arXiv:2605.17172v1 [cs.LG] 16 May 2026

Personal AI, On Personal Devices

Jon Saad-Falcon*[1] · Avanika Narayan*[1] · Robby Manihani[1]
Tanvir Bhathal[1] · Herumb Shandilya[1] · Hakki Orhun Akengin[1]
Gabriel Bo[1] · Andrew Park[1] · Matthew Hart[1] · Caia Costello[2]
Chuan Li[2] · Christopher Ré[1] · Azalia Mirhoseini[1]

* Equal contribution. [1] Stanford University [2] Lambda Labs

Personal AI stacks, like OpenClaw and Hermes Agent, are becoming central to daily work, yet they route nearly every query (often over sensitive local data) to cloud-hosted frontier models. Replacing frontier models with local models inside existing stacks does not work: swapping `Claude Opus 4.6` for Qwen3.5-9B drops accuracy by 25–39 pp across personal AI tasks like PinchBench and GAIA. Existing stacks bundle agentic prompts, tool descriptions, memory configuration, and runtime settings around a specific cloud model. Only the prompts can be tuned, and state-of-the-art prompt optimizers close just 5 pp of the local–cloud gap on their own. This motivates a decomposed personal AI stack: one that exposes individual primitives which can be optimized individually or jointly to close the local–cloud gap. We present OpenJarvis, an architecture that represents a personal AI system as a typed *spec* over five primitives: Intelligence, Engine, Agents, Tools & Memory, and Learning. Each primitive is an independently editable field, making the stack end-to-end optimizable and measurable against accuracy, cost, and latency. Towards closing the local–cloud gap without surrendering local-model properties, OpenJarvis introduces *LLM-guided spec search*, a local–cloud collaboration in which frontier cloud models propose edits across the *spec* at search time, only non-regressing edits are accepted, and the resulting spec runs entirely on-device at inference time. With LLM-guided spec search, on-device *specs* match or exceed cloud accuracy on 4 of 8 benchmarks and land within 3.2 pp of the best cloud baseline on average. They also reduce marginal API cost by $\sim$800$\times$ and end-to-end latency by 4$\times$.

Code: https://github.com/openjarvis/openjarvis
Website: https://open-jarvis.github.io/OpenJarvis/

1 Introduction

Personal AI stacks, like OpenClaw [80] and Hermes Agent [60], are now central to daily writing, research, coding, and scheduling. Yet today's personal AI is rarely private or local. Most systems route each query to a cloud-hosted frontier model, often including sensitive personal data. This design demands thousands of dollars per year in recurring API and subscription spend [96, 78]. It exposes private data to third-party servers, requires network connectivity to function, and leaves users without ownership of the models they depend on. It also consumes orders of magnitude more energy per token than local execution [75]. Meanwhile, consumer accelerators [5, 70, 34] now run 1B–128B open-weight models with FP8 quantization [21], and open-weight families such as Qwen3.5 [71] and Gemma4 [28] trail frontier cloud models by 6–12 months on personal AI tasks rather than years [13, 75]. On-device models nonetheless remain confined to trivial tasks like tone adjustment and text completion [6].

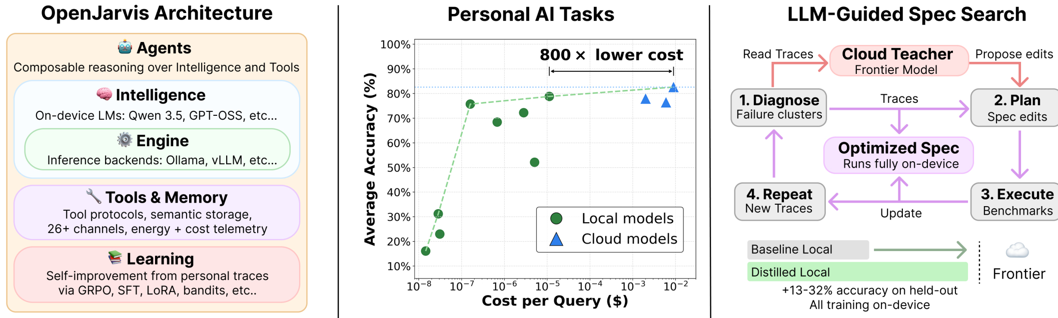

Figure 1: **Overview of OPENJARVIS. (Left)** Five composable primitives (Intelligence, Engine, Agents, Tools & Memory, Learning) are composed through a declarative *spec* that can be shared, evaluated, and optimized end-to-end (Section 3.1). **(Middle)** Joint accuracy-efficiency evaluation across the evaluated local *specs* (green) and 3 cloud baselines (blue) reveals that on-device configurations approach within 3.2 pp of the best cloud model at roughly 800× lower marginal API cost per query (Section 4.3). **(Right)** LLM-guided spec search uses frontier cloud models as reflective proposers: the teacher diagnoses failures, proposes edits across the full *spec*, and a held-out gate accepts only non-regressing improvements, closing the remaining local–cloud accuracy gap by 13–32 pp with student training on-device (Section 4.4).

In this work we ask: *can the core of a personal AI stack (i.e., model inference, agent execution, memory, and learning) run on-device while remaining competitive with cloud-only stacks?* We study this question across eight personal AI benchmarks spanning writing, research, coding, and scheduling, including PinchBench [40] and GAIA [50] (Section 4.1).

As a first attempt, we ask whether one can simply swap the cloud model for a local one inside an existing stack. Replacing `Claude Opus 4.6` with `Qwen3.5-9B`, while keeping the rest of OpenClaw or Hermes Agent fixed, drops accuracy by 25–39 percentage points across PinchBench and GAIA (Section 4.2, Table 1). We identify two reasons for the local-cloud performance gap:

- *Substitution breaks monolithic stacks.* Agent prompts, tool descriptions, memory configuration, and runtime settings are fused into the framework, co-designed for the intended cloud model. Substituting a local model breaks all of these at once (Section 4.2).
- *Single-primitive optimization plateaus.* Of the bundled components, only prompts can be tuned without source-level changes. The rest are baked into the framework. Applying state-of-the-art prompt optimizers (GEPA [1] and DSPy [39]) to the swapped-in local model, while holding everything else fixed, closes only 5 pp of the cloud–local gap on their own. Optimizing one primitive at a time cannot deliver the coordinated changes across model, prompts, tools, and runtime that misalignment demands.

Motivated by these limitations, we introduce OPENJARVIS, an architecture that makes personal AI stacks end-to-end optimizable. OPENJARVIS has two core components. First, the *spec*: a typed configuration object that decomposes the personal AI system as five editable primitives: *Intelligence*, the language model architecture and its weights; *Engine*, the inference runtime; *Agents*, the reasoning loop; *Tools & Memory*, data integration and persistent storage; and *Learning*, the optimizer that updates the system from traces (Section 3.1). The spec exposes every component of the stack as a degree of freedom in one optimizable object. Second, LLM-guided spec search: a local–cloud search algorithm that jointly optimizes all four editable primitives of the spec: Intelligence, Engine, Agents, and Tools & Memory. LLM-guided spec search runs entirely on-device at inference time but uses frontier cloud models at search time to improve the local spec as much as possible. This gives us the best of both worlds: cloud-model capability transferred into a spec that runs locally. At each search step, a frontier model reads traces from the current spec and proposes coordinated edits (i.e., rewrite a tool description, adjust a runtime setting) across the four primitives. Each

edit is accepted only if it improves performance on a held-out evaluation, and accepted edits become the new spec for the next step.

Across eight benchmarks for personal AI, optimized specs match or exceed cloud accuracy on four, land within 3.2 pp of the best cloud baseline on average, and reduce marginal API cost by $\sim 800\times$ and end-to-end latency by $4\times$ (Section 4.4). LLM-guided spec search closes 13–32 pp of the cloud–local gap at 7–11× lower optimization cost than single-primitive baselines. We perform a detailed analysis of LLM-guided spec search's search space. Our analysis highlights three axes that govern the local–cloud gap:

(a) **Editable surface.** *How does expanding the move space from prompts alone to all four editable primitives affect gap closure?* In Figure 8 and Appendix C.5, we show that expanding the editable set adds 5.5–16.5 pp.

(b) **Proposer choice.** *Does diagnosing failures before proposing edits matter, or would evolutionary spec search at the same move space suffice?* In Figure 8 and Appendix C.7, we show that the diagnose-and-propose loop in LLM-guided spec search adds 10.0 pp on average over evolutionary spec search at the same four-primitive move space.

(c) **Search budget.** *How does optimization cost trade off against gap closure?* In Figure 7, we show that LLM-guided spec search matches single-primitive baselines at 7–11× lower optimization cost.

