Autonomous Skill Development: Neuro-Symbolic Agents that Self-Generate Verifiable Tools

Elia Pacioni[1,2], Valerio Crocetti[1,3], Andrei C. Coman[1], Davide Calvaresi[1], Gaetano Manzo[1], and Michael Ignaz Schumacher[1]

[1] HES-SO Valais-Wallis, 3960 Sierre, Switzerland
{elia.pacioni, valerio.crocetti, andrei.coman, davide.calvaresi, gaetano.manzo, michael.schumacher}@hevs.ch,
[2] University of Extremadura, 06800 Mérida, Spain
[3] Università Politecnica delle Marche, 60121, Ancona, Italy

Abstract. Large language model (LLM)-based agents can address highly heterogeneous tasks. However, their result and justifications are generated through opaque (non-inspectable) sub-symbolic processing. Thus, even a plausible explanation may be post-hoc and unfaithful to the process that produced the answer. Such behavior undermines safety, auditability, and accountability in high-stakes domains such as healthcare and finance. This paper presents Autonomous Skill Development (ASD), a neuro-symbolic agent architecture that separates open-ended language reasoning from deterministic computation. Rather than relying on the LLM to repeatedly solve and explain computational problems, ASD dynamically generates, validates, stores, and reuses explicit symbolic skills. A neuro-symbolic router first determines whether a request contains a deterministic computation that can be extracted and formalized. When such a computation is identified, ASD generates a skill as a pure function exposed through an interoperable Model Context Protocol (MCP) tool. The skill is then evaluated through a layered validation pipeline that combines deterministic tests with an independent LLM judge and, once validated, is stored in a reusable, versioned skill library. Subsequent computations are performed by inspectable code, while the LLM is restricted to interpreting and communicating the resulting output. Each answer is therefore accompanied by explicit provenance identifying the skill executed, the input values provided, and the result produced. Unlike a free-form LLM justification, this trace is faithful by construction because it directly reflects the computation that generated the answer.

Dynamic skill generation is essential for scalability, as manually implementing rules for every possible deterministic subtask would be impractical. We evaluate ASD in the nutrition domain by examining the skills it generates and validates, and assessing whether delegating deterministic computation to verified tools allows smaller language models to retain correctness.

Keywords: Agentic AI · LLM-based Agents · MCP Tools · Agent Skills · Neuro-Symbolic AI

1 Introduction

Large language models (LLMs) increasingly support agents that address heterogeneous tasks by interpreting requests, planning actions, and invoking external tools [19]. However, their flexibility comes with limited process transparency. This limitation is particularly consequential when a request contains a deterministic computation, such as applying formulas, consulting structured tables, or performing unit conversions. Such operations admit reproducible and directly inspectable solutions. Nevertheless, to date, both the result and its justification are produced through non-inspectable sub-symbolic computations. Best-case scenario, an explanation can be produced by post-hoc approaches. However, such explanations (LLM- and post-hoc-based) can reflect the process that produced the answer unfaithfully [11]. In high-stakes domains such as healthcare and finance, this makes incorrect calculations difficult to detect, audit, and contest [18]. A natural response is to separate responsibilities: the LLM can interpret requests and communicate results, while deterministic computations are delegated to explicit executable procedures. Tool-augmented language models already follow this principle by offloading arithmetic, logical, and information-retrieval operations to external resources [19]. However, static tool libraries provide only limited coverage because their developers must anticipate the computations that users will request. Generating tools on demand can address this limitation. However, it introduces a further challenge: a dynamically generated procedure should not be executed or reused unless its behavior has been assessed first. A scalable architecture must therefore determine when a request contains an extractable deterministic component, generate an appropriate procedure, validate it before use, and preserve an inspectable record of its subsequent execution.

This paper presents Autonomous Skill Development (ASD), a neuro-symbolic agent architecture that combines neural language understanding with explicit rule-based computation. Whereas existing tool-making approaches primarily emphasize tool generation and reuse, ASD makes validation a prerequisite for admitting a generated skill to its library. A neuro-symbolic router first determines whether a request contains an extractable deterministic core. When such a core is identified, ASD generates a skill as a pure, typed function exposed through the Model Context Protocol (MCP) [4]. The generated skill is evaluated through a layered validation pipeline comprising deterministic tests (L1), semantic assessment by an LLM judge distinct from the generating model (L2) [29], and optional human review (L3). Skills that fail validation are returned to a repair loop, while accepted skills are stored in a reusable and versioned library. Once a skill has been admitted, the corresponding computation is performed by explicit executable code rather than regenerated through the LLM at each request. The LLM is subsequently used to contextualize and communicate the result. ASD also records the skill invoked, the input values supplied to it, and the output it produced. This computational provenance is faithful by construc-

[4] `https://modelcontextprotocol.io`

tion in a precise sense: it describes the function call that actually generated the reported value, rather than reconstructing a possible reasoning process after the fact. The trace is therefore inspectable and reproducible, although its semantic correctness remains dependent on the adequacy of the skill-generation and validation process. We evaluate ASD on a controlled nutrition-domain benchmark containing 38 tasks: 28 deterministic tasks spanning 18 computation families and 10 non-deterministic tasks used to assess routing behavior. The evaluation considers two open-weight model families, Gemma 4 [7] and Qwen 3.5 [22], with different model sizes assigned to routing, skill generation, validation, and execution roles. We assess routing accuracy, generated skill quality, tool selection reliability, end-to-end task correctness, and computational cost. All experiments are conducted with locally hosted models, without external model APIs, demonstrating the architecture's feasibility on a single workstation.

