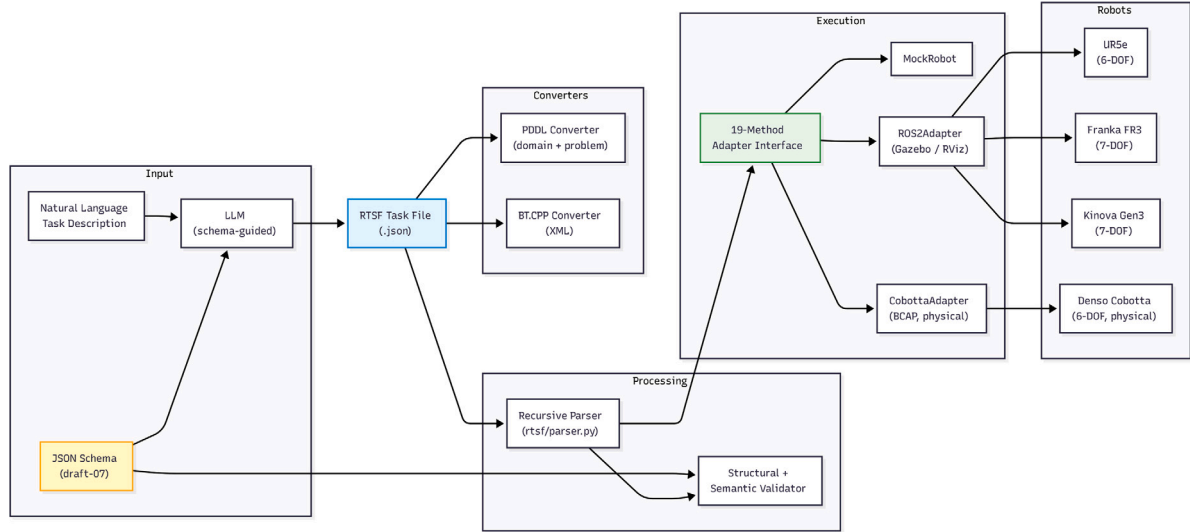


Fig. 1. RTSF system architecture. Natural-language task descriptions are converted to RTSF JSON via schema-guided LLM prompting. The recursive parser traverses the task and dispatches adapter calls; the same RTSF file can also be converted to PDDL or BehaviorTree.CPP XML. The 19-method adapter interface decouples task logic from execution backends (Mock, ROS 2, BCAP), enabling vendor-independent execution across robot platforms.

step lists. Subtask calls construct a variable binding context ($\$name \rightarrow value$) that is threaded into the subtask body. Safety constraints from the `safety` section (`max_speed_ms`, `max_force_n`, `max_payload_kg`) are applied before the first step via adapter calls (`set_max_speed`, `set_max_force`, `set_max_payload`). No robot-specific code appears in the parser: all hardware interactions are mediated exclusively through the adapter interface.

4.2. PDDL converter

RTSFToPDDL (in `rtsf/to_pddl.py`) generates a PDDL domain file and a PDDL problem file from an RTSF task. `workspace`, `objects` and `workspace.locations` become PDDL `:objects`; `actions_library` entries become PDDL `:action` schemas, with preconditions and effects in first-order logic; the program step sequence initialises the PDDL `:init` state. RTSF can thus serve as a task authoring interface above PDDL-based planning systems such as PlanSys2 [8], so that teams already using PDDL planners can adopt RTSF without discarding their planning infrastructure.

4.3. BehaviorTree.CPP converter

RTSFToBT (in `rtsf/to_bt.py`) transpiles RTSF steps to BehaviorTree.CPP v3 XML. The structural mapping is well-defined: `loop` (count) maps to a `WhileDoElse` node with counter decorator; `conditional` maps to `Sequence/Fallback`; `parallel` maps to a `Parallel` node; `error_handler` maps to `Fallback` with `RetryUntilSuccessful` decorator; `subtask_call` maps to a `SubTree` reference; `human_action` maps to a custom `HumanConfirmation` action leaf. The converter is intentionally one-way (RTSF

$\rightarrow$ BT XML): BT XML lacks the information (safety, workspace, HRI field semantics) required to reconstruct a complete RTSF task.

4.4. Robot adapter interface

The adapter interface is the single extension point of the architecture. Any Python class implementing 19 methods (`set_max_speed`, `set_max_force`, `set_max_payload`, `move`, `pick`, `place`, `rotate`, `wait`, `inspect`, `process`, `scan`, `open_gripper`, `close_gripper`, `send_message`, `receive_message`, `read_sensor`, `get_state`, `wait_for_signal`, `notify_operator`) can be passed to `RTSFParser` as the execution backend. Three adapter implementations are provided: `MockRobot` (no-op, for instantaneous dry-run testing), `RobotStub` (returns deterministic sensor values), and `ROS2Adapter` (executes `FollowJointTrajectory` actions on any ROS 2-controlled robot in Gazebo Harmonic, with the robot name, controller action server path, and named-location joint configurations read from a per-robot YAML file; validated on Universal Robots UR5e, Franka Emika FR3, and Kinova Gen3). Decoupling the task `.json` file from the execution backend is an architectural guarantee: deploying to a new robot requires only a YAML configuration file, and all existing RTSF tasks run unchanged. Listing 3 shows the adapter contract.

4.5. Scalability

The four main processing operations, JSON parsing, schema validation, semantic validation, and format conversion (PDDL and BT XML), were benchmarked on six synthetic tasks ranging from 10 to 5000 steps, generated with the same step-type distribution as real examples. Each

Listing 1: Excerpt from `ur5e_pick_place.json`: main program nesting loop, `error_handler`, and `subtask_call`, with recovery to home on failure.

```

1 "program": { "steps": [
2   { "id": "s0_home", "type": "action",
3     "action": "move", "agent": "ur5e_1",
4     "args": { "to_location": "home" } },
5   { "id": "s1_batch", "type": "loop",
6     "loop_type": "count", "count": 3,
7     "body": [
8       { "id": "s1_guard", "type": "error_handler",
9         "body": [
10          { "id": "s1_call", "type": "subtask_call",
11            "subtask": "subtask_pick_inspect_place"
12          },
13          { "args": { "part": "part_a",
14                    "tray_loc": "source_tray" } }
15        ],
16        "on_error": [
17          { "id": "s1_home", "type": "action",
18            "action": "move", "agent": "ur5e_1",
19            "args": { "to_location": "home" } }
20        ],
21        "max_retries": 1 }
22    ],
23    { "id": "s2_end", "type": "action",
24      "action": "move", "agent": "ur5e_1",
25      "args": { "to_location": "home" } }
26  ] }

```

Table 3

RTSF processing time (ms, mean of 10 runs) by task size. PDDL generation time depends on `actions_library` size (constant across tasks), not step count.

Steps	JSON parse	Schema val.	Semantic val.	BT gen.	PDDL gen.
10	0.35	1.49	0.01	0.03	0.09
50	0.43	4.19	0.05	0.09	0.08
100	1.41	7.82	0.11	0.18	0.09
500	1.98	30.49	0.47	0.67	0.09
1000	2.40	54.29	0.86	1.31	0.09
5000	7.71	295.36	4.72	7.22	0.10

operation was timed over 10 independent iterations; Table 3 reports the means.

Schema validation dominates at large scales, growing as $O(n^{0.85})$ ($R^2 = 0.993$), consistent with the `jsonschema` validator performing per-step recursive checks. JSON parsing scales as $O(n^{0.50})$ ($R^2 = 0.931$), as expected for file I/O dominated operations. Semantic validation and BT generation grow linearly. PDDL generation time is effectively constant (≈ 0.09 ms across all sizes) because it depends on `actions_library` size, not step count. Even at 5000 steps, more than two orders of magnitude larger than any real task in the evaluation, total processing time (JSON parse + schema + semantic + BT) is under 316 ms on a standard laptop, well below any practical threshold for manufacturing task files. Fig. 2 visualises the log-log scaling of all four operations with fitted power-law exponents.

Three consequences follow. The cost profile is dominated by one operation: schema validation is 93.8% of pipeline time at 5000 steps (295.36 of 315.01 ms) and already 79.3% at 10 steps, so optimisation

Listing 2: Excerpt from `multi_robot_assembly.json`: a `multi_robot_sync` step with barrier synchronisation. Two agents (`robot_1`, `robot_2`) execute independent sub-assembly steps concurrently; the parser dispatches per-agent steps via threads and joins them at the barrier before continuing.

```

1 { "id": "s1_dual_prep_sync",
2   "type": "multi_robot_sync",
3   "agents": ["robot_1", "robot_2"],
4   "sync_type": "barrier",
5   "steps_per_agent": {
6     "robot_1": [
7       { "id": "r1_move_station_a", "type": "action",
8         "action": "move", "agent": "robot_1",
9         "args": { "to_location": "station_a" } },
10      { "id": "r1_weld_a", "type": "action",
11        "action": "process", "agent": "robot_1",
12        "args": { "object": "sub_assy_a",
13                  "operation": "weld" } },
14      { "id": "r1_pick_a", "type": "action",
15        "action": "pick", "agent": "robot_1",
16        "args": { "object": "sub_assy_a",
17                  "from_location": "station_a" } }
18    ],
19    "robot_2": [
20      { "id": "r2_move_station_b", "type": "action",
21        "action": "move", "agent": "robot_2",
22        "args": { "to_location": "station_b" } },
23      { "id": "r2_rivet_b", "type": "action",
24        "action": "process", "agent": "robot_2",
25        "args": { "object": "sub_assy_b",
26                  "operation": "rivet" } },
27      { "id": "r2_pick_b", "type": "action",
28        "action": "pick", "agent": "robot_2",
29        "args": { "object": "sub_assy_b",
30                  "from_location": "station_b" } }
31    ]
32  }
33 }

```

has a single target. Its sub-linear exponent reflects amortisation rather than the validator's per-step work, the marginal cost per step falling from 149 μ s at 10 steps to 59 μ s at 5000 as the fixed cost of compiling the schema is spread further; validation is thus relatively most expensive on the smallest files, where the absolute cost is negligible. And the constant PDDL time locates the planning frontend's cost in the `actions_library` rather than the task body, so a task file can grow arbitrarily without affecting conversion, though a large action vocabulary would shift this term. At the sizes real tasks reach the whole pipeline is therefore comfortably inside an interactive budget, 9.5 ms at 100 steps, so validation can run on every edit in an authoring tool rather than as a batch step, which is what a schema-guided authoring