To summarize, our main contributions are as follows:

- Propose OpenJarvis, an architecture that decomposes a personal AI system into five typed primitives composed into an editable spec, making the stack end-to-end optimizable and measurable on accuracy, cost, and latency (Section 3).
- Propose LLM-guided spec search, a local–cloud collaboration that uses frontier models to propose spec edits at search time and runs the resulting spec entirely on-device at inference time, closing 13–32 pp of the cloud–local gap at 7–11× lower optimization cost than single-primitive baselines (Section 4).
- Conduct an in-depth analysis across eight personal AI benchmarks, showing that on-device specs are Pareto-optimal on marginal API cost and latency with $\sim$800× lower marginal API cost and 4× lower end-to-end latency, and isolating the contributions of the proposer and the move space to the overall gain (Section 4.3).

2 Related Work

Unifying abstractions have historically accelerated ML progress. Imperative deep-learning frameworks [67] and type-based declarative ML toolboxes [55] unified model construction, while data systems for monitoring deployed ML products [72] and LM pipeline compilers [39] unified iterative improvement. The personal AI ecosystem is undergoing a similar fragmentation-to-unification transition. Existing tools each formalize a subset of the five primitives (Intelligence, Engine, Agents, Tools & Memory, and Learning) while hardcoding or omitting the rest.

Agents and Tools: personal AI stacks. Existing personal AI stacks formalize Agents and Tools as configurable layers but tie Intelligence, Engine, and Learning to specific cloud models, so substituting a different Intelligence degrades the full pipeline (Section 4.2). This pattern holds across open frameworks such as OpenClaw [80], Hermes Agent [60], LangChain [10], CrewAI [15], Google ADK [23], OpenAI Symphony [64], Qwen-Agent [2], and related systems [68, 19, 57, 35, 84, 93, 51], whose Agent prompts, Tool descriptions, Memory configuration, and Engine/runtime settings are tied to specific model and engine choices. Closed systems (Apple Intelligence [6], Gemini Nano [26]) additionally hardcode the Engine to proprietary runtimes and expose no Learning interface.

Engine and Intelligence: local inference tools. Local inference tools formalize Engine and Intelligence as configurable layers but provide no Agents, Tools, or Learning, leaving

the upper half of the stack unaddressed. This includes serving runtimes (Ollama [63], llama.cpp [21], LM Studio [47], LocalAI [56], gemma.cpp [22]), Engine-side performance optimizations such as quantization (GPTQ [20], AWQ [43]), speculative decoding [42, 11], and hardware-aware serving (MLC-LLM [53], ExecuTorch [49], vLLM [41], SGLang [95]), and Intelligence-side on-device architectures (MobileLLM [46], Gemma 3n [24], Liquid Nanos [45], Nemotron-Flash [61]) co-designed for efficient execution. OPENJARVIS's Engine abstracts over these backends with built-in energy and cost telemetry, extending learned query routing [85] to local-vs-cloud decisions.

Learning: optimization methods. Optimization methods formalize Learning but address one primitive at a time, falling into two families. Weight optimizers update Intelligence: classical knowledge distillation [30] (see [90] for a survey) trains students to match teacher soft targets, MiniLLM [29] and DeepSeek-R1 [16] extend this to LLMs, and PEFT methods (LoRA [31], QLoRA [17], GRPO [77]) make on-device weight updates feasible even on microcontrollers [44], with cross-platform mobile LoRA now practical [83, 12]. Prompt and agent optimizers update Agents (and, in some variants, Tools): DSPy [39] and ACE [94] compile or evolve agent prompts, and GEPA [1] uses LLM-based reflection over trajectories with Pareto-efficient evolutionary machinery to preserve and merge complementary candidates. Inference-time collaboration (Minions [58], Advisor Models [7]) decomposes tasks across local and cloud Intelligence but does not persistently update any primitive, and OpenClaw-RL [88] is the closest prior system to continuous on-device learning, training an RL policy from live user interactions but updating only Intelligence weights in a cloud-hosted loop. Prior work has also observed that combining weight and prompt optimization can outperform either alone [79]. LLM-guided spec search (Section 3.3) extends this direction to four editable primitives (Intelligence, Engine, Agents, and Tools & Memory) with Learning as the optimizer slot, making it complementary to single-primitive methods rather than a replacement for them.

Joint evaluation. On the evaluation side, tools exist for individual efficiency dimensions (Zeus [52], AI Energy Score [32], MLCommons [54]) but measure a single axis of a single primitive. Standard agent benchmarks (GAIA [50], SWE-bench [37], Terminal-Bench [48]) report accuracy alone and assume unlimited cloud compute. Because the spec provides a common representation for complete configurations, we are not aware of a prior framework that jointly evaluates accuracy, energy, latency, power, and cost across the full five-primitive composition (Section 4).

Summary. Each prior effort formalizes a subset of the structure: personal AI frameworks address Agents and Tools, local inference tools address Engine and Intelligence, optimization methods address Learning one primitive at a time, and evaluation tools measure one axis of one primitive. No existing framework formalizes the full five-primitive composition or exposes it as a single optimizable object. This gap motivates the spec abstraction and LLM-guided spec search we introduce in Section 3.

3 Methods

We present OPENJARVIS in three parts: the spec abstraction (Section 3.1), joint accuracy-efficiency evaluation (Section 3.2), and LLM-guided spec search (Section 3.3). The core idea is that every local model needs the surrounding agent system reconfigured around it; the spec makes that reconfiguration explicit and optimizable. Section 2 situates these design choices in prior work, and Appendix A provides expanded method details.

3.1 Primitives and the Spec Abstraction

OPENJARVIS decomposes a personal AI stack into five primitives, each with a typed interface and registry (Figure 1, left). *Intelligence* specifies the language model and generation parameters. *Engine* specifies the runtime, hardware execution path, batching, quantization, and cache settings. *Agents* specify the reasoning loop, prompts, and tool-use policy. *Tools & Memory* specify external interfaces, retrieval, and persistent user state. *Learning* specifies

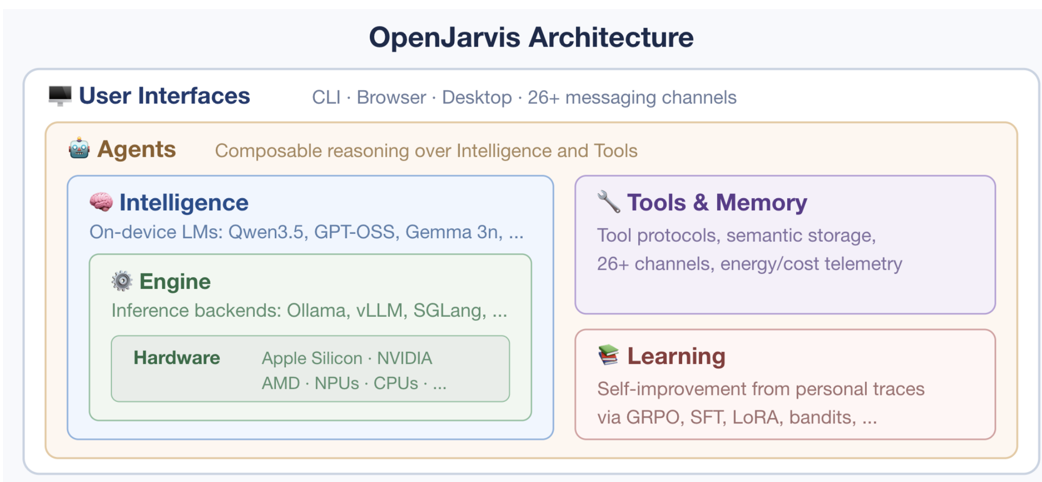

Figure 2: **OpenJarvis architecture.** Five composable primitives decouple model selection (*Intelligence*), inference runtime (*Engine*), agent logic (*Agents*), data integration (*Tools & Memory*), and on-device learning (*Learning*) into independently swappable layers. A *spec* (Section 3.1) composes all five into a declarative configuration that can be shared, evaluated, and optimized end-to-end.

the optimizer that updates the spec from traces, including weight-only optimizers such as LoRA [31], prompt optimizers such as DSPy [39], and LLM-guided spec search (Section 3.3). Implementation details, supported runtimes, connectors, memory systems, and optimizer variants are in Appendix A.1.

Five-primitive architecture. Each primitive is an independent degree of freedom that prior systems optimize in isolation. The decomposition is designed to separate choices that are independently configurable in deployed systems. For example, Intelligence and Engine separate model selection (`Qwen3.5-9B` vs. `Qwen3.5-122B`) from runtime selection (`vLLM` vs. `Ollama`). Agents and Tools separate the reasoning loop (ReAct vs. CodeAct) from the interfaces it invokes (filesystem, web search, MCP servers). Learning is structurally different from the other four primitives: it is the primitive we use to optimize all the others, making edits to Intelligence weights, Agent prompts/logic, Tools & Memory selection and descriptions, and Engine runtime specifications. This primitive lets the same framework describe systems that share Intelligence and Agent configurations but differ only in Learning policy, such as OpenClaw-RL [88]. Section 4.4 tests whether these distinctions between primitives matter for constructing and optimizing local AI.