The contribution of this paper is threefold. It presents:

- ASD, a neuro-symbolic agent architecture that identifies deterministic components of user requests and dynamically generates, validates, versions, and reuses executable skills exposed as MCP tools.
- An evaluation that separates routing, skill generation, skill validation, tool selection, and execution, thereby distinguishing failures caused by incorrect skills from those caused by selecting or invoking the wrong skill.
- Evidence that offloading deterministic computation to validated skills preserves correctness for smaller execution models. As execution models shrink, tool selection, rather than skill quality, becomes the dominant bottleneck, motivating tool retrieval for large libraries [20].

2 Related Work

Recent studies explore hybrid approaches in which LLMs delegate part of the reasoning process to symbolic or executable components. Faithful CoT [11] shows that explanations are more reliable when the symbolic reasoning chain is the one actually used by a deterministic solver to compute the answer. In a related neuro-symbolic direction, Crocetti and Dragoni [3] connect LLMs with logic programming, using the language model as an interface toward Prolog-based [8] symbolic reasoning.

A growing line of work moves from one-shot symbolic execution to reusable, self-generated tools. VOYAGER [25] builds an ever-growing library of executable skills refined through feedback and self-verification. LATM [2] uses a powerful model to generate reusable Python [24] tools that a cheaper model can later invoke through a functional cache. CRAFT [28] constructs specialized tool-sets offline and retrieves the relevant tool at inference time. TroVE [26] induces reusable functions while solving programmatic tasks, keeping a candidate when the solution that uses it executes correctly. APIGen [10] automates the generation of verifiable function-calling data, and ToolVerifier [13] adds a self-verification step that distinguishes between close candidates during tool selection.

ASD builds on these directions but differs in its validation regime. In LATM and CRAFT, a tool is reused once created, and in TroVE, a function is accepted based on its execution inside the generator's loop. Conversely, ASD admits a skill only upon validation of a pipeline external to the generator. Moreover, unlike Tool Verifier's self-verification, the judging model is distinct from the one that produced the skill. Doing so shifts trust from the generator's self-assessment to an independent check, mitigating self-bias in the validation process.

3 Methodology

Some requests have a deterministic computational core: a formula, a conversion, a lookup. When a language model enacts these on itself, computation and explanation are fused and non-inspectable. The model's output and justification come from the same opaque process, with no way to verify one against the other. ASD addresses it directly. For requests with a deterministic core, the computation moves out of the model and into verified code. For each request, the agent decides whether a deterministic core exists. If so, it generates, validates, and exposes the corresponding code as a reusable tool.

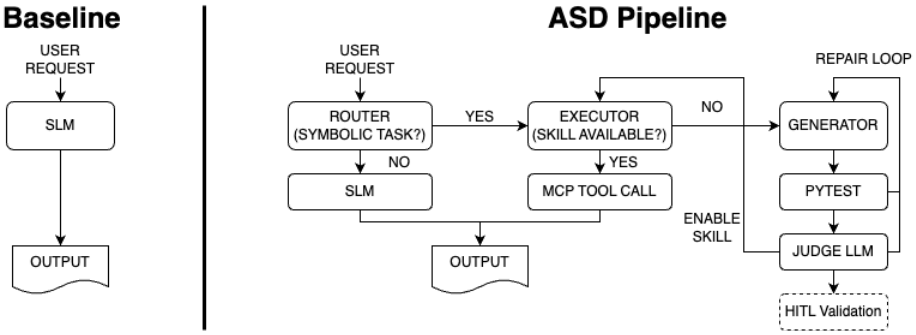

Fig. 1: Baseline vs ASD Pipeline. The baseline shows the user making a request to an SLM and receiving a natural-language response. On the right is the ASD Pipeline, where the user's request is analyzed by a first model that determines the nature of the request (symbolic task or not). If the task is not symbolic, the SLM responds; otherwise, the skills are checked, and if no adequate skill is found, one is created. The creation process involves generating Python code, tests, the LLM judge, and, in case of need, a human-in-the-loop (HITL).

3.1 Request Pipeline

ASD defines four roles, *router*, *executor*, *generator*, and *judge*. A request is handled in one of two ways. A router first decides whether the request has an extractable deterministic core, a verifiable input-to-output function computable

without semantic judgment, like a formula, a table lookup, a unit conversion, or arithmetic, even when it is wrapped in interpretive advice. If the answer is no, the execution model replies directly, as an ordinary assistant would. If yes, the request enters the skill branch.