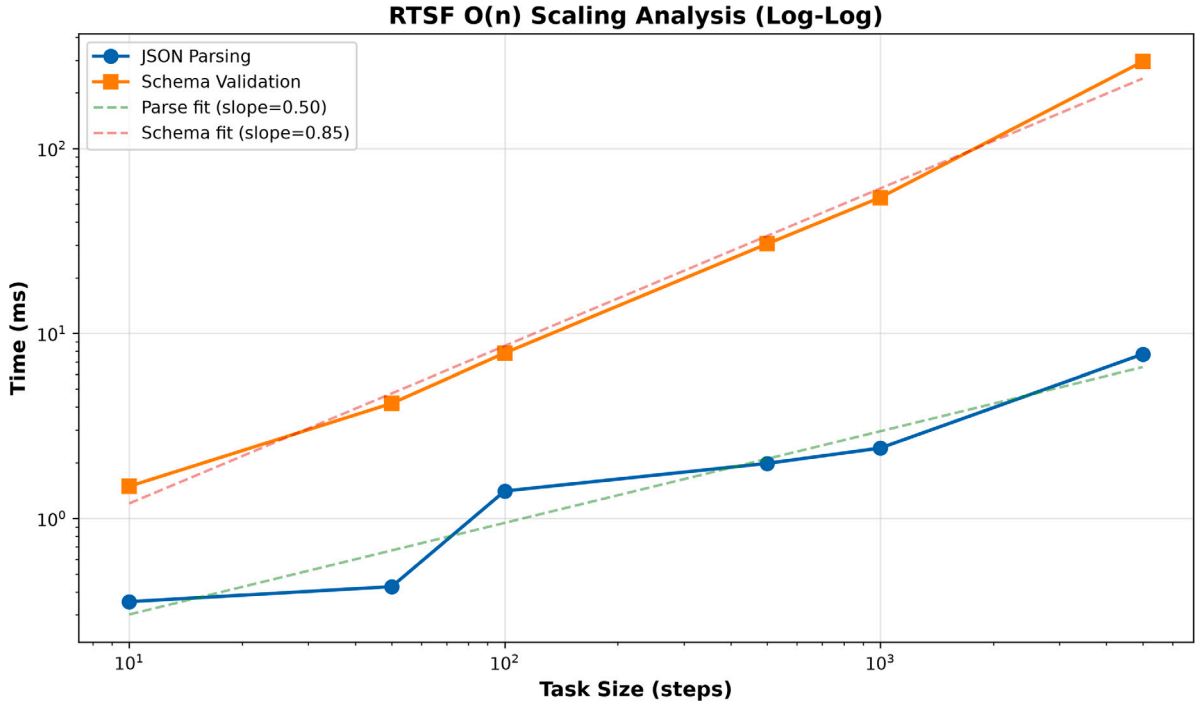


Fig. 2. Log-log scaling of RTSF processing operations versus step count. Power-law fits confirm $O(n^{0.85})$ for schema validation ($R^2 = 0.993$) and $O(n^{0.50})$ for JSON parsing ($R^2 = 0.931$). Semantic validation and BT generation grow linearly; PDDL generation is constant.

loop depends on. At 5000 steps, two orders of magnitude above the largest task in this evaluation, the 315 ms total no longer fits a per-keystroke budget but remains well within the delay tolerated for an explicit validate-and-save action.

5. LLM evaluation

5.1. Experimental setup

The LLM evaluation measures whether schema-guided prompting can reliably generate valid, construct-complete RTSF tasks from natural language descriptions. The experiment uses two models: `gpt-5.4-mini` (OpenAI) and `meta-llama/llama-4-scout-17b-16e-instruct` (Ollama), both at temperature 0.2 with a maximum of 3 self-correction retries (`max_retries=3`). The task set comprises 30 manually authored natural language descriptions across four domains: manufacturing (10 tasks), laboratory (7), logistics (7), and healthcare (6). Descriptions range from 30 to 120 words and cover scenarios such as batch pick-and-place, dual-robot assembly, order picking, and pharmaceutical preparation. Each description is annotated with an `expected_constructs` list (the step types that a complete RTSF encoding of the task should include), manually assigned by the author; an independent second annotator independently annotated all 30 descriptions using the annotation guide released with the repository, achieving Cohen's $\kappa = 0.92$ (95% CI [0.88, 0.95]), confirming annotation reliability (full statistics in the Threats to Validity section). The second annotator also assessed semantic accuracy of the generated RTSF tasks (see Section 5.2).

The system prompt contains 13 rules in three categories: (i) JSON conformance and schema-compliance rules (rules 1–6); (ii) field-name and step-type guidance rules (rules 7–10); and (iii) disambiguation rules that resolve common step-type ambiguities (rules 11–13, e.g., “use `error_handler`, not a loop, whenever the task mentions retry on failure”). The full schema (`rtsf_schema_v1.0.json`) is included verbatim in the prompt.

Two metrics are computed per task: *construct recall* = $|\text{expected} \cap \text{generated}| / |\text{expected}|$, which measures whether all

expected step types appear in the generated output; and *validity* = whether the generated JSON passes both structural and semantic validation. A task is corrected by feeding the validation error message back to the model and requesting a revised JSON. A third metric, *semantic accuracy*, was assessed by the independent second annotator on all 30 generated tasks: for each present construct, the annotator verified that field values, argument bindings, and nesting structure were semantically correct relative to the natural-language description. Semantic accuracy is reported as the percentage of present constructs that pass this check.

5.2. Main evaluation results

Table 4 reports construct recall and validity per domain. All 30 tasks produced valid RTSF after self-correction (100% corrected validity). The average construct recall across all tasks is nearly 98%, with manufacturing at 100% and healthcare the most challenging due to tasks requiring complex combinations of `multi_robot_sync`, `human_action`, and `error_handler`. First-pass validity, without self-correction, was 80% (24/30 tasks); the remaining 6 tasks required at most 2 retries to fix schema errors.

The independent second annotator also assessed semantic accuracy: for each present construct, the annotator verified that field values, argument bindings, and nesting structure were semantically correct relative to the natural-language description. The average across all 30 tasks is 91%, with manufacturing highest and logistics lowest (Table 4). The gap between construct recall and semantic accuracy reflects cases where a step type is present but its field values or nesting do not fully match the task description, for example, a `conditional` with correct `then/else` branches but incorrect sensor check arguments, or an `error_handler` with a recovery body that does not match the described recovery procedure.

Validity and recall move in opposite directions across domains: manufacturing has the lowest first-pass validity (6/10) with perfect recall (100.0%), healthcare perfect first-pass validity (6/6) with the second-lowest recall (96.7%). The two therefore capture independent failure modes. Schema errors are surface-level conformance failures,

Listing 3: The robot adapter contract (excerpt from `rtsf/-mock_robot.py`). Any class implementing these methods can be passed to `RTSFParse` as the execution backend. `MockRobot`, `ROS2Adapter`, and `CobottaAdapter` are three such implementations.

```

1 class RobotStub:
2     """No-op robot. Every method accepts any
3       arguments and
4       returns None (or a safe default for
5       queries)."""
6
7     # Safety limits
8     def set_max_speed(self, value): pass
9     def set_max_force(self, value): pass
10    def set_max_payload(self, value): pass
11
12    # Motion and manipulation
13    def move(self, to_location, **kw): pass
14    def pick(self, object, from_location, **kw): pass
15    def place(self, object, to_location, **kw): pass
16    def rotate(self, object, degrees, **kw): pass
17    def open_gripper(self, **kw): pass
18    def close_gripper(self, **kw): pass
19
20    # Perception and processing
21    def inspect(self, object, sensor, **kw): pass
22    def scan(self, object, sensor, **kw): pass
23    def process(self, object, operation, **kw): pass
24
25    # Communication and synchronisation
26    def wait(self, duration_ms, **kw): pass
27    def send_message(self, to_agent, message_type, payload=None, **kw): pass
28    def receive_message(self, from_agent, message_type, **kw): pass
29    def wait_for_signal(self, source, signal, timeout_ms=None):
30        return True
31    def notify_operator(self, step_id): pass
32
33    # State queries
34    def read_sensor(self, sensor, object, property): return True
35    def get_state(self, property): return False

```

mechanically detectable and repairable, which is why self-correction resolves all of them within two retries; missing constructs follow from an interpretive decision about the description and are invisible to the validator, since a task lacking its `parallel` step is fully conformant. This is why both metrics are reported, and why the correction loop raises validity to 100% while leaving recall untouched. The 6.8 pp gap between average recall (97.8%) and semantic accuracy (91.0%) preserves the domain ordering, suggesting graded domain difficulty rather than a separate phenomenon, with logistics penalised on both counts by descriptions that leave concurrency and routing conditions implicit.

Table 4

LLM evaluation results per domain (gpt-5.4-mini and llama-4-scout, temperature 0.2, max 3 retries). FP valid = first-pass valid (no self-correction); Avg/Min recall = construct recall; Sem. acc. = semantic accuracy assessed by independent annotator.

Domain	Tasks	Valid	FP valid	Avg recall	Min recall	Sem. acc.
Manufacturing	10	10/10	6/10	100.0%	100.0%	93.0%
Laboratory	7	7/7	6/7	97.1%	80.0%	90.0%
Logistics	7	7/7	6/7	96.4%	75.0%	88.0%
Healthcare	6	6/6	6/6	96.7%	80.0%	92.0%
Total	30	30/30	24/30	97.8%	75.0%	91.0%

Fig. 3 visualises construct coverage per task across all four domains. The minimum recall of 75% (logistics task `log_05`) reflects a task requiring four step types including `multi_robot_sync` and `parallel` for a dual-AGV cross-docking scenario; the model produced a schema-valid but structurally incomplete output on the first attempt (missing `parallel`).

The heatmap makes the failure profile legible in a way the aggregates cannot. Three of the 30 tasks miss exactly one expected construct each (`lab_06`, `log_05`, `hc_05`); none misses two, and none misses action, expected and generated in all 30. Normalising by how often each construct is expected reveals a pattern rather than three isolated accidents: `parallel` is missed in 1 of the 5 tasks expecting it (20.0%), against 1 of 13 for conditional and 1 of 19 for loop, while the remaining five constructs are never missed. The construct with the highest miss rate is also the one on which the two annotators agreed least ($\kappa = 0.64$ against $\kappa \geq 0.86$ for all others, Section 5.4). We read this convergence as evidence that the residual errors track genuine ambiguity in the source material rather than model noise: when a description says two operations happen “at the same time”, whether strict simultaneity is required is often underdetermined, and annotators and model hesitate at the same points. The residual gap is therefore more likely closed by disambiguating the authoring interface, for example by asking the author to confirm concurrency, than by further prompt engineering, whose ceiling Section 5.3 quantifies.

Two caveats qualify these figures, both detailed in Section 5.4: construct recall counts the presence of a step type rather than its correct use, which the semantic accuracy assessment above partially addresses; and the task descriptions, annotation scheme, and prompting rules share an author with the format, so the recall figures are an upper bound under ideal alignment rather than an estimate for external users. The independent benchmark below addresses the second directly.

5.2.1. Independent benchmark validation

An independent benchmark of 30 additional natural-language task descriptions was constructed by three evaluators with different professional backgrounds: a robotics integrator (E1, 10 tasks), a process engineer (E2, 10 tasks), and a software developer with no robotics or RTSF knowledge (E3, 10 tasks). None of the evaluators designed or contributed to the RTSF format. The task descriptions cover the same four domains as the original set (manufacturing 10, laboratory 7, logistics 7, healthcare 6) but use entirely different scenarios, written in each evaluator’s own professional vocabulary and without exposure to RTSF terminology, the schema’s step-type taxonomy, or the prompting rules. Each evaluator independently annotated the `expected_constructs` for their own descriptions using the same annotation guide.

The same generation pipeline (gpt-5.4-mini, temperature 0.2, max_retries=3, identical system prompt and schema) was re-run on this independent set. Table 5 reports the results alongside the original benchmark.