The spec abstraction. The five primitives are composed through the *spec*: a typed configuration object on which a single optimization algorithm operates. Figure 3 shows the spec as a structured declaration alongside the optimization signature; Figure 9 (Appendix A.1) shows the corresponding TOML serialization. A spec can be versioned, shared, evaluated on standardized benchmarks, and optimized end-to-end (Section 3.3).

3.2 Evaluation Metrics

We evaluate complete *specs* rather than isolated models. For each spec, an instrumented wrapper records five quantities per query: accuracy, energy, latency, power, and dollar cost. Accuracy is exact match or LLM-judge win rate, depending on the benchmark. Energy is measured through vendor APIs for local hardware and estimated for cloud baselines following prior work [75]. Latency is end-to-end wall-clock time, and power is energy divided by latency. Dollar cost is per-token API pricing and tool fees; local model inference is reported as $0 marginal API cost, with hardware and electricity reported separately. Together, the

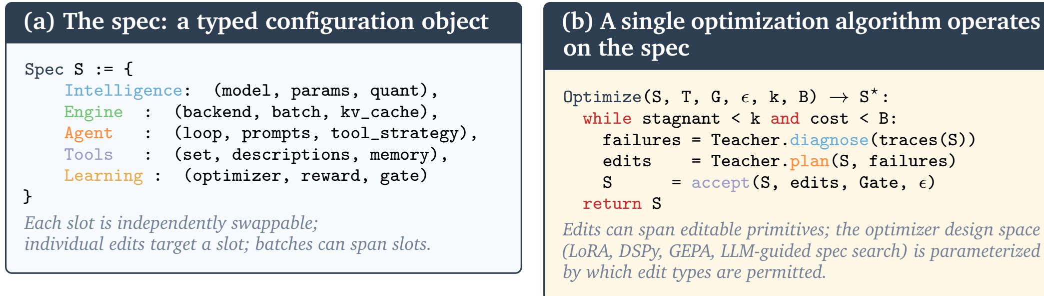

Figure 3: **The spec abstraction.** **(a)** A spec is a typed configuration object with five primitives: Intelligence, Engine, Agents, Tools & Memory, and Learning. **(b)** Optimizers instantiate the same signature by restricting which fields they edit. LoRA edits Intelligence weights; DSPy and GEPA edit Agent prompts; LLM-guided spec search edits Intelligence, Engine, Agents, and Tools & Memory jointly.

spec and wrapper expose Pareto structure that accuracy-only benchmarks and single-axis efficiency tools miss [50, 37, 48, 52, 32, 54].

3.3 LLM-guided spec search

LLM-guided spec search optimizes a spec by searching over its primitive fields, where each *spec* is one complete local AI configuration and changing its fields defines the search space. The Learning primitive specifies how a spec is updated from traces, namely which edits are considered, how edited specs are evaluated, and when optimization stops. In LLM-guided spec search, this loop is a local–cloud collaboration: frontier cloud models, which excel at reading traces and reasoning about coordinated edits across many primitives, propose changes to the spec at search time, while local hardware, which excels at running the resulting configuration with low latency and zero marginal API cost, executes the optimized spec at inference time. A frontier model reads eligible traces, identifies failure patterns, and proposes candidate edits across the editable primitives.

Failure clusters and the gate. The frontier model groups traces sharing a common failure mode into *failure clusters*, each annotated with student vs. teacher success rates and a natural-language characterization of the skill gap (e.g., "student fails on multi-hop questions requiring calendar lookups because it does not invoke the calendar tool"); see Appendix A.4 for the full clustering protocol. Each candidate edit is evaluated on held-out examples for the task, and accepted only if it improves the targeted failure cluster without causing unacceptable regressions elsewhere. We refer to this held-out validation check as the *gate*. Intelligence edits change model parameters or training triggers; Engine edits change runtime and serving choices; Agent edits change prompts, reasoning loops, and verification; Tools & Memory edits change Tool availability, Tool descriptions, and Memory configuration.

What runs where. At inference time, the resulting spec runs on-device for model inference and agent execution, and when an Intelligence edit triggers student training, that training also runs locally. Teacher calls provide diagnoses, edit proposals, and labels, not inference-time model calls. Cloud-as-tool use is disabled in the headline local configurations.

Relation to single-primitive optimizers. Most existing optimizers update one primitive at a time, falling into two families. Prompt and agent optimizers such as DSPy [39], GEPA [1], and ACE [94] edit Agents and, in some cases, Tools, while weight optimizers such as SFT, LoRA [31], and GRPO [77] edit Intelligence. LLM-guided spec search is complementary rather than competing: it operates on the same spec these methods would edit, and a single proposal can change any combination of the four editable primitives at once. The defining property of the spec is that Intelligence, Engine, Agents, and Tools & Memory fields are degrees of freedom in one optimizable object, so a single spec edit can simultaneously rewrite a Tool description and update model weights against the resulting prompt, something neither

Algorithm 1. LLM-guided spec search: greedy gated edits across primitives

Require: Spec S_0, teacher T, gate G, regression tolerance ϵ, budget B
1: $S \leftarrow S_0$
2: **while** not converged **and** cost $< B$ **do**
3: $C \leftarrow T.\text{diagnose}(\text{traces}(S))$
4: $e \leftarrow T.\text{propose}(S, C)$ ▷ may edit Intelligence, Engine, Agents, and Tools & Memory jointly
5: $S' \leftarrow \text{apply}(S, e)$
6: **if** $\text{GateOK}(S', S, C, \epsilon)$ **then**
7: $S \leftarrow S'$ ▷ greedy accept
8: **end if**
9: **end while**
10: **return** S

GateOK *means the target cluster improves and every non-target cluster regresses by at most* ϵ.

Algorithm 2. Evolutionary spec search

Require: Initial candidate S_0, reflection LM T, evaluator G, budget B
1: Initialize population $\mathcal{P} \leftarrow \{S_0\}$
2: **while** cost $< B$ **do**
3: $S \leftarrow$ sample candidate from Pareto frontier of $\mathcal{P}$
4: $e \leftarrow T.\text{reflect}(S, \text{traces}(S))$
5: $S' \leftarrow \text{apply}(S, e)$ ▷ reflective mutation
6: Optionally merge S' with frontier candidate S_j
7: Evaluate candidates and update Pareto frontier $\mathcal{P}$
8: **end while**
9: **return** best candidate in $\mathcal{P}$

Algorithm 3. Single-component optimizer

Require: Spec S_0, dataset $\mathcal{D}$, edit type $\tau \in \{\text{Intelligence}, \text{Agent}\}$, budget B
1: $S \leftarrow S_0$
2: **while** cost $< B$ **do**
3: $e \leftarrow$ propose edit of type τ ▷ restricted to one primitive
4: $S' \leftarrow \text{apply}(S, e)$
5: **if** loss or validation score improves on $\mathcal{D}$ **then**
6: $S \leftarrow S'$
7: **end if**
8: **end while**
9: **return** S

Cannot exploit coupling across primitives.

Figure 4: **Three ways to optimize a spec.** OPENJARVIS proposes edits to the four editable primitives and keeps an edit only if held-out performance does not regress. Evolutionary spec search maintains and merges a population of candidate *specs* [1]. Single-component baselines edit one primitive at a time.

family of single-primitive optimizers expresses natively. Figure 4 summarizes the three regimes compared in Section 4.4: LLM-guided spec search (Algorithm 1), an evolutionary spec-search baseline (Algorithm 2), and single-component optimization (Algorithm 3); Figure 8 separates the proposer axis from the move-space axis.

Search loop. Each session repeats four steps: diagnose failures, propose edits, execute candidates, and commit only gated improvements. Diagnose clusters traces where the student underperforms. Plan proposes edits over Intelligence, Engine, Agents, and Tools & Memory fields. Execute evaluates each edited spec on a held-out gate built from eligible traces, agentic datasets with known answers [73, 82], and standard benchmark splits [89, 50, 91]. Repeat commits accepted edits and rolls back rejected ones. The loop stops when gate scores stagnate for k sessions (default $k{=}5$) or the budget is exhausted. Full per-phase examples are in Appendix A.4.