The agent exposes the trusted library to the execution model as callable tools. Calling one is a reuse, a *hit*, and the computation runs as verified code. Calling none is a *miss* and triggers skill generation. Once generated and validated, the new skill is added to the library, and the execution model is queried again so it can call the skill just created, as shown in Figure 1. Either way, the model writes the natural-language answer, but on this branch, the reported number comes from inspectable, tested code rather than the model's own arithmetic.

3.2 Router: Extractable Deterministic Core

The router is a binary classifier over the request and the highest-risk component in the pipeline. A false negative misses the opportunity to compute deterministically, and a false positive triggers the generation of a task with no verifiable core. A fixed instruction defines the extractable deterministic core and asks for a two-line reply, a yes/no decision, and a one-line reason.

3.3 Skill Generation and Layered Validation

When the request is on the skill branch and no existing tool matches, the agent generates a new skill, a single pure, deterministic Python function with its own tests. The generation role returns a fixed set of sections in order, a `snake_case` name, the function signature, a description, a code block with the function, and a code block with `pytest` tests that import the function by name. The function must be pure and typed, with no I/O, no network, no globals, and no randomness. Type annotations matter because the tool's input schema is derived from them (Section 3.4). The description drives tool selection in the execution model, so it must state exactly what the function computes and from which inputs, specifically enough not to be chosen for a different quantity. The reply is parsed into a draft of name, signature, description, function code, and test code. A reply that does not follow the section format is not discarded. It is returned with a short reminder of the format, and the model is asked again up to a fixed number of times before a parse failure is raised. A draft must pass the validation pipeline below before it is trusted. Failures are repaired rather than rejected, and a single repair loop interleaves up to three layers.

The first layer (L1) writes the function and tests to a temporary directory and runs them with `pytest` in a separate subprocess under a timeout. It catches functions that do not run, that raise, or that disagree with the values the model itself asserts.

The second layer (L2) is an LLM judge, distinct from the generator. Given the task and the candidate's code and tests, the judge decides whether the function correctly and completely solves the task and whether its tests genuinely verify it. It replies with an approve-or-reject verdict and a one-line reason at temperature

zero, and an unparseable verdict counts as a rejection. L2 targets a gap L1 cannot see, a wrong formula paired with weak tests that agree with it. Neither layer compares the skill against an external oracle. L1 checks that code and tests agree, L2 adds an independent opinion on the formula, so together they certify internal consistency rather than ground-truth correctness.

When a layer fails and attempts remain, its failure is fed back to the generator role as repair feedback: the `pytest` output for L1, or the judge's reason for L2. The conversation is then rebuilt from the task, the previous draft, and that feedback, and the generation runs again. The loop runs up to a fixed maximum, and only a draft that passes the active layers is accepted. Every evaluated draft is kept with its code, tests, L1 output, L2 verdict, attempt count, and rejecting layer for analysis.

The third layer (L3) is optional and involves human review. A skill that passed the automatic layers can be flagged as human-validated in its metadata. The flag is recorded and surfaced in provenance rather than enforced, so a reviewed skill is reported as such, but review does not block use.

3.4 Library, Reuse, and Provenance

Accepted skills are stored on disk in a versioned library. Each skill is represented by three artifacts: the typed function that implements the tool, the generated tests, and a metadata record. The metadata records the skill status, validation outcomes, human-review flag, and reuse count.

An in-process server turns the library into callable tools. Each trusted skill becomes a tool whose name is the skill's name and whose description is the skill's description, with an input schema automatically derived from the function's type annotations.

On the skill branch, the agent builds the tool server from the current trusted library, lists the tools over an in-memory transport, and converts them to the execution model's format. Through native tool calling, the model decides which tool to invoke and whether to invoke it. A tool call is a reuse, so the verified function computes the result, the reuse count is incremented, and the model comments on the returned value. No tool call is a miss, which triggers generation and one re-entry into the tool loop so the new skill can be called. Selection is left entirely to the execution model.

Every answer in the skill branch has an inspectable provenance: which skill was applied, at which version, and the value it produced. The trust level is computed by code from the skill's metadata, not by the model, which distinguishes a system-validated skill from one that is also human-reviewed. The reported number is taken directly from the function's output, so the explanation reflects the computation that produced it rather than the model's own arithmetic.

4 Experimental Setup

This section presents the ASD evaluation: domain and benchmark, model configurations, conditions, metrics, and the run protocol. All experiments are run on a

single NVIDIA DGX Spark workstation (GB10 Grace Blackwell, 128 GB unified memory), which hosts models for all four roles simultaneously (see Section 3.1), with no external models or APIs in the loop.

4.1 Domain and Benchmark

We instantiate the domain-agnostic core on nutrition, a domain where many requests carry a deterministic core (formulas, table lookups, unit conversions, arithmetic) wrapped in interpretive advice, and where a wrong number has real consequences. The benchmark is a single labeled dataset of 38 tasks, 28 deterministic and 10 non-deterministic. The 28 deterministic tasks span 18 families: 10 literature rules (each with 2 cases) and 8 generic conversion and arithmetic tasks (each with 1 case).