The independent benchmark achieves 92.7% average construct recall, a drop of 5.1 pp from the original set. All 30 tasks still produce valid RTSF (100% corrected validity), indicating that the format and

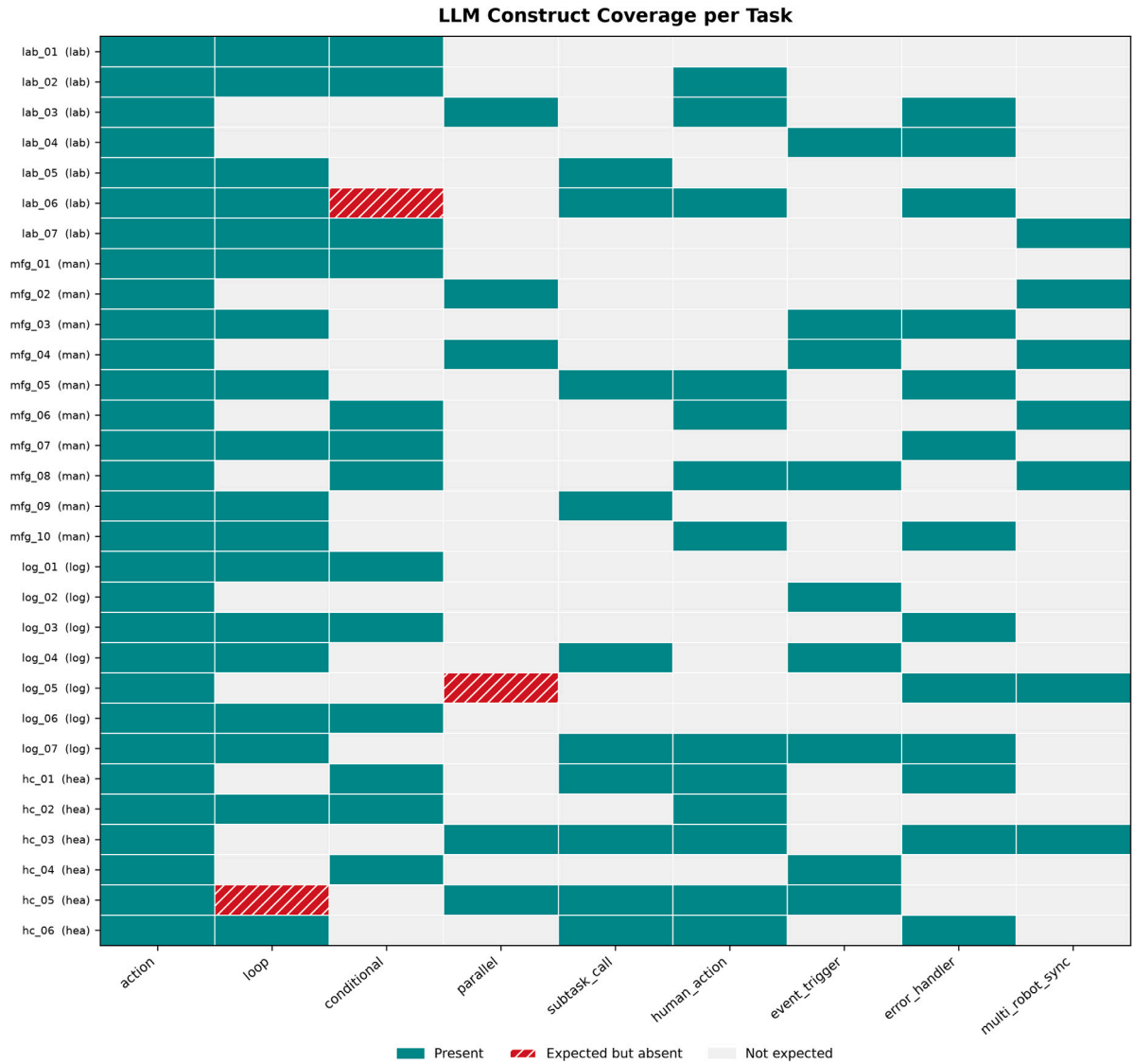


Fig. 3. Construct coverage heatmap per task across four domains. Each cell indicates whether an expected step type was generated by gpt-5.4-mini and llama-4-scout with schema-guided prompting. All 30 tasks achieve $\geq 75\%$ construct recall; the three missing constructs are parallel in log_05, conditional in lab_06, and loop in hc_05. Cells marked “expected but absent” are shown in red with a hatch overlay so that the three states remain distinguishable in greyscale and under colour-vision deficiency.

Table 5

Independent benchmark results per domain (gpt-5.4-mini, temperature 0.2, max 3 retries). FP valid = first-pass valid; Avg/Min recall = construct recall. The independent set was authored and annotated by three evaluators unfamiliar with RTSF.

Domain	Tasks	Valid	FP valid	Avg recall	Min recall
Manufacturing	10	10/10	7/10	91.7%	66.7%
Laboratory	7	7/7	4/7	93.6%	75.0%
Logistics	7	7/7	5/7	97.1%	80.0%
Healthcare	6	6/6	5/6	88.3%	75.0%
Total	30	30/30	21/30	92.7%	66.7%

prompting pipeline generalise beyond the format designer’s mental model. First-pass validity drops from 80% to 70%, indicating that descriptions written without knowledge of RTSF conventions are slightly more likely to produce schema errors on the first attempt. Per-annotator recall is 90.5% (E1, robotics integrator), 95.5% (E2, process engineer), and 92.2% (E3, software developer); the narrow spread (5.0 pp) suggests that the pipeline is robust to the evaluator’s background.

Nine of the 30 independent tasks have at least one missing expected construct (compared to 3 of 30 in the original set). Each of the nine omits exactly one construct, so the nine misses distribute as follows: error_handler in 2 tasks (ind_mfg_04, ind_hc_01), parallel in 2 (ind_hc_02, ind_log_07), multi_robot_sync in 2 (ind_lab_02, ind_lab_05), loop in 2 (ind_mfg_08, ind_hc_06), and human_action in 1 (ind_mfg_05). Notably, the model sometimes substitutes conditional for error_handler when the evaluator describes retry logic in ambiguous terms (e.g., “if the cap doesn’t seat properly” without explicitly framing it as a failure-recovery pattern), and substitutes subtask_call for multi_robot_sync when the evaluator describes concurrent workstations without an explicit barrier. Such substitution patterns align with the disambiguation challenges that prompting rules 11–13 are designed to address, and suggest that evaluators unfamiliar with RTSF’s construct taxonomy describe the same robotic scenarios in terms that map less cleanly to the format’s step types.

Semantic accuracy (Section 5.2) was not reassessed on the independent benchmark, as it requires manual verification of field values and argument bindings against each description by a domain-aware

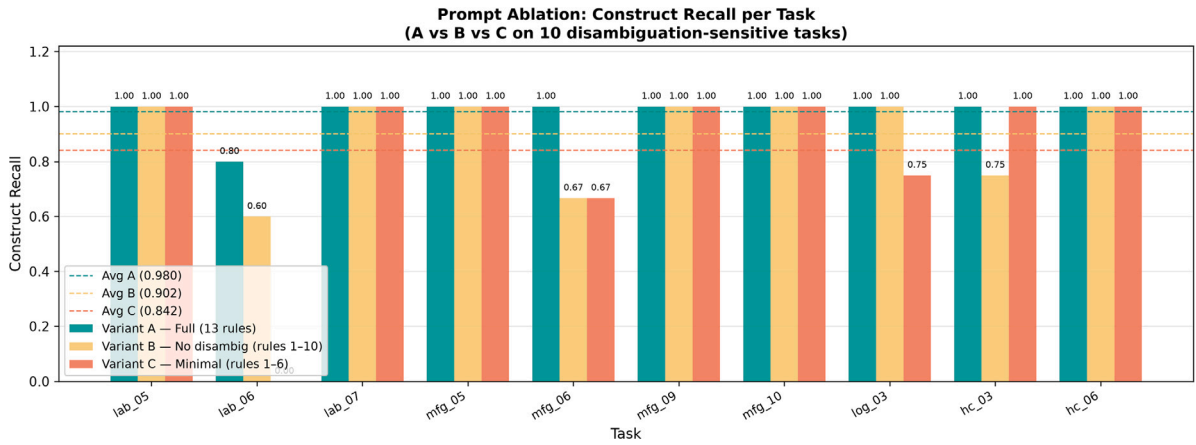


Fig. 4. Per-task construct recall under three prompt variants. Variant A (full, 13 rules) achieves 98.0% average recall; Variant B (no disambiguation, 10 rules) drops to 90.2%; Variant C (minimal, 6 rules) degrades to 84.2%. The A→C degradation of roughly 14 pp is decomposed into −7.8 pp (disambiguation rules) and −6.0 pp (field-name guidance).

Table 6

Prompt ablation on 10 targeted tasks. FP = first-pass valid; Valid = corrected validity.

Variant	Rules	Avg recall	FP valid	Valid
A: Full	13	98.0%	6/10 (60%)	10/10 (100%)
B: No disambig	10	90.2%	8/10 (80%)	10/10 (100%)
C: Minimal	6	84.2%	6/10 (60%)	9/10 (90%)

annotator; this is identified as a direction for future work. The independent benchmark therefore validates construct-level (structural) generalisation but does not yet confirm whether semantic fidelity (91% on the original set) also generalises to external task descriptions.

5.3. Prompt ablation study

A controlled ablation on a 10-task subset, chosen to target disambiguation-sensitive constructs (rules 11–13), isolates the contribution of each prompt rule category. Three prompt variants were evaluated:

- **Variant A (Full):** all 13 rules, including disambiguation rules 11–13.
- **Variant B (No disambig):** rules 1–10 only; disambiguation rules 11–13 removed.
- **Variant C (Minimal):** rules 1–6 only; field-name guidance (7–10) and disambiguation (11–13) removed.

Table 6 and Fig. 4 show a degradation from A to C that is monotonic in aggregate. The A→B drop (−7.8 pp) is attributable to disambiguation rules 11–13: tasks involving `error_handler` (lab_06, hc_03) and `multi_robot_sync` + `human_action` (mfg_06) regressed, because without explicit guidance the model defaults to simpler but less complete constructs that still pass schema validation. The B→C drop (−6.0 pp) is attributable to field-name guidance rules 7–10: without them, task lab_06 (5 expected constructs) fails on all 4 attempts because the model consistently invents non-schema field names (`try_steps`, `recovery_steps`) instead of the correct `body/on_error` fields. This is the only task that is schema-invalid even after self-correction in Variant C, yielding 9/10 validity (90%). Validity is unaffected in Variants A and B because disambiguation rules alter which step type is chosen, not whether the chosen type is schema-conformant. The total A→C degradation of roughly 14 pp decomposes as −7.8 pp (disambiguation) + −6.0 pp (schema field-name guidance).

Fig. 4 shows that this degradation is concentrated rather than distributed, which qualifies how the ablation should be read. Six of

the ten tasks return identical recall under all three variants, and only three degrade from A to C. One dominates: lab_06 falls from 0.80 to 0.00 under Variant C, being the single task whose output stays schema-invalid even after self-correction once field-name guidance is removed, so no construct is credited; it alone accounts for 8.0 of the 13.8 pp degradation, with mfg_06 (−33 pp) and log_03 (−25 pp) making up the rest. One task, hc_03, degrades under Variant B yet recovers under the stricter Variant C, a non-monotonicity best attributed to sampling variation at temperature 0.2 that bounds the resolution of a ten-task ablation. Two conclusions follow. The roughly 14 pp figure is the average effect of a mechanism close to binary in individual cases, in which a rule either prevents a specific structural failure or is irrelevant to the task, rather than a uniform loss of quality. And the largest single contribution comes from field-name guidance preventing a total validity failure, a schema-conformance effect, whereas disambiguation rules redirect the choice among conformant step types: the two categories contribute comparably in magnitude but through different mechanisms. Estimating them with confidence intervals requires a larger ablation set, identified as future work.