Gate score. Let $G(S)$ be the held-out gate score of spec S, and let $G_c(S)$ be the score restricted to failure cluster c. For an edit e targeting cluster c, with $S' = \text{apply}(S, e)$, we accept if and only if:

$$G_c(S') > G_c(S) \quad \text{and} \quad G_{c'}(S') \geq G_{c'}(S) - \epsilon \quad \forall c' \neq c.$$

The default tolerance is $\epsilon = 1\%$. This single rule applies across Intelligence weights, Agent prompts, Tool descriptions, Memory configuration, and Engine/runtime settings.

(Optional) Composite reward for Intelligence edits. If Intelligence edits are requested involving model training, each candidate response y to a query q is scored by a composite reward over accuracy, energy, latency, and cost:

$$R(q, y) = \alpha R_{\text{acc}}(q, y) - \beta \hat{E}(q, y) - \gamma \hat{L}(q, y) - \delta \hat{C}(q, y). \tag{1}$$

The default weights are $(\alpha, \beta, \gamma, \delta) = (0.5, 0.1, 0.1, 0.3)$. The reward scores responses during GRPO training within an Intelligence edit. The gate evaluates the resulting spec as a whole. Normalization details and robustness checks are in Appendix C.6.

4 Experiments

4.1 Models, Benchmarks, and Hardware

We evaluate OPENJARVIS across 8 benchmarks, 11 local models from 4 families, 3 cloud baselines, and 7 hardware platforms. The suite contains 508 tasks, and agentic benchmarks enforce a 2-hour timeout per task. All results report the mean over 5 independent runs. Scoring uses `GPT-5-mini` as judge except where benchmarks provide deterministic grading. Appendix B details benchmarks (Appendix B.1), model choices and full accuracy tables (Appendices B.2 and B.3), hardware (Appendix B.4), and protocol (Appendix B.5).

4.2 The Spec Recovers 56–77% of the Drop and Lands Within 3.2 pp of Cloud on Average

The spec abstraction (Section 3.1) restores most of the portability that is lost when local models are dropped into cloud-designed frameworks, and the resulting on-device specs land within 3.2 pp of the best cloud model on average. We establish this with a controlled portability experiment on two production frameworks (Table 1) and a local-vs-cloud accuracy frontier drawn from the full accuracy table in Appendix B.3.

		PinchBench		GAIA	
Framework	**Condition**	Acc. (%)	Δ vs. (a)	Acc. (%)	Δ vs. (a)
OpenClaw [80]	(a) Default + cloud model	96.0	Ref.	58.0	Ref.
	(b) Default + Qwen3.5-9B	62.3	−33.7	19.2	−38.8
	(c) OPENJARVIS (with Qwen3.5-9B)	88.4	−7.6	41.5	−16.5
Hermes Agent [60]	(a) Default + cloud model	93.5	Ref.	55.1	Ref.
	(b) Default + Qwen3.5-9B	68.7	−24.8	22.4	−32.7
	(c) OPENJARVIS (with Qwen3.5-9B)	87.9	−5.6	40.8	−14.3
Recovery (a)→(b) drop closed by (c)		77% of PB drop		57% of GAIA drop	

Table 1: **Portability triangulation: cloud-designed frameworks collapse when local models are substituted; OPENJARVIS recovers most of the drop.** Three conditions per framework: (a) default + intended cloud model, (b) default + `Qwen3.5-9B`, (c) OPENJARVIS spec with Intelligence fixed at `Qwen3.5-9B` (only Engine, Agent, and Tool & Memory fields optimized). All Δ values are reported relative to the default cloud configuration (a), so (b) shows the full local-substitution drop and (c) shows the residual gap to cloud after the spec is retargeted at constant Intelligence. *Recovery* is the fraction of the (a)→(b) drop closed by (c), isolating the contribution of the spec at constant Intelligence. The cloud model is `Claude Opus 4.6` for both frameworks. Results are averaged over 5 independent runs.

OPENJARVIS reduces the local-substitution drop from 25–39 pp to 5.6–16.5 pp. Table 1 isolates portability at fixed Intelligence. Replacing each framework's intended cloud model with `Qwen3.5-9B` drops accuracy by 24.8–38.8 pp. With the same local model under an OPENJARVIS spec, the remaining drop relative to the cloud framework is only 5.6–16.5 pp. Equivalently, the spec closes 77% of the `PinchBench` drop and 56–57% of the `GAIA` drop. The drop is architectural rather than capability-bound: OpenClaw and Hermes Agent package Agent prompts, Tool descriptions, Memory configuration, Engine/runtime settings, and model-specific output expectations into personal AI stacks tuned to their intended cloud models. The OPENJARVIS *spec* exposes these assumptions as typed slots. Engine, Agent, and Tool fields can then be retargeted while holding Intelligence fixed. `GAIA` recovers less because its deep-reasoning demands exceed what a 9B model can deliver, regardless of configuration.

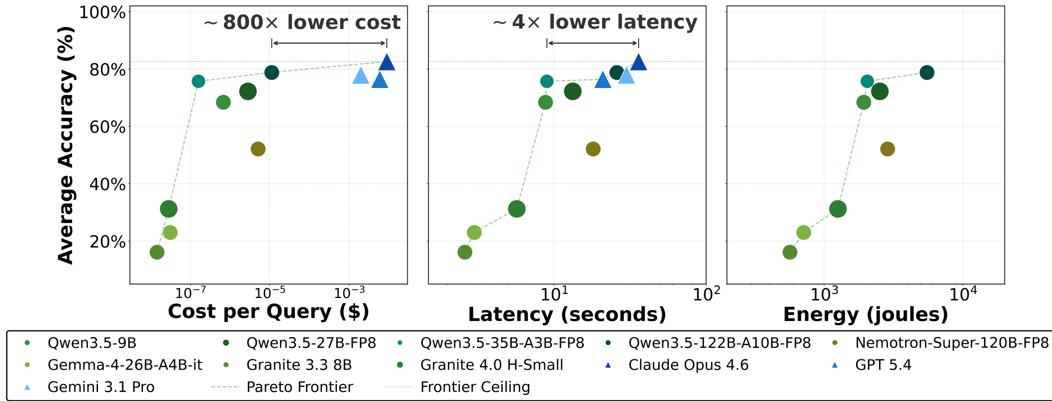

Figure 5: **Accuracy-efficiency frontier.** Local configurations approach the best cloud accuracy within 3.2 pp while reducing marginal API cost by roughly 800× and end-to-end latency by roughly 4× under our benchmark protocol. Energy and hardware-specific breakdowns are in Appendix C.3.

The best local model lands within 3.2 pp of the best cloud model on average and matches or exceeds cloud on 4 of 8 benchmarks. Across the evaluated local *specs*, the best single on-device spec, `Qwen3.5-122B`, reaches 80.3% average accuracy, within 3.2 pp of `Claude Opus 4.6` at 83.5% (Appendix B.3, Table 5). At the per-benchmark level, the best local model on each benchmark matches or exceeds the best cloud model on 4 of 8 benchmarks: `ToolCall-15`, `PinchBench`, `LiveCodeBench`, and τ-Bench V2. The remaining gaps concentrate on `GAIA`, τ^2-Bench Telecom, and `DeepResearchBench`, motivating the search experiments in Section 4.4.

4.3 Local Models Trade 3.2 pp Accuracy for 800× Lower Cost and 4× Lower Latency

Local configurations form the accuracy–cost Pareto frontier: they match cloud accuracy on four of eight benchmarks while reducing marginal API cost by roughly 800× and end-to-end latency by roughly 4×. This matters because heavy users of cloud personal AI spend thousands per year on one assistant, and none of that spend buys ownership, offline access, or price stability. We establish the comparison by treating the spec as the unit of analysis and instrumenting every query (Section 3.2), so local and cloud configurations can be compared on the same axes.

Local configurations define the accuracy and cost frontier. Figure 5 plots average accuracy against dollar cost, latency, and energy per query for the evaluated local *specs* and 3 cloud baselines. Across cost and latency, the non-dominated configurations are local. `Qwen3.5-122B` reaches 80.3% average accuracy at roughly a thousandth of a cent per query, versus \$0.009 for `Claude Opus 4.6` at 83.5%, giving an approximately 800× marginal API-cost advantage for a 3.2 pp accuracy deficit. The cost axis reports API fees only; hardware and electricity are accounted for separately via measured energy and amortization (Appendix C.3). Local configurations also complete the full agentic workloads roughly 4× faster in our protocol, though single-shot prompts can favor cloud serving due to time-to-first-token optimizations. The resulting Pareto structure motivates spec-level model and Engine selection: for a given accuracy target, a *spec* can select the local configuration that minimizes cost, latency, or energy.