At the core of the benchmark are 10 nutrition rules from the literature, each drawn from a peer-reviewed source and instantiated as a family of two cases. Two families come from the Mifflin–St Jeor equation [14], one as a resting energy expenditure rule for men and one as a BMR estimate for women. The remaining eight are body-fat percentage from BMI and age [4], estimated energy requirement (EER) [17], the Digestible Indispensable Amino Acid Score [5], the Body Roundness Index (BRI) [23], the carbohydrate lower bound of the Acceptable Macronutrient Distribution Range (AMDR) [16], A Body Shape Index (ABSI) [9], energy availability [15], and the triglyceride-glucose (TyG) index [21]. These rules span a difficulty gradient. Some are common knowledge for a capable model, such as BMR and TyG. In contrast, others are highly specific, such as the exact 2023 NASEM EER coefficients and the AMDR percentages, so a model that does not know them precisely will miss the value, which is itself informative.

Each request gives only the input data, never the formula or the conversion factor needed to compute the result. Each request specifies the rule or target quantity and provides the inputs, including only the qualifiers needed to fix the ground truth, such as sex, population, and activity level. The generator role must therefore know the formula and implement it correctly. Each deterministic task also carries a verified ground-truth value and a per-task absolute tolerance.

The eight generic conversion and arithmetic tasks, such as kilocalories to kilojoules, Atwater calories [1], and a multi-ingredient meal total, span the easy-to-hard range, so the results show where ASD begins to help. The ten non-deterministic tasks, such as whether a ketogenic diet is preferable, need semantic judgment and have no deterministic core. These form the router's negative class and exercise the direct, non-skilled answer path used by the baseline.

4.2 Model Configurations

We evaluate a matrix over two open-weight families, Gemma 4 [7] and Qwen 3.5 [22]. We chose them because they come from two independent providers, are released in a wide range of small sizes suitable for local deployment, and support the native tool calling the executor role requires. This lets us vary model size on

one axis at a time. All models are the F16 GGUF builds released by Unsloth [5], served locally with `llama.cpp` [6], one model per role. We set the sampling temperature per role: 0.0 for the router and the judge, 0.2 for generation, and 0.7 for execution. The remaining sampling parameters are left at their default values. Across the entire matrix, the judge is held fixed to a single 31B Gemma instance. Table 1 lists the full matrix. Each role can run with or without explicit chain-of-thought reasoning [27]. We treat reasoning as an ablative axis, comparing it on and off in the baseline and in the generator role, with everything else held constant.

Table 1: Model configuration matrix.

Config	Family	Router	Executor	Generator
C1	Gemma 4	E2B	E4B	12B
C2	Gemma 4	E2B	E4B	26B-A4B
C3	Gemma 4	E2B	E2B	12B
C4	Gemma 4	E2B	E2B	E4B
C5	Qwen 3.5	2B	4B	9B
C6	Qwen 3.5	0.8B	2B	4B
C7	Qwen 3.5	0.8B	0.8B	4B
C8	Qwen 3.5	0.8B	0.8B	2B

4.3 Setup

Each task runs under one or more setup, and a run can select any subset of them. The conditions isolate different costs and behaviors of the system.

- **baseline**: the pure-language-model behavior, answering every task directly with no router and no skills. It uses only the executor role.
- **create**: the skill branch with a cold library, generating one skill per family with L1 and L2 validation.
- **reuse**: all skills are preloaded, so every measured request is a hit. This measures the cost and correctness of reuse.

4.4 Metrics and Protocol Execution

The following metrics are derived from logs.

- **Correctness**: for a deterministic task, whether the answer contains a number within the task's tolerance of the ground-truth value.
- **Router quality**: the router is a binary classifier over the full benchmark, with the 28 deterministic tasks as the positive class and the 10 non-deterministic tasks as the negative class. We report its accuracy, precision, and recall.
- **Correctness at hit (c@hit)**: correctness restricted to cases where the model invokes a skill from the same family as the task. Requests with no skill invocation or with a skill from a different family are excluded. We compute this metric only on deterministic requests with a graded answer.
- **Tool selection (right, wrong, miss)**: *right*, a same-family skill is invoked; *wrong* a different-family skill is invoked; *miss* no skill is invoked.

[5] `https://huggingface.co/unsloth`

– **Latency (lat)**: the operating time for each request.
– **Repair and rejection**: the number of repair attempts driven by each layer (L1 test failures and L2 judge rejections) and the number of skills finally rejected by each layer. These quantify each layer's contribution during a run.

Each setup is repeated 5 times, and we report the mean and standard deviation. A request that fails, for example, on a generation timeout, is recorded as failed, and the run continues.

5 Results

This section centers on four questions. (i) How does a pure language model behave on the symbolic core? (ii) How reliably does the router separate deterministic from non-deterministic requests? (iii) What does ASD add when it creates and reuses verified skills, where does it fail, and does reasoning help? And (iv) what do the layered validation and the inspectable artifact contribute beyond accuracy? Tables 2a to 5 report all results, which we discuss in turn below.