5.4. Threats to validity

Several threats to validity apply to the LLM evaluation. The 30-task main evaluation and 10-task ablation are convenience samples across four domains, not a statistically representative benchmark; results may not generalise beyond manufacturing, logistics, laboratory, and healthcare scenarios.

A second concern is annotation bias. The expected `_constructs` field is hand-annotated by the author, and a different annotator might label the same task differently (e.g., whether a retry should be `error_handler` or `loop`). An independent second annotator annotated all 30 descriptions using the released annotation guide, achieving Cohen’s $\kappa = 0.92$ (95% CI [0.88, 0.95], *almost perfect agreement* [30]). The lowest per-construct agreement was for `parallel` ($\kappa = 0.64$), reflecting genuine ambiguity in whether NL descriptions of concurrent actions imply strict simultaneity; all other constructs reached $\kappa \geq 0.86$. A more fundamental circularity bias remains: the schema, task descriptions, annotation scheme, and prompting rules were all designed by the same author. The independent benchmark (Section 5.2.1) partially addresses this, three external evaluators with different professional backgrounds authored 30 new descriptions, on which the same pipeline achieved 92.7% recall with 100% corrected validity. However, the 13-rule system prompt was still authored by the RTSF designer and was not co-designed or reviewed by the three external evaluators. The recall drop therefore reflects both genuine

Table 7

The six RTSF task scenarios used for multi-platform validation. Each row shows which of the nine step types the scenario exercises (✓) and the total number of adapter method calls generated per full execution.

Scenario	act	loop	cond	par	err	hum	evt	sub	sync	Calls
S1 pick-inspect-place	✓	✓	✓		✓			✓		23
S2 machine tending	✓	✓			✓		✓			32
S3 parcel sorting	✓	✓	✓							40
S4 parallel cross-dock	✓			✓						13
S5 pharmacist verify	✓				✓	✓		✓		15
S6 dual-arm assembly	✓								✓	14

format generalisation and residual prompt-rule specificity to the author’s phrasing conventions, which the current experiment cannot fully disentangle.

Construct recall also measures whether a step type *appears* in the output, not whether it is used *correctly*: a semantically vacuous `error_handler` with an empty body would still be counted. The independent second annotator assessed semantic accuracy of all 30 generated tasks (Section 5.2), verifying that each present construct has correct field values and appropriate nesting; the average is 91% (Table 4). This check was not repeated on the independent benchmark, so semantic fidelity under external authorship remains to be confirmed.

Finally, the self-correction mechanism may inflate the corrected validity metric; first-pass validity (80%) is the less gameable indicator. Results also depend on the specific gpt-5.4-mini and llama-4-scout versions evaluated, and LLM weights are updated by providers without notice, making exact replication over time difficult.

6. Multi-platform validation

6.1. Experimental setup

The multi-platform validation experiment tests whether a single RTSF file produces behaviourally equivalent execution across the MockRobot and physics-based (UR5e) or controller-stub (Franka, Kinova) ROS 2 robot execution on multiple robot platforms, and between mock and physical robot execution. Six task scenarios are evaluated on four different robot setups (Universal Robots UR5e (6-DOF), Franka Emika FR3 (7-DOF), Kinova Gen3 (7-DOF), and Denso Cobotta (6-DOF)) using the *identical, unmodified* RTSF task files. Table 7 summarises the RTSF step types exercised by each scenario.

S1: Pick-inspect-place (`pick_place.json`): a batch pick-inspect-place loop with 3 cycles. The subtask performs `move` → `pick` → `inspect` → `conditional-place` (OK bin or NOK bin), exercising `action`, `loop`, `error_handler`, `subtask_call`, and `conditional`. 23 adapter calls.

S2: Machine tending (`machine_tending.json`): the robot tends a CNC machine over 3 blank cycles. Before each cycle, it waits for a machine-ready `event_trigger`; an `error_handler` wraps the pick step (recovery: return to home). After loading the blank, a second `event_trigger` waits for the machining-done signal before unloading. 32 adapter calls.

S3: Parcel sorting (`sorting.json`): five parcels are sorted by weight class using `scan` and nested conditional branches that route each parcel to one of three bins (light/medium/heavy). No subtask is used, providing a contrast in structural pattern. 40 adapter calls.

S4: Parallel cross-dock (`parallel_crossdock.json`): two parallel branches each perform a pick-move-place sequence for different parcels (A and B) with `wait_all` synchronisation. The parser dispatches both branches concurrently via Python threads. 13 adapter calls.

S5: Pharmacist verify (`human_pharmacist_verify.json`): the robot prepares a drug compound (mix, transfer to vial) inside a `subtask_call` wrapped by an `error_handler`, then pauses for pharmacist inspection via a `human_action` step with `confirmation`

required. In automated runs (MockRobot, ROS 2), the step auto-confirms; in interactive runs, the operator presses ENTER. 15 adapter calls.

S6: Dual-arm assembly (`multi_robot_assembly.json`): two robots (`robot_1`, `robot_2`) prepare sub-assemblies independently via `multi_robot_sync` with a barrier synchronisation, then `robot_1` joins the parts at the join station. The parser dispatches per-agent steps via threads and joins them at the barrier. 14 adapter calls.

The environment runs Ubuntu 24.04 (WSL2) with ROS 2 Jazzy for the three digital twin platforms, and Windows for the physical robot. Four robot platforms are evaluated:

- **Universal Robots UR5e (6-DOF)**: `ros-jazzy-ur-simulation-gz` package, Gazebo Harmonic (physics engine: DART); action server `/scaled_joint_trajectory_controller/follow_joint_trajectory`; poses in `ur5e_poses.yaml`.
- **Franka Emika FR3 (7-DOF)**: official `frankarobotics/franka_ros2` stack (branch `jazzy`), RViz with controller stub (`mock_components/GenericSystem`, no physics engine); action server `/joint_trajectory_controller/follow_joint_trajectory` (added via a controller-manager override, not present in the vendor’s default configuration); poses in `panda_poses.yaml`.
- **Kinova Gen3 (7-DOF)**: `ros2_kortex` package (Kinova official), RViz with controller stub; action server `/joint_trajectory_controller/follow_joint_trajectory`; poses in `kinova_gen3_poses.yaml`.
- **Denso Cobotta (6-DOF)**: physical robot with 2-DOF parallel gripper, controlled via proprietary BCAP protocol (Windows-only); `CobottaAdapter` (`real_robot/denso_cobotta/cobotta_adapter.py`) reads joint configurations from `cobotta_poses.yaml`; executed using standalone runner `run_cobotta_task.py` (no ROS 2 dependency).

The three digital twin platforms share a single generic `ROS2Adapter` (`digital_twin/ros2_adapter.py`) that reads the robot name, action server path, joint names, and named-location joint configurations from the per-robot YAML file. No code change is needed to add a new ROS 2 robot; only a YAML file is required. The Denso Cobotta uses a dedicated `CobottaAdapter` implementing the same 19-method adapter interface, showing that RTSF can integrate with non-ROS 2 communication protocols. Each scenario task file is executed eight times (four robots × two adapters): once with MockRobot (instantaneous, for baseline) and once with the robot-specific adapter (`ROS2Adapter` for digital twins, `CobottaAdapter` for the physical robot). For S5, the `human_action` step is auto-confirmed in all backends to ensure deterministic trace comparison. For S6, both agents share the same adapter instance (the parser dispatches per-agent steps via threads). All runs save an execution trace (`JSON` array of timestamped adapter method calls and arguments). `compare_traces.py` computes three metrics: sequence match percentage, argument match percentage, and per-action timing delta Δt (relative timestamp difference between the two runs).

6.2. Results

Table 8 reports the behavioural equivalence metrics for all twenty-four scenario-robot pairs (six scenarios × four robots).

All twenty-four scenario-robot combinations achieve 100% sequence and argument equivalence between MockRobot and the robot-specific adapters. Across 548 total adapter calls (137 per robot), six structurally distinct task programs, and four robot platforms with different DOF counts, joint naming conventions, ROS 2 action server paths, and communication protocols, the parser generates identical

Table 8

Behavioural equivalence metrics: MockRobot vs. robot-specific adapters across all six validation scenarios and four robot platforms. Execution time = max relative timestamp (total execution time); Avg Δt = mean per-action relative timestamp difference. All twenty-four combinations achieve 100% sequence and argument equivalence.

Robot	Scenario	Actions	Seq.	Args	Exec. time	Avg Δt
UR5e	S1 pick-inspect-place	23	100%	100%	34,533 ms	17,311 ms
	S2 machine tending	32	100%	100%	42,713 ms	20,067 ms
	S3 parcel sorting	40	100%	100%	81,815 ms	39,841 ms
	S4 parallel cross-dock	13	100%	100%	43,072 ms	17,545 ms
	S5 pharmacist verify	15	100%	100%	54,807 ms	24,353 ms
	S6 dual-arm assembly	14	100%	100%	36,008 ms	17,251 ms
Franka FR3	S1 pick-inspect-place	23	100%	100%	26,514 ms	13,320 ms
	S2 machine tending	32	100%	100%	37,219 ms	17,729 ms
	S3 parcel sorting	40	100%	100%	56,578 ms	27,598 ms
	S4 parallel cross-dock	13	100%	100%	16,525 ms	6428 ms
	S5 pharmacist verify	15	100%	100%	21,118 ms	9097 ms
	S6 dual-arm assembly	14	100%	100%	15,244 ms	6675 ms
Kinova Gen3	S1 pick-inspect-place	23	100%	100%	26,536 ms	13,354 ms
	S2 machine tending	32	100%	100%	35,861 ms	17,277 ms
	S3 parcel sorting	40	100%	100%	53,720 ms	25,860 ms
	S4 parallel cross-dock	13	100%	100%	17,129 ms	6665 ms
	S5 pharmacist verify	15	100%	100%	21,080 ms	9083 ms
	S6 dual-arm assembly	14	100%	100%	15,522 ms	6723 ms
Denso Cobotta	S1 pick-inspect-place	23	100%	100%	25,933 ms	12,325 ms
	S2 machine tending	32	100%	100%	34,979 ms	17,839 ms
	S3 parcel sorting	40	100%	100%	55,845 ms	27,176 ms
	S4 parallel cross-dock	13	100%	100%	14,196 ms	7098 ms
	S5 pharmacist verify	15	100%	100%	17,015 ms	8507 ms
	S6 dual-arm assembly	14	100%	100%	15,665 ms	7832 ms
Total		548				

method call sequences regardless of the execution backend. The six RTSF task files are used without any modification across all four platforms; only the per-robot YAML configuration file changes. Full timing data for each scenario-robot pair are reported in Table 8.

The scenario-specific details confirm that all control-flow constructs propagate correctly through the adapter interface. S2 introduces `event_trigger` steps (`wait_for_signal` adapter calls, 6 per robot), which appear in all backends with identical arguments and source/signal values. S4 exercises `parallel` branches via Python threads; the interleaved method calls differ in ordering across runs but the `compare_traces.py` metric sorts by step ID before comparison, yielding 100% sequence match. S5 introduces a `human_action` step that is auto-confirmed in all backends, producing a deterministic adapter call sequence. S6 exercises `multi_robot_sync` with two agents sharing the same adapter instance; the barrier synchronisation ensures both agents complete before the post-sync steps begin. S3 demonstrates that nested conditional branches with `scan` and `read_sensor` calls produce deterministic, equivalent traces across all backends.