4.4 LLM-guided spec search Shrinks the Cloud–Local Gap by 13–32 pp at 7–11× Lower Cost

Sections 4.2 and 4.3 show that on-device *specs* can match cloud on 4 of 8 benchmarks, but gaps remain on reasoning- and research-heavy tasks. We evaluate whether LLM-guided spec

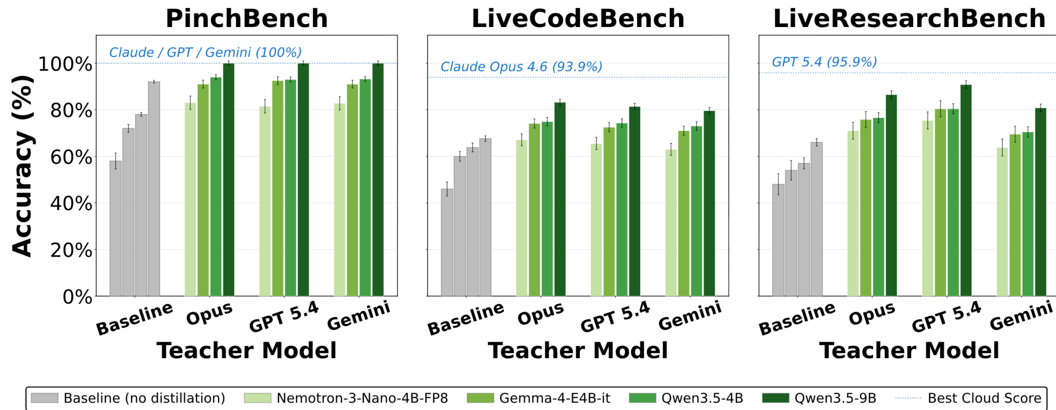

Figure 6: **LLM-guided spec search improves local *specs*.** Every student–teacher pair improves over the unoptimized spec on all three benchmarks. The strongest search-optimized `Qwen3.5-9B` student reaches 100.0% on `PinchBench`, 83.0% on `LiveCodeBench`, and 91.0% on `LiveResearchBench`.

search closes this gap on `PinchBench`, `LiveCodeBench`, and `LiveResearchBench`. For each benchmark, we run search with four local target models and three frontier proposer models; Appendix C.4 extends the evaluation to all 8 benchmarks.

LLM-guided spec search improves OpenJarvis students on personal AI benchmarks. Figure 6 shows accuracy before and after search. For `Qwen3.5-9B`, the best search-optimized spec reaches 100.0% on `PinchBench`, 83.0% on `LiveCodeBench`, and 91.0% on `LiveResearchBench`. Across the eight-benchmark suite, average gains per student model range from 13.1 to 31.5 pp (Appendix C.4, Table 9). These gains reflect joint optimization over Intelligence, Agent, Tool, Memory, and Engine primitives rather than any single component.

LLM-guided spec search improves accuracy at lower optimization cost. Figure 7 compares LLM-guided spec search with prompt-only and weight-only baselines (DSPy/SIMBA, prompt-only GEPA, SFT, LoRA [39, 1, 31]) under the same offline protocol. The prompt-only

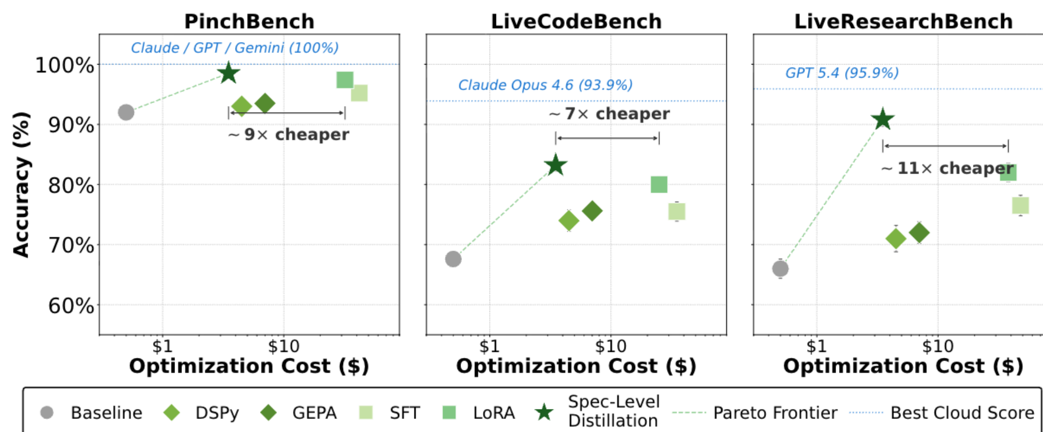

Figure 7: **Accuracy vs. optimization cost.** LLM-guided spec search reaches the best accuracy on all three main benchmarks. LoRA is the strongest single-primitive baseline, but LLM-guided spec search is 7.1–10.9× cheaper to optimize. DSPy/SIMBA and prompt-only GEPA produce modest gains over the unoptimized local spec.

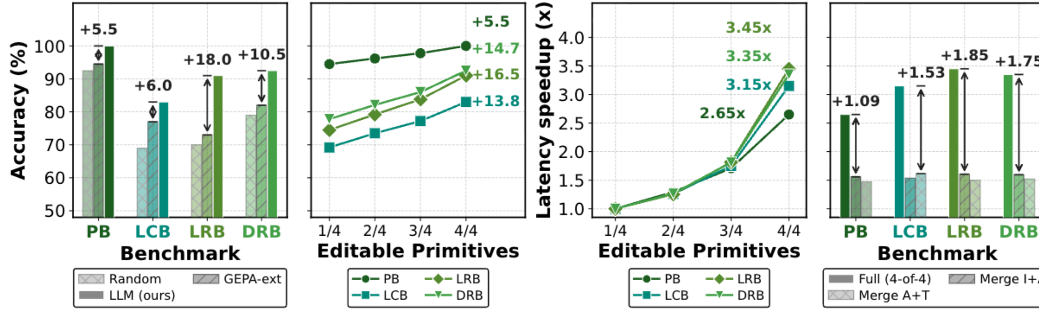

Figure 8: **Proposer and move-space ablations for LLM-guided spec search.** Student: `Qwen3.5-9B`; teacher: `Claude Opus 4.6`. **Left:** at fixed four-primitive move space, the LLM proposer outperforms a template-random proposer and evolutionary spec search. **Middle:** at fixed LLM proposer, expanding from one editable primitive to all four improves both accuracy and latency. **Right:** merging primitive pairs reduces accuracy and speedup relative to the full four-primitive spec.

baselines add 4.1–5.2 pp on average over the unoptimized local spec, and LoRA is the strongest weight-only baseline across all three panels. LLM-guided spec search adds 1.1–8.8 pp over LoRA and 5.0–18.8 pp over prompt-only GEPA at 7.1–10.9× lower optimization cost. The cost advantage comes from the acceptance gate: rejected edits do not trigger full training runs, and accepted edits compound across sessions.

Both proposer and move space drive the gain. Figure 8 separates two axes: which proposer suggests edits, and how many primitives the proposer can edit. With all four primitives editable, LLM-guided spec search reaches 83.0–100.0%, compared to 73.0–94.5% for an evolutionary spec-search proposer at the same four-primitive move space (a gap of 5.5–18.0 pp, 10.0 pp on average) and 69.0–92.5% for random sampling from the edit catalog (a gap of 7.5–21.0 pp, 14.0 pp on average). The edit catalog and gate alone therefore do not explain the gain. With the LLM teacher fixed as the proposer, expanding the editable set from one primitive to all four adds 5.5–16.5 pp in accuracy and 2.65–3.45× in latency speedup over the 1-of-4 baseline. Merging primitive pairs that deployed systems treat as independent reverses these gains, dropping accuracy by 2.8–9.7 pp and latency speedup by 1.09–1.95× relative to the full 4-of-4 configuration. Both axes matter: the LLM teacher needs the four-primitive spec to reach 83.0–100.0%, and the four-primitive spec needs the LLM teacher to extract the full gain available at that move space. Appendix C.7 specifies the random and evolutionary spec-search four-primitive baselines.

What the optimizer learns. Accepted edits are distributed across all four editable primitives rather than concentrated on Intelligence weights: weight updates make up only 16–44% of accepted edits across benchmarks, with the remainder spanning Engine, Agent, and Tool fields (Table 10). The dominant primitive within that distribution tracks task structure: Intelligence edits dominate code (44% on LiveCodeBench), Agent edits dominate customer-service and agentic tasks (41–45% on PB, TauB, and TBTel), and Tool edits dominate tool-calling and research (39–47% on TC15, GAIA, DRB, and LRB; Appendix C.5, Figure 10). At the per-failure level, the teacher maps each diagnosis to the corresponding primitive: retrieval failures receive mostly Tool edits, reasoning failures mostly Intelligence edits, control-flow failures mostly Agent edits, and efficiency-bounded failures mostly Engine edits (Figure 11). This explains why no single primitive reaches the joint-search ceiling: the useful intervention depends on the failure mode, and many failures require coordinated changes across multiple primitives.