The sole LLM setting is weak and unstable on the symbolic core. Answering directly, with no router and no skills, yields a weak and inconsistent execution model across families and sizes (Table 2a). Correctness ranges from 29.3% at the smallest executor to 72.9% at the 4B Qwen one, with smaller models at the low end and most configurations well below what a high-stakes setting would require. Chain-of-thought reasoning does not rescue the baseline (Table 2b): the strongest configuration stays flat (72.9% to 73.6%), others drop (39.3% to 25.0% for the Gemma-E4B executor, 47.1% to 38.6% for the 2B Qwen one), and latency roughly doubles. We take the no-reasoning baseline as the reference.

The router is accurate. The router is accurate and conservative by design (Table 2c). Accuracy is 83-95% and precision is 100% across all configurations; it never routes a non-deterministic request to the skill branch. Its only errors are false negatives, with recall from 77 to 93%, where a genuinely deterministic task is sent to the direct path and loses verification.

c	corr%	lat (s)
C1,2	39.3 ± 4.4	5.4 ± 0.4
C3,4	57.9 ± 3.0	3.4 ± 0.1
C5	72.9 ± 5.4	10.8 ± 5.3
C6	47.1 ± 1.6	5.9 ± 2.0
C7,8	29.3 ± 3.0	2.7 ± 0.8

(a)

c	corr%	lat (s)
C1,2	25.0 ± 4.4	10.5 ± 0.4
C3,4	57.9 ± 3.9	7.5 ± 0.2
C5	73.6 ± 3.2	22.5 ± 8.0
C6	38.6 ± 3.0	16.3 ± 2.5
C7,8	23.6 ± 2.0	12.5 ± 2.6

(b)

config	acc%	prec%	rec%
C1,2,3,4	95	100	93
C5	89	100	85
C6,7,8	83	100	77

(c)

Table 2: (a) Baseline no reasoning, (b) Baseline reasoning, and (c) Router quality.

ASD improves correctness. Generating one verified skill per family on a cold library raises correctness over the baseline across the board (Table 3a). We compare the baseline and the per-task results using an exact McNemar [12] test, taking the majority across the 5 repeats per configuration, without pooling. Create wins in every configuration compared, significantly in three (C1, C6, C4; p = .006, .031, .004) ranges, and the rest move the same way, with too few discordant pairs to reach significance. When the right skill is invoked, c@hit ranges from 84.9 to 100%, so the generated code is rarely the problem. What limits correctness is the model's ability to call the right skill. The gap between overall correctness (54 to 78%) and c@hit is therefore almost entirely due to the selection, the model failing to call the freshly created tool, rather than the code being wrong. The exception is C8, whose 2B generator is below the size needed to write working code: it yields an accepted skill in only 2.9% of creation attempts and collapses to 23.6% of creation attempts. A generator this small does not benefit from ASD because it cannot reliably write the skill in the first place, a limit of the generator rather than of the approach. Even the capable generators do not pass validation on every attempt, with accepted-skill rates around 60-80%.

c	corr%	c@hit	miss%	created%	lat (s)
C1	70.0 ± 3.2	96.1 ± 2.2	22.9 ± 7.8	71.4 ± 5.1	97 ± 8.7
C2	73.6 ± 4.1	93.6 ± 4	3.5 ± 3.7	77.9 ± 3.0	82.3 ± 10.7
C3	69.3 ± 3.2	94.9 ± 5.1	25.2 ± 5.5	69.3 ± 3.2	98.6 ± 13.4
C5	77.9 ± 1.6	95.7 ± 3	4.2 ± 2.9	81.4 ± 3.0	85.8 ± 12.1
C6	67.1 ± 3.0	98.8 ± 2.8	23.1 ± 5.9	59.3 ± 4.8	64.9 ± 8.5
C4	65.0 ± 5.3	89.1 ± 8.5	32.3 ± 5.2	62.9 ± 6	73.3 ± 6.3
C7	54.3 ± 4.7	84.9 ± 2.6	27.6 ± 4.8	68.6 ± 3.9	64.5 ± 6.6
C8	23.6 ± 3.2	100.0 ± 0.0	96.3 ± 3.8	2.9 ± 3.0	21.9 ± 3.4

(a)

c	corr%	c@hit	miss%	created%	lat (s)
C1	63.6 ± 3.0	96.8 ± 2.9	2.0 ± 2.7	65.7 ± 4.1	100.1 ± 10.1
C2	75.7 ± 1.6	100 ± 0.0	0.0 ± 0.0	75.7 ± 1.6	90.1 ± 16.6
C3	65.7 ± 3.2	100.0 ± 0.0	1.0 ± 2.2	65.7 ± 3.2	90.5 ± 7.1
C5	62.1 ± 7.4	94.7 ± 3.7	3.3 ± 3.1	66.4 ± 10	115.2 ± 22.4
C6	69.3 ± 4.1	97.9 ± 2.9	17.3 ± 5.9	67.9 ± 4.4	103.4 ± 17.2
C4	74.3 ± 1.6	97.2 ± 4.1	24.6 ± 3.1	72.1 ± 4.7	84.9 ± 8.8
C7	55.7 ± 4.8	90.7 ± 6.8	42 ± 7.1	64.3 ± 5.1	99.4 ± 12.1
C8	2.9 ± 3	–	100 ± 0	0 ± 0.0	130.0 ± 3.9

(b)

Table 3: ASD Pipeline: (a) Create no reasoning, (b) Create reasoning

Reasoning does not help creation. Repeating create with reasoning on the generation role gives no consistent benefit (Table 3b). The direction is split evenly across configurations, and the paired McNemar test shows no significant difference. The capable 9B Qwen generator drops from 77.9% to 62.1%, the smallest generator falls from 23.6% to 2.9%, and latency increases across the board. We keep no-reasoning as the main configuration.