Because the parser dispatches adapter calls via `getattr(self.robot, action_name)` without inspecting the backend type, and because all adapters return identical default values for `read_sensor`, `get_state`, and `wait_for_signal`, the call sequence is fully determined by the task JSON and the parser's control-flow logic. The 100% match is therefore a regression test that all adapters correctly implement the 19-method interface. The substantive findings are vendor independence, the same unmodified task files run on four robots from different manufacturers with only YAML configuration changes, and timing characterisation: per-action delays vary systematically across backends, so wall-clock performance is determined by the execution backend, not the task description.

The timing deltas vary systematically across platforms, reflecting differences in the execution backend rather than the task logic. UR5e (Gazebo Harmonic with DART physics engine) is consistently the slowest across all six scenarios because the physics simulation computes full rigid-body dynamics, collision detection, and friction at each control step. Franka FR3 and Kinova Gen3 (RViz controller stub, no physics engine) are faster than UR5e by a factor that depends on the scenario,

ranging from 1.15 $\times$ (S2) to 2.61 $\times$ (S4), since trajectory interpolation is performed by the `mock_components/GenericSystem` plugin without dynamic simulation. Franka and Kinova exhibit nearly identical timing (mean absolute difference 2.4%, maximum 5.1% on S3), as expected given that both use the same plugin with 7-DOF kinematics. Denso Cobotta (physical robot, BCAP protocol) is the fastest for S1 because it communicates directly with the physical controller without simulation overhead, though real motor acceleration limits and gripper actuation time impose physical constraints. For S2 and S3, the timing differences narrow as the number of move actions dominates total execution time across all platforms.

Because 100% sequence and argument equivalence guarantees an identical call sequence per scenario, cross-robot comparison within a scenario is sound while cross-scenario comparison is not, the six programs differing in their proportion of motion to sensor and signal calls. Normalising by adapter call count, the three non-physics backends cluster tightly at 1089–1414 ms per action across all eighteen combinations (medians 1217 ms Franka, 1236 ms Kinova, 1123 ms Cobotta), whereas UR5e has a higher median (2309 ms) and a far wider spread (1335–3654 ms). The spread, not the median, is informative: the physics engine makes per-action cost depend on what the action physically does, so the scenario dominated by short transfers between adjacent stations (S2, at 1335 ms, still inside the stub band) approaches stub-like cost, S1 and S3 sit above it at 1501 and 2045 ms, and those with concurrent branches or long repositioning motions (S6, S4 and S5, at 2572, 3313 and 3654 ms) diverge from it, which is why no single summary ratio would be adequate.

Two further regularities appear. The physical robot is among the fastest backends rather than the slowest, at 0.81–1.02 $\times$ the mean wall-clock time of the two controller-stub platforms across the six scenarios (0.81–1.04 $\times$ against either stub taken singly), so real motor dynamics are not the limiting factor at this granularity: the stubs' fixed trajectory-execution timing is, and a stub-based twin therefore yields a conservative rather than optimistic estimate. Second, the ratio of average per-action Δt to total execution time falls in a narrow band, 0.39–0.51 across all twenty-four combinations. Since the MockRobot baseline is effectively instantaneous, Δt approximates the mean relative timestamp of the robot run, which converges to half the total duration

precisely when per-action costs are near-uniform and incurred sequentially. Both conditions hold in S1–S3, where the ratio is 0.47–0.51 on every platform, and this independently indicates that no single call dominates them. The eight values below 0.45 are not noise but the signature of one condition or the other failing, and they are confined to the ROS 2 backends on S4, S5 and S6. In S4 and S6 it is sequentiality: the parser dispatches branches and per-agent steps on threads, so concurrency compresses total wall-clock without reducing the per-action costs that Δt averages, an effect that does not appear on the physical robot, whose single arm and single BCAP connection serialise the concurrent branches in practice, leaving its ratio at 0.50 on both scenarios. In S5, which is itself sequential, it is uniformity: a mean relative timestamp below half the total places the more expensive calls late in the trace, which matches a scenario ending in a transfer to the vial station after a comparatively cheap preparation subtask. The mock-run delta is therefore a sound linear estimator for scheduling only where both conditions hold; where they do not, it must be applied per branch or per phase rather than to the scenario as a whole.

7. Discussion

The three hypotheses of Section 1 can first be assessed against the results. H1 is supported within its stated scope: all twenty-four combinations produce identical call sequences and arguments from unmodified task files, with divergence confined to timing as predicted, the qualification being that portability is established for task logic in pre-taught workspaces, not for motion. H2 is supported including its sharper prediction: first-pass validity is imperfect (80% and 70%) while corrected validity is complete (100% on all 60 descriptions), and the predicted asymmetry holds, the correction loop repairing structural defects but leaving construct completeness unchanged. H3 is supported in direction but weakly in magnitude: recall degrades by roughly 14 pp while schema conformance is largely preserved, but the effect is concentrated in three of ten tasks, and the confound of Section 5.4, that prompt rules and task descriptions share an author, is only partly addressed by the independent benchmark.

7.1. Interpreting the multi-robot results

The 100% sequence and argument equivalence across all twenty-four combinations confirms that the adapter interface successfully decouples task logic from execution backend: the same RTSF parser traversal generates identical method calls regardless of whether the adapter is a no-op stub, a physics-based (UR5e) or controller-stub (Franka, Kinova) ROS 2 execution, or a physical robot using a proprietary protocol. The multi-robot result also validates vendor independence: the same RTSF files drive robots from four different manufacturers with different kinematics, joint-space dimensionalities (6 vs. 7 DOF), joint naming conventions, ROS 2 controller interfaces, and communication protocols, without any modification to the task description. The only adaptation required per robot is a YAML configuration file mapping named locations to joint configurations and specifying the action server path (or BCAP connection parameters). Beyond correctness, RTSF can serve as a digital twin interface [31,32]: the mock run enables rapid task validation before deploying to any conforming simulation or hardware, and the timing delta provides a practical estimate of real execution time for task scheduling and safety planning.

The portability demonstrated here concerns *task-logic portability*, running the same RTSF task file on different robot brands without modifying the task description. The current ROS2Adapter resolves RTSF named locations to pre-computed joint configurations stored in per-robot YAML files (13 locations per robot in our experiments). The approach works for fixed workspaces where joint configurations can be manually tuned once, but it does not perform inverse kinematics or collision avoidance. In a production deployment, a full motion planning

stack (e.g., MoveIt 2) would be interposed between the adapter and the robot controller. The remaining gap to additional physical hardware lies entirely in the adapter layer (motion planning, force/torque control), which is external to RTSF by design.

Together the six scenarios exercise all nine RTSF step types across six structurally distinct program patterns, on all four platforms (Table 7).

7.2. RTSF vs. Code-as-Policy

Both RTSF and Code-as-Policy [10] achieve high LLM generability from natural language task descriptions. The fundamental difference is the output representation: Code-as-Policy outputs executable Python code, which is maximally flexible but not portable, not formally validatable, and inseparably coupled to the robot API in the prompt. RTSF trades some flexibility (no arbitrary Python logic) for three guarantees: (1) schema-enforced structural correctness, catchable before execution; (2) vendor neutrality, enabling a task to run on any robot with a conforming adapter; and (3) human readability without programming knowledge [33]. The ablation study (Section 5) shows that roughly 14 pp of recall depends on prompt engineering rather than the format itself; this is a design opportunity, not a fundamental limitation: prompt rules can be updated as the format evolves without changing the schema.

Directly comparing RTSF with hand-authored Behaviour Tree or PDDL programs on common benchmarks would be methodologically ill-posed: BT.CPP XML and PDDL are execution-level formats requiring framework-specific programming expertise, whereas RTSF is an authoring format that targets them as output backends. The relevant evidence for RTSF correctness is (a) that the authoring-to-execution pipeline produces identical adapter call sequences across all backends, a regression-level verification across all six digital twin scenarios, and (b) that no surveyed format simultaneously offers vendor neutrality, formal schema validation, LLM generability, and human readability without programming knowledge (Table 1). Section 7.3 develops this comparison along a further axis, robustness, separating the failure modes on which RTSF is stronger than code generation from those on which it is not.

7.3. Robustness compared with alternative approaches

Comparing robustness requires first agreeing on what can fail, since the candidate approaches do not share a failure surface. For an interchange format three modes are distinguishable: a specification may be malformed, well-formed but incomplete, or complete but non-portable. RTSF is compared on each against the two alternatives targeting the same authoring role, LLM-generated code (Code-as-Policy, ProgPrompt) and library-bound representations (BT.CPP, PlanSys2).

On malformedness, RTSF converts what is elsewhere a runtime failure into a static one, with 100% corrected validity across all 60 descriptions from first-pass rates of 80% and 70%. The mechanism matters more than the rate: failures are detected before any motion command is issued, and the error message is specific enough to drive automated repair. Generated Python offers no equivalent guarantee, which is why ProgPrompt must introduce an empirical executability metric and why a semantically wrong but syntactically valid policy is discovered only when the robot moves. This categorical difference in failure surface is what makes an unsupervised generate-then-execute loop defensible.

On incompleteness RTSF is not more robust, and the evidence says so plainly: 9 of 30 independent tasks omit an expected construct, which the schema cannot detect since a task missing its `parallel` step is fully conformant. Code generation faces the same problem, a Python policy equally able to omit an implied retry loop, but lacks a construct-level annotation against which the omission could be counted. Both

approaches are at similar risk here; RTSF differs in being able to quantify it.

On portability the comparison is asymmetric by construction, and is where the strongest claim is available: the multi-platform experiment holds the task file fixed while varying vendor, middleware, joint-space dimensionality, and protocol, with identical results throughout. The equivalent operation for Code-as-Policy or ProgPrompt is not a configuration change but a re-prompt against a different API, after which correctness is again an empirical question; a BT.CPP tree is portable only among installations providing the same C++ leaf nodes, and a PlanSys2 specification does not exist outside ROS 2. Two qualifications apply. The equivalence is a regression-level guarantee rather than a discovery, since the parser dispatches without inspecting the adapter type, so it verifies adapter compliance rather than predicting robustness under adapter faults; and it holds for task logic in pre-taught workspaces, not for motion. What the experiment establishes is that the parser-to-adapter boundary is the only place vendor-specific knowledge enters, which is the precondition for portability rather than portability of a complete robot program.

Robustness against faults during execution is not evaluated here and should not be inferred: no experiment injects sensor, grasp, or communication failures, and the adapters return default values for state queries. Reactive frameworks hold a genuine advantage we do not contest, in that a Behaviour Tree re-evaluates its conditions on every tick whereas an RTSF task is traversed once and reacts only where the author placed a handler or trigger. Fault injection on the six scenarios is the natural next step, and the BT.CPP converter offers a path to reactive semantics for an RTSF-authored task rather than reimplementing them.

7.4. RTSF as a proposed open standard

RTSF occupies a position in the standards landscape that is currently unfilled. NIST CRCL [1] standardises motion-level commands (joint/Cartesian poses, gripper states) with an XSD schema, but has no concept of task structure, objects, conditions, or error handling. ISA-88 (IEC 61512) [20] standardises procedural control for batch process industries, but lacks parallelism, multi-robot synchronisation, and a machine-readable schema. RTSF complements both: it operates at the task abstraction level above CRCL's motion commands, adapts ISA-88's procedural hierarchy for discrete manufacturing, and adds a normative JSON Schema with `additionalProperties: false` for formal validation. The adapter interface (Section 4) serves as the interoperability boundary: any robot conforming to the 19-method adapter contract can execute RTSF tasks, analogous to how CRCL decouples motion software from vendor controllers but at a higher abstraction level. The MIT licence and versioned schema (`rtsf_schema_v1.0.json`) are intended to facilitate adoption as a de facto open standard for robot task interchange, and the planned v1.1 roadmap (recursive subtasks, resource locks, timing constraints) addresses gaps identified through comparison with existing industrial standards.