5 Discussion and Conclusion

OpenJarvis shows that local personal AI requires more than replacing a cloud model with an open-weight model. The surrounding stack must be portable, measurable, and optimizable around the local model. The spec abstraction provides the needed representation: it makes model substitution recoverable, exposes the local–cloud accuracy–efficiency frontier, and gives LLM-guided spec search a target for transferring frontier-model guidance into an on-device system. We close by clarifying what runs on-device, how frontier proposer models are used, and which robustness checks support the main claims.

Cloud at search time, local at inference time. The local–cloud division of labor in LLM-guided spec search is deliberate: frontier cloud models are used at search time, where their capability advantage in reading traces and reasoning about coordinated edits matters most, and local hardware runs the resulting spec at inference time, where its latency, cost, and ownership properties matter most. Capability flows from cloud to local through the optimized spec and stays there: the resulting local spec makes zero cloud calls at inference time, and undistilled local specs already match or beat cloud models on 4 of 8 benchmarks, so frontier proposer models extend local capability rather than being required for local systems to function. When a frontier proposer is used at search time, only eligible traces are transmitted according to the protocol in Appendix A.3, and at 100 queries per day the amortized proposer cost falls below $0.001 per query within six months (Appendix C.2).

Robustness and limitations. The headline gains from LLM-guided spec search (Section 4) survive the four robustness perturbations we test: reward-weight variants, search-seed variance, random-restart gains, and proposer comparison all produce variation smaller than the headline effect (Appendix C.6). Remaining limitations are statistical precision from 5 runs per configuration, possible judge bias from `GPT-5-mini`, and single-machine evaluation only (Appendix D). Broader impacts and release safeguards are discussed in Appendix E.

Acknowledgements

We thank Kelly Buchanan, Francois Chaubard, Mayee Chen, Vivien Cheng, Catherine Deng, Neel Guha, Simon Guo, Braden Hancock, Junmiao Hu, Hangoo Kang, Andy Konwinski, Hermann Kumbong, Jacky Kwok, Adrian Lafuente Gamarra, Jerry Liu, Yuzhen Mao, Christopher Rytting, Tarun Suresh, Shayan Talaei, Roberto Torres, and Michael Zhang. We would also like to thank our collaborators at the Stanford Artificial Intelligence Laboratory (SAIL) and Stanford HAI.

We gratefully acknowledge support from federal sources: NIH under Nos. U54EB020405 (Mobilize) and R01GM124443 (SimTK); NSF under Nos. CCF2247015 (Hardware-Aware), CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), 1937301 (RTML), 1900638 (Open Federated Virtual Assistants), and 2028602 (PPoSS); DARPA under No. HR00112520038 (Fallingwater); US DEVCOM ARL under Nos. W911NF-23-2-0184 (Long-context) and W911NF-21-2-0251 (Interactive Human-AI Teaming); NSTXL under No. N00164-23-9-G057-01 (Energy-Efficient AI Hardware); and ONR under Nos. N000142312633 (Deep Signal Processing), N000142212426 (Misinformation Analysis), N000142012480 (Non-Euclidean Geometry), and N000142012275 (Data Programming).

We also gratefully acknowledge support from Stanford HAI under Nos. 247183, 215955, and 345077 (Evo); Google DeepMind, Google Research, and Google Cloud; member companies including NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, Salesforce, and Total; the Laude Institute, Prime Intellect, Anthropic, and the HAI-GCP Cloud Credits for Research program; the Stanford Data Science Initiative (SDSI) and the Stanford Marlowe Computing Platform; the NSF Graduate Research Fellowship Program, the Knights-Hennessy Scholarship, the Stanford Graduate Fellowship, the JP Morgan AI/ML Fellowship, the Stanford EDGE Fellowship, and the GEM Fellowship; members of the Stanford DAWN project (Meta, Google, VMWare); and members of the Stanford SEAMS project (IBM, Felicis).

The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of NIH, ONR, DARPA, or the U.S. Government.

References

[1] Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, Christopher Potts, Koushik Sen, Alexandros G. Dimakis, Ion Stoica, Dan Klein, Matei Zaharia, and Omar Khattab. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026.

[2] Alibaba Cloud. Qwen-Agent. https://github.com/QwenLM/Qwen-Agent, 2025. Agent framework built on Qwen models with tool use and RAG capabilities.

[3] Anthropic. Model context protocol, 2024. Open standard for connecting AI assistants to external tools and data sources.

[4] Anthropic. Claude Opus 4.6: frontier model with extended-thinking and tool-use capabilities. Anthropic API Documentation, 2026. https://docs.anthropic.com.

[5] Apple Inc. Apple M4 chip. https://www.apple.com/newsroom/2024/05/apple-introduces-m4-chip/, 2024. Accessed: April 2026.

[6] Apple Machine Learning Research. Apple intelligence foundation language models. *arXiv preprint arXiv:2407.21075*, 2024.

[7] Parth Asawa, Alan Zhu, Abby O'Neill, Matei Zaharia, Alexandros G. Dimakis, and Joseph E. Gonzalez. How to train your advisor: Steering black-box LLMs with advisor models. *arXiv preprint arXiv:2510.02453*, 2025.

[8] Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-bench: Evaluating conversational agents in a dual-control environment. *arXiv preprint arXiv:2506.07982*, 2025.

[9] BCD International. Inside the 2025–2027 compute crunch: What supply chain volatility really means for you. BCD Video Blog, 2025. Accessed: April 2026.

[10] Harrison Chase. LangChain. https://github.com/langchain-ai/langchain, 2022. Accessed: 2026.

[11] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[12] Xiaopei Chen, Liang Li, Fei Ji, and Wen Wu. MobileFineTuner: A unified end-to-end framework for fine-tuning LLMs on mobile phones. *arXiv preprint arXiv:2512.08211*, 2025.

[13] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Banghua Zhu, Hao Zhang, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating LLMs by human preference, 2024.

[14] Gordon V. Cormack, Charles L. A. Clarke, and Stefan Büttcher. Reciprocal rank fusion outperforms Condorcet and individual rank learning methods. In *Proceedings of the 32nd International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 758–759. ACM, 2009.

[15] CrewAI, Inc. CrewAI: Framework for orchestrating role-playing, autonomous AI agents. https://github.com/crewAIInc/crewAI, 2024. Accessed: 2026.

[16] DeepSeek-AI. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[17] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized language models. In *Advances in Neural Information Processing Systems*, volume 36, 2023.

[18] Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. Deepresearch bench: A comprehensive benchmark for deep research agents, 2025.

[19] EdgeClaw Contributors. EdgeClaw: Local-cloud router plugin for OpenClaw. https://github.com/edgeclaw/edgeclaw, 2025. Adds a local-cloud routing layer on top of OpenClaw as a plugin.

[20] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2023.

[21] Georgi Gerganov. llama.cpp. https://github.com/ggml-org/llama.cpp, 2023. C/C++ LLM inference engine with quantization support. 85K+ GitHub stars as of 2025.

[22] Google. gemma.cpp. https://github.com/google/gemma.cpp, 2024. Lightweight C++ inference engine for Gemma models.

[23] Google. Agent development kit (ADK). https://github.com/google/adk-python, 2025. Python framework for building AI agents with tool use and multi-step reasoning.

[24] Google. Gemma 3n. https://ai.google.dev/gemma/docs/gemma-3n, 2025. On-device Gemma variant with per-layer embeddings and elastic inference for phones and laptops.

[25] Google. Gemma 4 26B: instruction-tuned open-weight 26B-parameter mixture-of-experts model. Model card, Hugging Face, 2025. https://huggingface.co/google/gemma-4-26b-it.

[26] Google. Gemini nano. https://developer.android.com/ai/gemini-nano, 2026. Android Developers Documentation, last updated April 2, 2026.

[27] Google DeepMind. Gemini 3.1 Pro: frontier multimodal cloud model with extended-context reasoning. Google AI Developer Documentation, 2026. https://ai.google.dev.

[28] Google DeepMind. Gemma 4: Byte for byte, the most capable open models. https://blog.google/innovation-and-ai/technology/developers-tools/gemma-4/, April 2026. Blog post, April 2, 2026.

[29] Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. MiniLLM: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[30] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[31] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations*, 2022.

[32] HuggingFace. AI energy score. https://huggingface.github.io/AIEnergyScore/, 2024. Standardized energy scoring system for AI models.

[33] IBM Research. Granite 4.0 language models. https://github.com/ibm-granite/granite-4.0-language-models, 2025. Accessed: 2025-10-01.