Reuse: given the right skills, ASD is near-perfect. With the verified library pre-built and every request a hit, ASD reaches the ceiling (Table 4). c@hit is 100% in every configuration, and overall correctness is 94-100% for every executor but the smallest. Partitioning the routed tasks into right, wrong, and miss, the losses are entirely selection errors, concentrated in the smallest executors (C7 and C8, 12% wrong tool and 13% no tool), while the other executors select 94-100% of the time correctly. This is the paper's central finding, made precise: given the right skills, the dominant failure mode is tool selection, and it worsens as the executor shrinks. Reuse is also the fastest path: a hit answers in about 1 to 2 seconds, below the baseline, and far below the tens of seconds

spent on generation. That cost is paid once per family and then amortized across subsequent requests, so the expensive step occurs only at creation.

c	right%	wrong%	miss%	corr%	c@hit%	lat(s)
C1,2	98	2	0	98	100	1.4 ± 0.0
C3,4	100	0	0	100	100	0.9 ± 0.0
C5	96	4	0	98	100	1.8 ± 0.1
C6	94	6	0	94	100	1.3 ± 0.1
C7,8	74	12	13	80	100	1.1 ± 0.6

Table 4: Reuse Task - No Reasoning

Validation layers contribution. Table 5 shows the effort the pipeline spends during creation. L1, the deterministic tests, is the workhorse: L1-triggered repairs are common and grow as the generator weakens, from 28% of attempts for the Gemma-12B generator to 84-97% for the small Qwen generators, and almost all final rejections are L1's. The independent judge L2 adds a distinct signal: on the Gemma generators, it triggers a repair in 19 to 30% of attempts and a final rejection in 1 to 9%, catching consistency gaps that L1 had passed, such as a wrong formula paired with agreeing tests, while on the Qwen generators, it rarely fires. Most repairs are recoverable rather than fatal: C2 needs a L1 repair in 56% of attempts but is finally rejected in only 3%, so the loop fixes rather than discards. config8 is again the floor, with 97% of attempts requiring a L1 repair and 94% finally rejected, so below a certain generator size, the layers correctly refuse broken code rather than letting it through. Read together with the *created* column of Table 3a, these rates explain the accepted-skill percentages: the stronger generators recover most drafts through the repair loop and end around 60-80%. At the same time, C8's 94% rejection leaves almost nothing accepted, the 2.9% reported there.

c	attempts	L1 rep. (%)	L2 rep. (%)	rej. L1 (%)	rej. L2 (%)
C1	1.7 ± 0.1	28 ± 11	25 ± 5	15 ± 8	8 ± 4
C2	1.8 ± 0.1	56 ± 6	19 ± 5	3 ± 4	1 ± 2
C3	1.7 ± 0.1	28 ± 4	28 ± 5	16 ± 5	9 ± 3
C4	1.9 ± 0.1	39 ± 6	30 ± 4	25 ± 4	8 ± 6
C5	1.5 ± 0.1	39 ± 8	3 ± 2	4 ± 3	0 ± 0
C6	2.3 ± 0.1	93 ± 4	4 ± 4	22 ± 5	1 ± 2
C7	2.1 ± 0.1	84 ± 4	4 ± 4	10 ± 5	2 ± 3
C8	2.9 ± 0.1	97 ± 4	2 ± 3	94 ± 6	2 ± 3

Table 5: Repair Loop Statistics

Cost in time: creation is a one-time expense, reuse is light. Creation is expensive, about 65 to 130 seconds per request (Tables 3a and 3b), since it runs generation, the test subprocess, the judge, and the repair loop. This cost is paid only during creation. During reuse, ASD runs only the router and the executor over an already validated tool, answering in about 1 to 2 seconds. We

therefore do not claim that ASD is cheaper than the baseline for every request. Its value is verifiability: the expensive step is front-loaded, while repeated use remains lightweight.

Human inspection. Beyond accuracy, the create branch leaves behind an artifact: each skill carries its own tests and a provenance record of which rule, inputs, value, and validation produced the answer, as well as whether a human reviewed it. Unlike a black-box answer, it can be read, audited, and corrected by a domain expert. This matters where the pipeline is imperfect: a skill with a subtly wrong formula, which L1 and L2 certify only for internal consistency, can be fixed once, and because skills are versioned and reused, the correction then serves every future request. A single human review thus becomes permanent correctness, the safety and audit advantage that justifies the one-time creation cost in high-stakes domains.