7.5. End-user development implications

RTSF is designed to be more accessible to domain experts than vendor-specific programming languages, though a basic understanding of JSON structure is assumed, aligning with end-user development (EUD) principles [34,35]. The JSON format is structured enough to be generated by Blockly-based visual editors (a planned future extension) and is designed to capture real-world tasks across manufacturing, logistics, laboratory, and healthcare domains. The `human_action` step explicitly models human participation, enabling collaborative robot applications where operators and robots share task responsibilities. A formal user study comparing authoring time and usability between RTSF and vendor-specific languages is identified as future work. However, preliminary evidence supports the viability of the underlying interaction paradigm: in related work [36], a chat-based LLM interface

Table 9

Qualitative comparison of RTSF against vendor-specific robot programming languages on five SME-relevant dimensions (✓ = fully supported; ~ = partial/workaround; ✗ = not supported). The “Human readability (non-programmer)” row reflects a design goal of RTSF; preliminary evidence from a related chat+Blockly prototype (12 non-programmer participants, SUS = 80.83) supports the underlying interaction paradigm, though a formal user study on RTSF itself remains future work.

Dimension	RTSF	URScript	RAPID	KRL
Vendor portability	✓	✗	✗	✗
Formal schema validation	✓	✗	✗	✗
LLM-generability (schema-guided)	✓	~	~	~
Human readability (non-programmer)	✓	✗	✗	✗
Motion-level control	✗	✓	✓	✓

combined with a Google Blockly graphic representation was evaluated for defining pick-and-place robot tasks, generating a custom, non-schema-validated JSON structure. A user study with 12 participants without programming or robotics background reported high usability (SUS = 80.83, Grade A) and low-to-moderate workload (NASA-TLX). While that system's ad-hoc JSON format covers only a subset of RTSF's constructs (loops and conditionals, lacking formal schema validation, error handling, parallelism, and multi-robot synchronisation), the positive usability results for the chat+block-based mediation paradigm motivate ongoing work to retarget its Blockly output to RTSF-conformant JSON, combining the demonstrated end-user accessibility of that interaction model with RTSF's formal validation and full control-flow coverage.

7.6. Adoption cost and portability for SMEs

The industrial motivation is worth stating explicitly. Reconfigurable production, in which a cell is retooled as product mixes change, presupposes that the knowledge describing what a cell does can be separated from the hardware currently executing it. Where that separation is absent, each change of robot model or vendor requires re-authoring the task in another proprietary language, and that cost falls hardest on smaller manufacturers, who have the least capacity to absorb integrator effort and the strongest incentive to reuse a task definition across a gradually renewed fleet. Analyses of Industry 4.0 adoption in European SMEs identify precisely this asymmetry, the financial and competence cost of adopting cyber-physical production technologies being the dominant barrier for smaller firms and a source of market exclusion relative to large manufacturers [37], which is the constituency a vendor-neutral task representation is intended to serve. The same separation lets a task description serve as the artefact exchanged between a digital twin and the shop floor [31,32].

A practical concern for industrial adoption is whether RTSF imposes a higher authoring cost than existing vendor-specific approaches. Table 9 compares RTSF against the three dominant proprietary robot programming languages (URScript (Universal Robots), RAPID (ABB), and KRL (KUKA)) across five dimensions relevant to SME adoption.

RTSF deliberately operates at the task-abstraction level (above motion planning). What vendor languages cannot offer simultaneously, portability across robot brands, formal validation that catches errors before execution, schema-guided LLM generation, and readability for non-programmers, RTSF achieves by delegating motion-level control to the adapter layer. For SMEs that author tasks once and deploy on different robot models as their fleet evolves, a common scenario in flexible manufacturing, the portability and validation properties alone can offset the loss of low-level control.

A structural comparison illustrates the authoring advantage. The pick-inspect-place scenario (S1, 231 lines of RTSF) requires no Cartesian coordinates, no joint configurations, and no vendor API calls: all spatial knowledge is encoded as named location identifiers resolved at execution time by the adapter. An equivalent URScript program

must additionally specify five pose variables (each requiring six joint angles), tool configuration, and socket-based sensor communication code, coupling the program to a single robot and workspace setup. The RTSF version is portable across all robots with a conforming adapter, and can be generated end-to-end from the natural-language description with nearly 98% construct recall (Section 5), reducing the authoring burden to prompt writing rather than robot programming.

7.7. Timing and real-time constraints

RTSF does not encode timing constraints beyond advisory `time-out_ms` fields. The digital twin experiment shows average per-action delays ranging from 6.4 s (Franka S4) to 39.8 s (UR5e S3) between mock and robot execution, with an overall mean of approximately 15 s across all twenty-four combinations: this is acceptable for pick-and-place tasks but may be insufficient for time-critical assembly processes. A planned `timing_constraints` section in RTSF v1.1 will support velocity profiles and joint-space path constraints, narrowing the gap to motion-level languages such as CRCL [1].

7.8. Perception, sensor fusion, and the adapter boundary

RTSF expresses conditions, event triggers, and sensor checks as symbolic predicates over named workspace entities. How such a predicate acquires its truth value is deliberately outside the format, and in the experiments reported here it is trivial, since all adapters return fixed defaults for state queries to keep traces deterministic. That is a legitimate simplification for validating the parser-to-adapter boundary, but it means these results say nothing about predicate reliability in a production cell.

That reliability would come from the adapter, and the multi-source fusion literature indicates what is achievable there: multimodal human action recognition combining Wi-Fi channel state information with smartwatch inertial sensing substantially outperforms either modality alone, the failure cases of an ambient RF channel and a body-worn inertial unit being largely uncorrelated [38]. The same principle applies to RTSF's predicates: a `human_action` step with `confirmation_required` currently needs explicit operator confirmation, whereas a fused estimate over vision, wearable inertial data, and ambient sensing could confirm the expected contribution without interrupting the operator, and grasp success is more reliably established by combining force/torque signals with visual verification than by either channel alone. This is an adjacent field and the analogy should not be overstated, but the architectural consequence is concrete: predicate evaluation sits behind the 19-method adapter interface, so fusion-based perception can be introduced without changing the schema or existing task files. The cost is that the reliability of a task's conditional behaviour becomes an adapter-level property the format cannot validate, the same boundary that limits motion-level guarantees (Section 7.9).

7.9. Known limitations

Several format limitations are acknowledged and motivate the RTSF v1.1 roadmap. RTSF operates strictly at the task-abstraction level: it does not encode joint configurations, Cartesian waypoints, or velocity profiles, leaving these to downstream adapters. Subtasks cannot call other subtasks, which prevents recursive task decomposition. Parallel branches that operate on the same workspace object cannot declare a resource lock. The PDDL and BT XML converters are one-way: neither target format retains the information needed to reconstruct safety, workspace, and HRI fields, so reverse conversion is not feasible.

On the validation side, the semantic validator is best-effort: it checks action name existence in `actions_library` and `$variable` scope correctness in subtask bodies, but does not verify that action argument values match the typed parameters declared in `actions_library`, nor that location or object references in `args` are

declared in `workspace`. These cross-type checks are left to the adapter at execution time. Physical robot integration has been tested on the Denso Cobotta (6-DOF, BCAP protocol), demonstrating 100% sequence and argument equivalence between mock and physical robot execution. Physical hardware validation on the three digital twin platforms (UR5e, Franka Emika FR3, Kinova Gen3) remains future work. Bridging to additional hardware requires only one adapter implementation per robot; the task `.json` file and parser are hardware-agnostic by design.

The v1.1 roadmap addresses the structural limitations above: a `max_depth` parameter will enable recursive subtask calls, a `resource_lock` primitive will guard parallel access to shared workspace objects, and a `timing_constraints` section will support velocity profiles and joint-space path constraints.

7.10. Reproducibility

All code, examples, and the JSON Schema are released under the MIT licence. The 30 main NL task descriptions, 10 ablation descriptions, and 30 independent benchmark descriptions are versioned in the repository. The full test suite (107 tests) is deterministic and requires no network access. LLM evaluation results are reproducible with the fixed configuration (`-model gpt-5.4-mini -temperature 0.2` and `-model meta-llama/llama-4-scout-17b-16e-instruct -temperature 0.2`), subject to provider model updates (see Section 5.4). The independent benchmark can be reproduced with `python evaluate_llm.py -tasks examples/llm_generated/tasks_independent.json -out results/llm_evaluation_independent.csv`.

8. Conclusion

The central result is vendor independence: six structurally distinct task scenarios ran on four robot platforms (UR5e, Franka Emika FR3, Kinova Gen3, Denso Cobotta) using the same unmodified RTSF files, accumulating over 500 adapter calls with identical method-call sequences and arguments across all backends. Because the parser dispatches calls without inspecting the adapter type, this match holds by construction and serves as a regression test of adapter compliance. The runtime architecture, a recursive parser, PDDL and BehaviorTree.CPP converters, and a 19-method adapter interface, enables a single RTSF file to drive mock, simulated, and planned execution without modification, with sub-second end-to-end processing confirmed at up to 5000 steps.

On the generation side, schema-guided prompting achieved high construct recall on 30 natural-language task descriptions across four domains, with all outputs valid after self-correction. An independent benchmark of 30 additional descriptions, authored by three evaluators unfamiliar with RTSF, confirmed generalisability with a modest recall drop and the same pipeline. A prompt ablation study showed that removing disambiguation and field-name guidance rules degrades recall by double-digit percentage points, with the degradation roughly evenly split between the two rule categories.

RTSF v1.0 itself is an open, schema-validated, vendor-neutral JSON format for describing robot manipulator tasks at the task abstraction level. The format defines nine step types covering all six control-flow constructs relevant to manufacturing and service robots, governed by a normative JSON Schema (draft-07) with `additionalProperties: false` on all objects. A gap analysis against ten existing formats shows that, to our knowledge, no prior open format combines all of these properties. By providing a formal schema, a versioned specification, and a well-defined adapter interface, RTSF is positioned as a proposed open standard for vendor-independent robot task interchange, complementing existing standards such as NIST CRCL (motion level) and ISA-88 (process control) at the task abstraction level.

Four bounds delimit what the format can be claimed to do today, detailed in Section 7.9: a deliberate one of abstraction level, since RTSF encodes no motion-level knowledge and portability is therefore

established for task logic in pre-taught workspaces; an expressive one, with no recursive subtasks, no resource locks, advisory timing, and one-way converters; one of validation, the semantic validator being best-effort and deferring cross-type checks to execution; and one of evidence, physical hardware validation covering one platform of four, the LLM results resting on 60 descriptions in four domains, and the accessibility claim on a study of a related prototype rather than of RTSF itself. The measured per-action delays of 6.4 to 39.8 s also confine current applicability to pick-and-place and machine-tending granularities, not time-critical assembly.

Within those bounds, the applications follow from the reuse argument of Section 7.6: a validated task file is an asset that outlives the hardware it was written for, and the same file is the artefact exchanged between a digital twin and the shop floor. Two further settings are worth naming. Where production data must reach an MES or ERP layer, a schema-validated task description is a natural payload for OPC UA companion specifications [39]. And because the format is generated rather than written in the pipeline evaluated here, it offers a validation boundary for autonomous agents: a model proposes a task, the schema can reject it before any motion is commanded, and a human can read the result without knowing a vendor language.