[34] Intel Corporation. Intel Core Ultra processors (series 1, meteor lake). https://www.intel.com/content/www/us/en/products/details/processors/core-ultra.html, 2023. Accessed: April 2026.

[35] IronClaw Contributors. IronClaw. https://github.com/ironclaw/ironclaw, 2025. Enterprise-focused fork of OpenClaw.

[36] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

[37] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world Github issues? In *The Twelfth International Conference on Learning Representations*, 2024.

[38] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[39] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines. In *The Twelfth International Conference on Learning Representations*, 2024.

[40] Kilo Code. PinchBench: Benchmarking LLM models as OpenClaw coding agents. https://github.com/pinchbench/skill, 2025. 23 real-world agent tasks spanning scheduling, email, research, coding, and multi-step workflows. Open-source grading via automated checks and LLM judge.

[41] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[42] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, 2023.

[43] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *Proceedings of Machine Learning and Systems*, volume 6, 2024.

[44] Ji Lin, Ligeng Zhu, Wei-Ming Chen, Wei-Chen Wang, Chuang Gan, and Song Han. On-device training under 256KB memory. In *Advances in Neural Information Processing Systems*, volume 35, 2022.

[45] Liquid AI. Liquid nanos: Frontier-grade performance on everyday devices. https://www.liquid.ai/blog/introducing-liquid-nanos-frontier-grade-performance-on-everyday-devices, 2025.

[46] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, Vikas Chandra, et al. MobileLLM: Optimizing sub-billion parameter language models for on-device use cases. *arXiv preprint arXiv:2402.14905*, 2024.

[47] LM Studio. LM Studio. https://lmstudio.ai, 2024. Desktop application for running LLMs locally.

[48] Mike A. Merrill, Alexander G. Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E. Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. In *The Fourteenth International Conference on Learning Representations*, 2026.

[49] Meta. ExecuTorch: End-to-end solution for enabling on-device inference capabilities. https://github.com/pytorch/executorch, 2024.

[50] Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. GAIA: A benchmark for general AI assistants. In *The Twelfth International Conference on Learning Representations*, 2024.

[51] MimicLaw Contributors. MimicLaw. https://github.com/mimiclaw/mimiclaw, 2025. Persona-focused personal AI framework.

[52] ML Energy Initiative. Zeus: ML energy measurement framework. https://ml.energy/zeus/, 2023. GPU energy measurement toolkit for fine-grained energy profiling of ML workloads.

[53] MLC AI. MLC-LLM: Universal LLM deployment engine. https://github.com/mlc-ai/mlc-llm, 2023.

[54] MLCommons. MLCommons inference benchmark. https://github.com/mlcommons/inference, 2024. Industry-standard inference benchmarking suite covering latency, throughput, and accuracy across hardware platforms.

[55] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. Ludwig: a type-based declarative deep learning toolbox, 2019.

[56] Ettore Di Giacinto Mudler. LocalAI: Open-source self-hosted alternative to OpenAI API. https://github.com/mudler/LocalAI, 2024.

[57] NanoBot Contributors. NanoBot. https://github.com/nanobot-ai/nanobot, 2025. Minimalist personal AI agent in the OpenClaw ecosystem.

[58] Avanika Narayan, Dan Biderman, Sabri Eyuboglu, Avner May, Scott Linderman, James Zou, and Christopher Ré. Minions: Cost-efficient collaboration between on-device and cloud language models. *arXiv preprint arXiv:2502.15964*, 2025.

[59] National Institute of Standards and Technology. Artificial intelligence risk management framework (AI RMF 1.0). Technical Report NIST AI 100-1, U.S. Department of Commerce, January 2023.

[60] Nous Research. Hermes agent: The agent that grows with you. https://github.com/NousResearch/hermes-agent, 2025. Self-improving agent with FTS5 cross-session recall, Honcho user modeling, and autonomous skill creation. 40K+ GitHub stars as of April 2026.

[61] NVIDIA. Nemotron-Flash: Towards latency-optimal hybrid small language models. *arXiv preprint arXiv:2511.18890*, 2025.

[62] NVIDIA. Nemotron-Super-49B-v1. https://huggingface.co/nvidia/Nemotron-Super-49B-v1, 2025.

[63] Ollama, Inc. Ollama. https://github.com/ollama/ollama, 2023. Local LLM serving platform. 162K+ GitHub stars as of March 2026.

[64] OpenAI. Symphony: Multi-agent orchestration framework. https://github.com/openai/symphony, 2025. Multi-agent orchestration framework for coordinating agent workflows.

[65] OpenAI. GPT-5.4: frontier reasoning and multimodal model. OpenAI Platform Documentation, 2026. https://platform.openai.com.

[66] OWASP Foundation. OWASP top 10 for large language model applications, 2025. Version 2025. Accessed: April 2026.

[67] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019.

[68] PicoClaw Contributors. PicoClaw. https://github.com/picoclaw/picoclaw, 2025. Lightweight variant of the OpenClaw personal AI ecosystem.

[69] Konstantin F. Pilz, Yusuf Mahmood, and Lennart Heim. AI's power requirements under exponential growth: Extrapolating AI data center power demand and assessing its potential impact on U.S. competitiveness. Research Report RR-A3572-1, RAND Corporation, 2025.

[70] Qualcomm Technologies. Qualcomm Hexagon neural processing unit. https://www.qualcomm.com/products/technology/processors, 2024. Accessed: April 2026.

[71] Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026.

[72] Christopher Ré, Feng Niu, Pallavi Gudipati, and Charles Srisuwananukorn. Overton: A data system for monitoring and improving machine-learned products, 2019.

[73] RJT1990. GeneralThoughtArchive: A large-scale dataset of reasoning traces, 2025. 431K reasoning traces with verifier scores. MIT license.

[74] Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In *The Third Text REtrieval Conference (TREC-3)*. NIST, 1994.

[75] Jon Saad-Falcon, Avanika Narayan, Hakki Orhun Akengin, J. Wes Griffin, Herumb Shandilya, Adrian Gamarra Lafuente, Medhya Goel, Rebecca Joseph, Shlok Natarajan, Etash Kumar Guha, Shang Zhu, Ben Athiwaratkun, John Hennessy, Azalia Mirhoseini, and Christopher Ré. Intelligence per watt: Measuring intelligence efficiency of local ai, 2026.

[76] Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. ColBERTv2: Effective and efficient retrieval via lightweight late interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3715–3734. Association for Computational Linguistics, 2022.

[77] Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, Y. K. Li, Y. Wu, and Daya Guo. DeepSeekMath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[78] Will Sommer. Gartner predicts that by 2030, performing inference on an LLM with 1 trillion parameters will cost GenAI providers over 90% less than in 2025. Gartner Press Release, 2026. Accessed: April 2026.

[79] Dilara Soylu, Christopher Potts, and Omar Khattab. Fine-tuning and prompt optimization: Two great steps that work better together. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 10696–10710. Association for Computational Linguistics, 2024.

[80] Peter Steinberger. OpenClaw: Open-source personal AI assistant. https://github.com/openclaw/openclaw, 2025. 131K+ GitHub stars as of March 2026. Originally named Clawdbot.

[81] stevibe. ToolCall-15: A visual benchmark for comparing LLM tool use. https://github.com/stevibe/ToolCall-15, 2025. 15 fixed scenarios across 5 categories with deterministic scoring.

[82] Hongjin Su, Shizhe Diao, Ximing Lu, Mingjie Liu, Jiacheng Xu, Xin Dong, Yonggan Fu, Peter Belcak, Hanrong Ye, Hongxu Yin, Yi Dong, Evelina Bakhturina, Tao Yu, Yejin Choi, Jan Kautz, and Pavlo Molchanov. ToolOrchestra: Elevating intelligence via efficient model and tool orchestration, 2025.

[83] Tether Data. An edge-first generalized LLM LoRA fine-tuning framework for heterogeneous GPUs, 2025. https://huggingface.co/blog/qvac/fabric-llm-finetune.

[84] TinyClaw Contributors. TinyClaw. https://github.com/tinyclaw/tinyclaw, 2025. Resource-constrained variant of OpenClaw.

[85] UIUC UILab. LLMRouter. https://github.com/ulab-uiuc/LLMRouter, 2025. Framework for routing queries to appropriate LLMs based on query characteristics and model capabilities.

[86] Jiayu Wang, Yifei Ming, Riya Dulepet, Qinglin Chen, Austin Xu, Zixuan Ke, Frederic Sala, Aws Albarghouthi, Caiming Xiong, and Shafiq Joty. LiveResearchBench: A live benchmark for user-centric deep research in the wild. *arXiv preprint arXiv:2510.14240*, 2025.

[87] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better LLM agents. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, volume 235, pages 50208–50232, 2024.