6 Conclusion

This study presented ASD, a neuro-symbolic agent that turns the deterministic core of a request into a verified, reusable tool. A router decides whether such a core exists. If so, ASD generates a skill as a pure typed function exposed as an MCP tool, validates it with deterministic tests and an independent LLM judge in a repair loop, and stores it in a versioned library. The computation then runs as inspectable, tested code, and the language model only reports the result. The whole pipeline runs locally on open-weight models on a single workstation.

Across a nutrition benchmark and two open-weight families, the results support three claims. Moving the symbolic core to verified code keeps smaller execution models correct, down to the point where the generator is too small to write a usable skill. Given the right skill, ASD is near-perfect, and the only residual error is in tool selection, which grows as the execution model shrinks. The validation layers do real work: the deterministic tests are the workhorse, and the independent judge catches consistency gaps they miss. Beyond accuracy, each skill is an inspectable, editable, reusable artifact, so a single human review becomes the definitive correctness for every subsequent request.

Several directions follow. Since the main residual error is tool selection, retrieval-based skill indexing, which exposes only a few relevant skills per request, is the natural next step and should also help as the library grows. Our validation checks internal consistency, not ground-truth correctness, so a skill with a subtly incorrect formula can still be admitted. A further open question is whether generator and judge, being trained on partly overlapping corpora, may share blind spots, so that a poorly represented rule could be implemented incorrectly and approved for the same reason. How often this happens and how the model behaves when a skill returns a wrong value are open robustness questions. Finally, the smallest generators mark a floor below which no usable skill is produced, and a stronger generator may widen the range of small execution models that ASD can serve.

Acknowledgments. This work was supported by the HES-SO RCSO ISNet Hybrid Ai foR RelIable perSonalized cOachiNg (HARRISON) grant, Swiss AI Initiative (a0138), HES-SO R&I Mind the Gap: Adapting Higher Education to the Age of AI (MTG, 140012/RI-PSTRAT25-06).

References

1. Atwater, W.O., Benedict, F.G.: An experimental inquiry regarding the nutritive value of alcohol. No. 233, US Government Printing Office (1902)
2. Cai, T., Wang, X., Ma, T., Chen, X., Zhou, D.: Large language models as tool makers. In: International Conference on Learning Representations. vol. 2024, pp. 54067–54089 (2024)
3. Crocetti, V., Dragoni, A.F.: Bridging large language models and logic programming. In: 2025 IEEE International Conference on Metrology for eXtended Reality, Artificial Intelligence and Neural Engineering (MetroXRAINE). pp. 1066–1071 (2025). https://doi.org/10.1109/MetroXRAINE66377.2025.11340308
4. Deurenberg, P., Weststrate, J.A., Seidell, J.C.: Body mass index as a measure of body fatness: age- and sex-specific prediction formulas. British Journal of Nutrition **65**(2), 105–114 (1991). https://doi.org/10.1079/BJN19910073
5. Food and Agriculture Organization of the United Nations: Dietary protein quality evaluation in human nutrition. Tech. Rep. 92, Food and Agriculture Organization of the United Nations, Rome (2013), https://openknowledge.fao.org/handle/20.500.14283/i3124e, report of an FAO Expert Consultation
6. Gerganov, G., llama.cpp contributors: llama.cpp: LLM inference in C/C++ (2023), https://github.com/ggml-org/llama.cpp, accessed: 2026-06-19
7. Google DeepMind: Gemma 4 model card. https://ai.google.dev/gemma/docs/core/model_card_4 (2026), accessed: 2026-05-28
8. Körner, P., Leuschel, M., Barbosa, J., Costa, V.S., Dahl, V., Hermenegildo, M.V., Morales, J.F., Wielemaker, J., Diaz, D., Abreu, S., et al.: Fifty years of prolog and beyond. Theory and Practice of Logic Programming **22**(6), 776–858 (2022). https://doi.org/10.1017/S1471068422000102
9. Krakauer, N.Y., Krakauer, J.C.: A new body shape index predicts mortality hazard independently of body mass index. PLoS ONE **7**(7), e39504 (2012). https://doi.org/10.1371/journal.pone.0039504
10. Liu, Z., Hoang, T., Zhang, J., Zhu, M., Lan, T., Kokane, S., Tan, J., Yao, W., Liu, Z., Feng, Y., et al.: Apigen: Automated pipeline for generating verifiable and diverse function-calling datasets. Advances in Neural Information Processing Systems **37**, 54463–54482 (2024)
11. Lyu, Q., Havaldar, S., Stein, A., Zhang, L., Rao, D., Wong, E., Apidianaki, M., Callison-Burch, C.: Faithful chain-of-thought reasoning. In: Park, J.C., Arase, Y., Hu, B., Lu, W., Wijaya, D., Purwarianti, A., Krisnadhi, A.A. (eds.) Proceedings of the 13th International Joint Conference on Natural Language Processing and the 3rd Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics (Volume 1: Long Papers). pp. 305–329. Association for Computational Linguistics, Nusa Dua, Bali (Nov 2023). https://doi.org/10.18653/v1/2023.ijcnlp-main.20, https://aclanthology.org/2023.ijcnlp-main.20/
12. McNemar, Q.: Note on the sampling error of the difference between correlated proportions or percentages. Psychometrika **12**(2), 153–157 (1947)