On the format side, v1.1 will introduce recursive subtask calls, shared resource locks, and a `timing_constraints` section. Tooling efforts are underway for a Blockly-based visual editor that exports RTSF JSON, a REST API validator, and extended ROS2Adapter support for additional robot platforms. Evaluation priorities include real-hardware adapter validation on UR5e, Franka Emika FR3, and Kinova Gen3, integration with OPC UA companion specifications for MES-connected manufacturing cells [39], and a publicly shared, independently annotated benchmark of natural-language robot task descriptions. The independent benchmark reported in Section 5.2.1 is a first step in this direction.

Four directions are specific enough to be actionable. The most immediate closes the motion-level gap by interposing a planning stack such as MoveIt 2 between adapter and controller, replacing pre-taught joint configurations with resolved inverse kinematics and collision-aware trajectories, and so extending the portability claim from task logic to execution in unstructured workspaces. The second concerns validation strength: the cross-type checks currently deferred to the adapter are decidable statically, and implementing them would support a stronger pre-deployment correctness statement, of the kind type checking provides for programming languages. The third is empirical and is the one on which the end-user development claim ultimately depends: a controlled study comparing authoring time, error rate, and perceived workload between RTSF authored through a chat-and-blocks interface and equivalent URScript, RAPID, or KRL programs, with non-programmer domain experts as participants. The fourth is the grounding of task-level predicates, which the format leaves to the adapter and which is where robustness in realistic settings is won or lost (Section 7.8). Adoption as an interchange format would additionally require a conformance test suite for third-party adapters, which would turn the 19-method interface into a checkable contract rather than a documented convention.

Code availability

All code, the JSON Schema, and more than 70 validated example tasks are openly available under the MIT licence at a public repository.

Declaration of generative AI and AI-assisted technologies in the manuscript preparation process

During the preparation of this work the author used ChatGPT and Grammarly in order to assist with grammar and spelling checking, paraphrasing, and rewording. After using these tools, the author reviewed and edited the content as needed and takes full responsibility for the content of the published article.

Funding

This research did not receive any specific grant from funding agencies in the public, commercial, or not-for-profit sectors.

Declaration of competing interest

The author declares no conflicts of interest or competing interests.

Acknowledgements

The author thanks the independent second annotator for annotating the 30 task descriptions and assessing the semantic accuracy of the generated RTSF tasks, and the three external evaluators for authoring and annotating the 30 independent benchmark task descriptions used in Section 5.2.1.

Appendix. RTSF JSON schema (draft-07)

The following listing contains the complete normative JSON Schema governing all RTSF v1.0 task files. Every object enforces `additionalProperties: false`, so any field not defined below is rejected at validation time. The schema is used both for offline structural validation (`rtsf/validator.py`) and as an in-context constraint for schema-guided LLM generation (Section 5).

Listing 4: Complete RTSF v1.0 JSON Schema
(`rtsf_schema_v1.0.json`).

```

1 {
2   "$schema": "http://json-schema.org/draft-07/schema",
3   "#",
4   "$id": "https://raw.githubusercontent.com/
5     luigigargioni/robot_task_specification_format
6     /main/schema/rtsf_schema_v1.0.json",
7   "title": "RTSF - Robot Task Specification Format",
8   "description": "Open JSON-based format for vendor-
9     independent description of robot manipulator
10    tasks.",
11   "type": "object",
12   "required": ["rtsf_version", "task"],
13   "additionalProperties": false,
14   "properties": {
15     "$schema": { "type": "string" },
16     "rtsf_version": {
17       "type": "string",
18       "description": "RTSF schema version, e.g. '1.0'
19     },
20     "task": { "$ref": "#/definitions/task" }
21   },
22   "definitions": {
23     "task": {
24       "type": "object",
25       "required": ["id", "name", "domain", "
26         workspace", "program"],
27       "additionalProperties": false,
28       "properties": {
29         "id": { "type": "string" },
30         "name": { "type": "string" },
31         "domain": { "type": "string" },
32         "description": { "type": "string" },
33         "metadata": { "$ref": "#/definitions/
34           metadata" },
35         "safety": { "$ref": "#/definitions/
36           safety" },
37         "subtasks": {
38           "type": "array",
39           "items": { "$ref": "#/definitions/subtask"
40         },
41         "workspace": { "$ref": "#/definitions/
42           workspace" },
43         "actions_library": {
44           "type": "array",

```

```

38     "items": { "$ref": "#/definitions/
39         action_definition" }
40 },
41     "program": { "$ref": "#/definitions/
42         program" },
43     "termination": { "$ref": "#/definitions/
44         termination" }
45 },
46 "metadata": {
47     "type": "object",
48     "additionalProperties": false,
49     "properties": {
50         "author": { "type": "string" },
51         "created": { "type": "string", "format": "
52             date" },
53         "robot": { "type": "string" },
54         "version": { "type": "string" },
55         "license": { "type": "string" }
56     },
57     "safety": {
58         "type": "object",
59         "additionalProperties": false,
60         "properties": {
61             "max_speed_ms": { "type": "number
62                 ", "minimum": 0 },
63             "max_force_n": { "type": "number
64                 ", "minimum": 0 },
65             "max_payload_kg": { "type": "number
66                 ", "minimum": 0 },
67             "human_proximity_zone_m": { "type": "number
68                 ", "minimum": 0 },
69             "on_proximity_detected": {
70                 "type": "string",
71                 "enum": ["pause", "stop", "slow_down"]
72             },
73             "emergency_stop": {
74                 "type": "object",
75                 "additionalProperties": false,
76                 "properties": {
77                     "trigger": { "type": "string" },
78                     "return_to": { "type": "string" }
79                 }
80             },
81             "timeout_global_ms": { "type": "integer", "
82                 minimum": 0 }
83         },
84     },
85     "workspace": {
86         "type": "object",
87         "required": ["objects", "locations", "agents"]
88         ,
89         "additionalProperties": false,
90         "properties": {
91             "objects": {
92                 "type": "array",
93                 "items": { "$ref": "#/definitions/
94                     workspace_object" }
95             },
96             "locations": {
97                 "type": "array",
98                 "items": { "$ref": "#/definitions/location
99                     " }
100             },
101             "agents": {
102                 "type": "array",
103                 "minItems": 1,
104                 "items": { "$ref": "#/definitions/agent" }
105             }
106         },
107     },
108     "workspace_object": {
109         "type": "object",
110         "required": ["id", "type"],
111         "additionalProperties": false,
112         "properties": {
113             "id": { "type": "string" },
114             "type": { "type": "string" },
115             "properties": { "type": "object" },
116             "initial_location": { "type": "string" }
117         }
118     }
119 }

```

```

112 },
113
114 "location": {
115     "type": "object",
116     "required": ["id", "type"],
117     "additionalProperties": false,
118     "properties": {
119         "id": { "type": "string" },
120         "type": {
121             "type": "string",
122             "enum": ["source", "destination", "discard
123                 ", "home",
124                 "processing", "storage", "generic
125                 " ]
126         },
127         "frame": { "type": "string" }
128     },
129 },
130 "agent": {
131     "type": "object",
132     "required": ["id", "type"],
133     "additionalProperties": false,
134     "properties": {
135         "id": { "type": "string" },
136         "type": {
137             "type": "string",
138             "enum": ["cobot", "industrial_robot", "
139                 mobile_robot", "
140                 human", "virtual"]
141         },
142         "initial_location": { "type": "string" },
143         "capabilities": {
144             "type": "array",
145             "items": { "type": "string" }
146         },
147         "role": { "type": "string" }
148     },
149 },
150 "action_definition": {
151     "type": "object",
152     "required": ["id", "parameters"],
153     "additionalProperties": false,
154     "properties": {
155         "id": { "type": "string" },
156         "description": { "type": "string" },
157         "parameters": {
158             "type": "array",
159             "items": { "type": "string" }
160         },
161         "preconditions": {
162             "type": "array",
163             "items": { "type": "string" }
164         },
165         "effects": {
166             "type": "array",
167             "items": { "type": "string" }
168         }
169     },
170 },
171 "subtask": {
172     "type": "object",
173     "required": ["id", "name", "steps"],
174     "additionalProperties": false,
175     "properties": {
176         "id": { "type": "string" },
177         "name": { "type": "string" },
178         "description": { "type": "string" },
179         "parameters": {
180             "type": "array",
181             "items": { "$ref": "#/definitions/
182                 subtask_parameter" }
183         },
184         "steps": {
185             "type": "array",
186             "items": { "$ref": "#/definitions/step" }
187         }
188     },
189 },
190 "subtask_parameter": {
191     "type": "object",
192     "required": ["name", "type"],
193     "additionalProperties": false,

```

```

194   "properties": {
195     "name": { "type": "string" },
196     "type": {
197       "type": "string",
198       "enum": ["object_ref", "location_ref", "string",
199               "integer", "float", "boolean"]
200     },
201     "default": {}
202   },
203 },
204
205 "program": {
206   "type": "object",
207   "required": ["steps"],
208   "additionalProperties": false,
209   "properties": {
210     "steps": {
211       "type": "array",
212       "minItems": 1,
213       "items": { "$ref": "#/definitions/step" }
214     }
215   },
216 },
217
218 "condition": {
219   "type": "object",
220   "required": ["type", "operator", "value"],
221   "additionalProperties": false,
222   "properties": {
223     "type": {
224       "type": "string",
225       "enum": ["sensor_check", "state_check",
226               "message_check", "counter_check"]
227     },
228     "sensor": { "type": "string" },
229     "object": { "type": "string" },
230     "property": { "type": "string" },
231     "operator": {
232       "type": "string",
233       "enum": ["==", "!=", ">", "<", ">=", "<="]
234     },
235     "value": {}
236   },
237 },
238
239 "step": {
240   "type": "object",
241   "required": ["id", "type"],
242   "additionalProperties": false,
243   "properties": {
244     "id": { "type": "string" },
245     "type": {
246       "type": "string",
247       "enum": [
248         "action", "loop", "conditional", "parallel",
249         "subtask_call", "human_action", "event_trigger",
250         "error_handler", "multi_robot_sync"
251       ]
252     },
253     "agent": { "type": "string" },
254     "timeout_ms": { "type": "integer", "minimum": 0 },
255     "on_timeout": {
256       "type": "string",
257       "enum": ["abort", "retry", "skip", "escalate"]
258     },
259     "retry_count": { "type": "integer", "minimum": 1 },
260
261     "action": { "type": "string" },
262     "args": { "type": "object" },
263
264     "loop_type": {
265       "type": "string",
266       "enum": ["count", "while", "for_each"]
267     },
268     "count": {},
269     "iterator": { "type": "string" },
270     "collection": { "type": "string" },
271     "condition": { "$ref": "#/definitions/condition" },
272