13. Mekala, D., Weston, J.E., Lanchantin, J., Raileanu, R., Lomeli, M., Shang, J., Dwivedi-Yu, J.: Toolverifier: Generalization to new tools via self-verification. In: Findings of the Association for Computational Linguistics: EMNLP 2024. pp. 5026–5041 (2024)
14. Mifflin, M., St Jeor, S., Hill, L., Scott, B., Daugherty, S., Koh, Y.: A new predictive equation for resting energy expenditure in healthy individuals. The American Journal of Clinical Nutrition **51**(2), 241–247 (1990). https://doi.org/https://doi.org/10.1093/ajcn/51.2.241
15. Mountjoy, M., Ackerman, K.E., Bailey, D.M., Burke, L.M., Constantini, N., Hackney, A.C., Heikura, I.A., Melin, A., Pensgaard, A.M., Stellingwerff, T., Sundgot-Borgen, J.K., Torstveit, M.K., Jacobsen, A.U., Verhagen, E., Budgett, R., Engebretsen, L., Erdener, U., Erdmann, L., Gerrard, D., Geyer, H., Jorstad, H.T., Khan, K.M., Ljungqvist, A., Meyer, N.L., Moseley, B., Mountjoy, M., Pitsiladis, Y., Schamasch, P., Soligard, T., Webborn, N.: 2023 international olympic committee's (IOC) consensus statement on relative energy deficiency in sport (REDs). British Journal of Sports Medicine **57**(17), 1073–1097 (2023). https://doi.org/10.1136/bjsports-2023-106994
16. National Academies of Sciences, Engineering, and Medicine: Dietary Reference Intakes for Energy, Carbohydrate, Fiber, Fat, Fatty Acids, Cholesterol, Protein, and Amino Acids. The National Academies Press, Washington, DC (2005). https://doi.org/10.17226/10490, https://doi.org/10.17226/10490
17. National Academies of Sciences, Engineering, and Medicine: Dietary Reference Intakes for Energy. The National Academies Press, Washington, DC (2023). https://doi.org/10.17226/26818
18. Pacioni, E., Coman, A.C., Calvaresi, D., Manzo, G., Schumacher, M.I.: Agent-based hybrid ai models and technologies: A systematic literature review. IEEE Access **14**, 21148–21166 (2026). https://doi.org/10.1109/ACCESS.2026.3661027
19. Qu, C., Dai, S., Wei, X., Cai, H., Wang, S., Yin, D., Xu, J., Wen, J.R.: Tool learning with large language models: A survey. Frontiers of Computer Science **19**(8), 198343 (2025)
20. Shi, Z., Wang, Y., Yan, L., Ren, P., Wang, S., Yin, D., Ren, Z.: Retrieval models aren't tool-savvy: Benchmarking tool retrieval for large language models. In: Findings of the Association for Computational Linguistics: ACL 2025. pp. 24497–24524 (2025)
21. Simental-Mendía, L.E., Rodríguez-Morán, M., Guerrero-Romero, F.: The product of fasting glucose and triglycerides as surrogate for identifying insulin resistance in apparently healthy subjects. Metabolic Syndrome and Related Disorders **6**(4), 299–304 (2008). https://doi.org/10.1089/met.2008.0034
22. Team, Q.: Qwen3 technical report (2025), https://arxiv.org/abs/2505.09388
23. Thomas, D.M., Bredlau, C., Bosy-Westphal, A., Mueller, M., Shen, W., Gallagher, D., Maeda, Y., McDougall, A., Peterson, C.M., Ravussin, E., Heymsfield, S.B.: Relationships between body roundness with body fat and visceral adipose tissue emerging from a new geometrical model. Obesity **21**(11), 2264–2271 (2013). https://doi.org/https://doi.org/10.1002/oby.20408
24. Van Rossum, G., Drake, F.L., et al.: Python reference manual, vol. 111. Centrum voor Wiskunde en Informatica Amsterdam (1995)
25. Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., Anandkumar, A.: Voyager: An open-ended embodied agent with large language models. arXiv preprint arXiv:2305.16291 (2023)

26. Wang, Z.Z., Neubig, G., Fried, D.: Trove: inducing verifiable and efficient toolboxes for solving programmatic tasks. In: Proceedings of the 41st International Conference on Machine Learning. ICML'24, JMLR.org (2024)
27. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., et al.: Chain-of-thought prompting elicits reasoning in large language models. Advances in neural information processing systems **35**, 24824–24837 (2022)
28. Yuan, L., Chen, Y., Wang, X., Fung, Y., Peng, H., Ji, H.: Craft: Customizing llms by creating and retrieving from specialized toolsets. In: International Conference on Learning Representations. vol. 2024, pp. 40097–40125 (2024)
29. Zheng, L., Chiang, W.L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al.: Judging llm-as-a-judge with mt-bench and chatbot arena. Advances in neural information processing systems **36**, 46595–46623 (2023)