```

```

273   "body": {
274     "type": "array",
275     "items": { "$ref": "#/definitions/step" }
276   },
277   "then": {
278     "type": "array",
279     "items": { "$ref": "#/definitions/step" }
280   },
281   "else": {
282     "type": "array",
283     "items": { "$ref": "#/definitions/step" }
284   },
285
286   "branches": {
287     "type": "array",
288     "items": { "$ref": "#/definitions/parallel_branch" }
289   },
290   "sync": {
291     "type": "string",
292     "enum": ["wait_all", "wait_any", "fire_and_forget"]
293   },
294
295   "subtask": { "type": "string" },
296
297   "description": { "type": "string" },
298   "confirmation_required": { "type": "boolean" },
299
300   "event": { "$ref": "#/definitions/event" },
301   "on_trigger": {
302     "type": "array",
303     "items": { "$ref": "#/definitions/step" }
304   },
305
306   "error_types": {
307     "type": "array",
308     "items": { "type": "string" }
309   },
310   "on_error": {
311     "type": "array",
312     "items": { "$ref": "#/definitions/step" }
313   },
314   "resume": { "type": "boolean" },
315   "max_retries": { "type": "integer", "minimum": 1 },
316
317   "agents": {
318     "type": "array",
319     "items": { "type": "string" }
320   },
321   "sync_type": {
322     "type": "string",
323     "enum": ["barrier", "pipeline"]
324   },
325   "steps_per_agent": {
326     "type": "object",
327     "additionalProperties": {
328       "type": "array",
329       "items": { "$ref": "#/definitions/step" }
330     }
331   },
332 },
333 },
334
335 "parallel_branch": {
336   "type": "object",
337   "required": ["id", "steps"],
338   "additionalProperties": false,
339   "properties": {
340     "id": { "type": "string" },
341     "agent": { "type": "string" },
342     "steps": {
343       "type": "array",
344       "items": { "$ref": "#/definitions/step" }
345     }
346   },
347 },
348
349 "event": {
350   "type": "object",
351   "required": ["type", "source", "signal"],

```

```

352     "additionalProperties": false,
353     "properties": {
354         "type": {
355             "type": "string",
356             "enum": ["external_signal", "timer",
357                     "message", "state_change"]
358         },
359         "source": { "type": "string" },
360         "signal": { "type": "string" },
361         "timeout_ms": { "type": "integer", "minimum":
362                        : 0 },
363         "on_timeout": {
364             "type": "string",
365             "enum": ["abort", "skip", "escalate"]
366         }
367     },
368
369     "termination": {
370         "type": "object",
371         "required": ["type"],
372         "additionalProperties": false,
373         "properties": {
374             "type": {
375                 "type": "string",
376                 "enum": ["step_complete", "condition_met",
377                         "event_received", "manual"]
378             },
379             "step": { "type": "string" },
380             "condition": { "$ref": "#/definitions/
381                           condition" },
382             "post_actions": {
383                 "type": "array",
384                 "items": { "$ref": "#/definitions/step" }
385             }
386         }
387     }
388 }
389 }

```

Data availability

The task descriptions, evaluation scripts, and experimental traces used in this study are available in the project repository https://github.com/luigigargioni/robot_task_specification_format and from the corresponding author upon request.

References

- [1] Proctor F, Balakirsky S, Kootbally Z, Kramer T, Schlenoff C, Shackleford W. The canonical robot command language (CRL). *Ind Robot: An Int J* 2016;43(5):495–502.
- [2] Fox M, Long D. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *J Artificial Intelligence Res* 2003;20:61–124.
- [3] Höller D, Behnke G, Bercher P, Biundo S, Fiorino H, Pellier D, Alford R. HDDL: An extension to PDDL for expressing hierarchical planning problems. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34, 2020, p. 9883–91.
- [4] Ghallab M, Nau D, Traverso P. *Automated planning: theory and practice*. San Francisco, CA: Morgan Kaufmann; 2004.
- [5] Colledanchise M, Ögren P. *Behavior trees in robotics and AI: An introduction*. Boca Raton, FL: CRC Press; 2018.
- [6] Iovino M, Scutkins E, Styrd J, Ögren P, Smith C. A survey of behavior trees in robotics and AI. *Robot Auton Syst* 2022;154:104096.
- [7] Cashmore M, Fox M, Long D, Magazzeni D, Ridder B, Carrera A, Palomeras N, Hurtos N, Carreras M. ROSPlan: Planning in the robot operating system. In: *Proceedings of the international conference on automated planning and scheduling*. Vol. 25, 2015, p. 333–41.
- [8] Martín F, Ginés Clavero J, Matellán V, Rodríguez FJ. PlanSys2: A planning system framework for ROS2. In: *2021 IEEE/RSJ international conference on intelligent robots and systems. IROS, IEEE*; 2021, p. 9742–9.
- [9] Mayr M, Rovida F, Krueger V. SkiROS2: A skill-based robot control platform for ROS. In: *2023 IEEE/RSJ international conference on intelligent robots and systems. IROS, IEEE*; 2023, p. 6273–80.
- [10] Liang J, Huang W, Xia F, Xu P, Hausman K, Ichter B, Florence P, Zeng A. Code as policies: Language model programs for embodied control. In: *IEEE international conference on robotics and automation. ICRA, IEEE*; 2023, p. 9493–500.
- [11] Singh I, Blukis V, Mousavian A, Goyal A, Xu D, Tremblay J, Fox D, Thomason J, Garg A. ProgPrompt: Program generation for situated robot task planning using large language models. *Auton Robots* 2023;47(8):999–1012.
- [12] Gargioni L, Fogli D. Integrating chatgpt with blockly for end-user development of robot tasks. In: *Companion of the 2024 ACM/IEEE international conference on human-robot interaction*. 2024, p. 478–82.
- [13] Gargioni L, Fogli D, Baroni P. Preparation of personalized medicines through collaborative robots: A hybrid approach to the end-user development of robot programs. *ACM J Responsible Comput* 2025;2(3):1–26.
- [14] Huang W, Xia F, Xiao T, Chan H, Liang J, Florence P, Zeng A, Tompson J, Mordatch I, Chebotar Y, Sermanet P, Brown N, Jackson T, Luu L, Levine S, Hausman K, Ichter B. Inner monologue: Embodied reasoning through planning with language models. In: *Proceedings of the 6th conference on robot learning (coRL)*. *Proceedings of machine learning research*, vol. 205, PMLR; 2023, p. 1769–82.
- [15] McDermott D, Ghallab M, Howe A, Knoblock C, Ram A, Veloso M, Weld D, Wilkins D. PDDL — The planning domain definition language. Technical Report CVC TR-98-003, Yale Center for Computational Vision and Control; 1998.
- [16] Helmert M. The fast downward planning system. *J Artificial Intelligence Res* 2006;26:191–246.
- [17] Coles A, Coles A, Fox M, Long D. Forward-chaining partial-order planning. In: *Proceedings of the international conference on automated planning and scheduling*. Vol. 20, 2010, p. 42–9.
- [18] Macenski S, Martín F, White R, Clavero JG. The marathon 2: A navigation system. In: *2020 IEEE/RSJ international conference on intelligent robots and systems. IROS, IEEE*; 2020, p. 2718–25.
- [19] Macenski S, Foote T, Gerkey B, Lalancette C, Woodall W. Robot operating system 2: Design, architecture, and uses in the wild. *Sci Robot* 2022;7(66):eabm6074.
- [20] International Society of Automation. ANSI/ISA-88.00.01-2010: batch control, part 1: Models and terminology. 2010, American National Standard (equivalent to IEC 61512-1).
- [21] Ichter B, Brohan A, Chebotar Y, Finn C, Hausman K, Herzog A, Ho D, Ibarz J, Irpan A, Jang E, et al. Do as I can, not as I say: Grounding language in robotic affordances. In: *Proceedings of the 6th conference on robot learning (coRL)*. *Proceedings of machine learning research*, vol. 205, PMLR; 2023, p. 287–318.
- [22] Zitkovich B, Yu T, Xu S, Xu P, Xiao T, Xia F, Wu J, Wohlhart P, Welker S, Wahid A, et al. RT-2: Vision-language-action models transfer web knowledge to robotic control. In: *Proceedings of the 7th conference on robot learning (coRL)*. *Proceedings of machine learning research*, vol. 229, PMLR; 2023, p. 2165–83.
- [23] Huang W, Wang C, Zhang R, Li Y, Wu J, Fei-Fei L. VoxPoser: Composable 3D value maps for robotic manipulation with language models. In: *Proceedings of the 7th conference on robot learning (coRL)*. *Proceedings of machine learning research*, vol. 229, PMLR; 2023, p. 540–62.
- [24] Ahn S, Choi W, Lee J, Park J, Woo H. Towards reliable code-as-policies: A neuro-symbolic framework for embodied task planning. In: *Advances in neural information processing systems. neurIPS 2025*, 2025, Spotlight presentation; preprint arXiv:2510.21302.
- [25] Grigorev DS, Kovalev AK, Panov AI. VerifyLLM: LLM-based pre-execution task plan verification for robots. 2025, <http://dx.doi.org/10.48550/arXiv.2507.05118>, arXiv preprint arXiv:2507.05118.
- [26] Pesjak D, Žabkar J. Robot planning via LLM proposals and symbolic verification. *Mach Learn Knowl Extr* 2026;8(1):22.
- [27] Bian S, Zhang Y, Tian G, Miao Z, Wu EQ, Yang SX, Hua C. Large language model-based task planning for service robots: A review. 2025, <http://dx.doi.org/10.48550/arXiv.2510.23357>, arXiv preprint arXiv:2510.23357.
- [28] Sapkota R, Cao Y, Roumeliotis KI, Karkee M. Vision-language-action (VLA) models: Concepts, progress, applications and challenges. 2025, <http://dx.doi.org/10.48550/arXiv.2505.04769>, arXiv preprint arXiv:2505.04769.
- [29] International Organization for Standardization. ISO/TS 15066:2016 — Robots and robotic devices — Collaborative robots. 2016, International Standard.
- [30] Landis JR, Koch GG. The measurement of observer agreement for categorical data. *Biometrics* 1977;33(1):159–74.
- [31] Grieves M, Vickers J. Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems. In: *Transdisciplinary perspectives on complex systems: new findings and approaches*. Springer; 2017, p. 85–113.
- [32] Tao F, Cheng J, Qi Q, Zhang M, Zhang H, Sui F. Digital twin-driven product design, manufacturing and service with big data. *Int J Adv Manuf Technol* 2018;94(9–12):3563–76.
- [33] Fogli D, Gargioni L, Guida G, Tampalini F. A hybrid approach to user-oriented programming of collaborative robots. *Robot Comput-Integr Manuf* 2022;73:102234.
- [34] Lieberman H, Paternò F, Klann M, Wulf V. End-user development: An emerging paradigm. In: *End user development*. Springer; 2006, p. 1–8.
- [35] Barricelli BR, Cassano F, Fogli D, Piccinno A. End-user development, end-user programming and end-user software engineering: A systematic mapping study. *J Syst Softw* 2019;149:101–37.

- [36] Gargioni L, Fogli D, Nava EM. A digital twin-enabled environment for end-user development of robot tasks. *Intell Serv Robot* 2026.
- [37] Nagy M, Lăzăroiu G, Valaskova K. Machine intelligence and autonomous robotic technologies in the corporate context of SMEs: Deep learning and virtual simulation algorithms, cyber-physical production networks, and Industry 4.0-based manufacturing systems. *Appl Sci* 2023;13(3):1681.
- [38] Wang J, Cai X, Dai X, Xia M, Shi C, Chen W. Multimodal fusion-based human action recognition using Wi-Fi CSI and smartwatch sensors. *IEEE Internet Things J* 2026;13(14):31652–64.
- [39] OPC Foundation and VDMA. OPC UA for robotics — Part 1: Vertical integration. 2025, Available at <https://reference.opcfoundation.org/specs/OPC-40010-1>, OPC Foundation Companion Specification OPC 40010-1, version 1.02.