[88] Yinjie Wang, Xuyang Chen, Xiaolong Jin, Mengdi Wang, and Ling Yang. OpenClaw-RL: Train any agent simply by talking. *arXiv preprint arXiv:2603.10165*, 2026.

[89] Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyan Jiang, Tianle Li, Max Ku, Kai Wang, Alex Zhuang, Rongqi Fan, Xiang Yue, and Wenhu Chen. MMLU-Pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37, 2024.

[90] Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. A survey on knowledge distillation of large language models. *arXiv preprint arXiv:2402.13116*, 2024.

[91] Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

[92] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

[93] ZeroClaw Contributors. ZeroClaw. https://github.com/zeroclaw/zeroclaw, 2025. Zero-configuration variant of OpenClaw.

[94] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, Urmish Thakker, James Zou, and Kunle Olukotun. Agentic context engineering: Evolving contexts for self-improving language models, 2026.

[95] Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody Hao Yu, Hao Zhang, and Ion Stoica. SGLang: Efficient execution of structured language model programs. *arXiv preprint arXiv:2312.07104*, 2024.

[96] Zylo. 2026 SaaS Management Index. https://zylo.com/2026-saas-management-index, January 2026. Analysis of $75B in SaaS spend and 40M licenses under management; reports 108% year-over-year growth in AI-native application spend. Accessed: May 2026.

Category	Tool	Runs	Cost / call
Built-in Tools			
Reasoning	`think`	Local	$0
Math	`calculator`	Local	$0
Code	`code_interpreter`	Local	$0
Code	`code_interpreter_docker`	Local	$0
Code	`repl`	Local	$0
Search	`web_search`	API	$0.005–0.01[†]
File I/O	`file_read`	Local	$0
HTTP	`http_request`	Mixed	Varies
Memory	`retrieval`, `memory_search`, `memory_index`	Local	$0
Inference	`llm`	Mixed	$0 local; API price cloud
Scheduler	`schedule_task`, `list/pause/resume/cancel`	Local	$0
Integration	`mcp_adapter`	Mixed	$0
Connectors (25+ data sources)			
Gmail, Google Calendar, Google Drive, Google Contacts, Google Tasks, Apple Health, Apple Notes, Apple Music, Apple Contacts, iMessage, Notion, Obsidian, Slack, Spotify, Strava, Oura, Outlook, Dropbox, GitHub, Hacker News, Weather, RSS, ...			
Channels (32+ messaging platforms)			
WhatsApp, Telegram, Discord, Slack, iMessage, SMS (Twilio), Signal, Teams, Messenger, Email, IRC, Matrix, Mastodon, Reddit, Line, Viber, Webchat, ...			

Table 2: **Built-In Tools, Connectors, and Channels in OpenJarvis.** *Cost* indicates whether the tool incurs a per-call API fee; all other tools run locally at zero dollar cost. Connectors and channels are listed by count; full lists are in the documentation.

[†]Approximate per-query cost for Google Custom Search ($5/1K queries), Tavily ($0.005/query on paid plans), or Brave Search ($0.009/query). Free tiers available with rate limits.

A Methods Details

A.1 Primitive Implementation Details

This section expands on the five primitives introduced in Section 3.1.

Intelligence: Model Catalog. The Intelligence catalog maintains metadata for each supported model: parameter count, context length, VRAM requirements, and compatible engines. At runtime, the catalog merges static entries with models discovered from running backends (e.g., models pulled into `Ollama`), so the system always has a current view of what is locally available. The user or the system can select the model best suited to the available hardware from this unified catalog.

Engine: Hardware Support. The hardware-detection module identifies the accelerator type and recommends the best Engine automatically. Supported device classes include:

- **Phones**: iPhone, Google Pixel
- **Laptops**: MacBook Pro M4, AMD Ryzen AI
- **Workstations**: NVIDIA RTX 4090, AMD RX 7900 XTX
- **Dedicated AI hardware**: NVIDIA DGX Spark

Agents: Discrete and Continuous. *Discrete Agents* perform actions only when called. Available types include single-turn chat, multi-turn orchestration with function calling, ReAct [92] loops, and CodeAct-style [87] code execution. *Continuous Agents* run persistently, monitoring for events, maintaining state across sessions, and taking actions autonomously over long horizons. Examples include scheduled morning digests that summarize email, calendar, and news; ongoing research tasks that monitor arXiv for relevant papers; and code review agents that watch GitHub repositories.

Tools & Memory: Built-in Tools, Connectors, and Channels. Tools are the interfaces through which an Agent interacts with the outside world and the user's personal data. Built-in Tools span seven categories: reasoning, math, code execution, web search, file I/O, memory, and inference delegation (Table 2). Web search is available via Google, Tavily, or Brave APIs; for on-device configurations, these tool API calls are the primary source of dollar cost, since local model inference has no marginal API fee. An MCP [3] adapter allows any external MCP server to be used as a Tool. Connectors pull data from 25+ sources (Gmail, Calendar, Apple Health, iMessage, Notion, Slack, and others); channels expose the Agent over 32+ messaging platforms (WhatsApp, Telegram, Discord, SMS, and others). Memory is the persistent, searchable storage layer, with interchangeable backends: SQLite/FTS5 (default), FAISS [38], ColBERTv2 [76], BM25 [74], and a hybrid using reciprocal rank fusion [14].

Learning: Training Data Sources. Intelligence-Based Learning can draw training data from three sources:

1. **Large-scale agentic datasets**: publicly available datasets such as GeneralThoughtArchive [73] and ToolScale [82] provide diverse, verifiable queries spanning reasoning, tool use, and multi-turn interaction.
2. **Synthetically generated traces**: the teacher model generates high-quality traces for specific query classes where the student underperforms.
3. **Eligible user traces**: user-approved scrubbed interaction traces, annotated by the teacher or by ground-truth labels, provide signal grounded in the user's actual workload.

Agent-Based Learning supports several optimization approaches:

- **DSPy-based prompt optimization** [39]: compiles declarative signatures into optimized prompts via bootstrapped few-shot demonstrations.
- **Structured editing**: direct modification of system prompts, user prompt formatting, tool-use exemplars, and few-shot examples.

Spec format and TOML serialization. Section 3.1 introduces the spec via the typed declaration in Figure 3. On disk, a spec is serialized as a TOML configuration that specifies:

- **Intelligence**: model identifier, temperature, top-p, max tokens, quantization format
- **Engine**: backend name, batch size, KV-cache settings
- **Agent**: agent type, system prompt, few-shot exemplars, turn limits
- **Tools & Memory**: enabled Tools, Tool descriptions, Memory configuration, Connectors, Channels
- **Learning**: optimization approach, training data sources, reward weights

The primitives communicate through a thread-safe publish-subscribe EventBus, so adding a new component requires implementing only the relevant interface. Figure 9 shows two example *specs* that differ across all five primitives while sharing the same MCP tool layer and security configuration.

A.2 Privacy and Security Architecture

This section expands on the privacy properties referenced in Section 3.1.

Privacy by architecture. Model inference, Engine execution, Agent state, telemetry, and eligible Learning updates run on-device. Tools may cross the device boundary when connectors fetch from opted-in external services or the system calls an external API. This separation provides privacy by construction, consistent with NIST AI RMF [59]. For ordinary inference, no traces, Memory contents, or training data leave the device. LLM-guided spec search is the exception: when a cloud teacher is enabled, only eligible scrubbed traces are transmitted during the bounded search phase (Appendix A.3).

(a) Consumer deployment
Mac Mini M4, 24 GB

```
[intelligence]
default_model = "gemma4:4b-it"
quantization = "fp16"
max_tokens = 4096

[engine]
default = "ollama"

[agent]
default_agent = "simple"
max_turns = 10
tools = "think,calc,web_search"

[tools.storage]
default_backend = "sqlite"

[learning]
enabled = false
```

(b) Workstation deployment
NVIDIA H100, 80 GB

```
[intelligence]
default_model = "qwen3.5:122b"
quantization = "fp8"
max_tokens = 8192

[engine]
default = "vllm"

[agent]
default_agent = "native_openhands"
max_turns = 50
tools = "think,calc,code_interpreter,
         web_search,file_read,git_tool"

[tools.storage]
default_backend = "bm25"

[learning]
enabled = true
policy = "spec_distillation"
```

Figure 9: **Two *specs* for the same personal AI system on different hardware.** The Consumer deployment serves Gemma4-4B via Ollama with a single-turn agent and learning disabled. The Workstation deployment serves Qwen3.5-122B (FP8) via vLLM with a multi-step coding agent, expanded tool set, and LLM-guided spec search enabled. `[tools.mcp]`, `[security]`, and connectors (omitted) are identical across both.

Security at the Tools boundary. A security layer at the Tools boundary addresses the following risk categories from the OWASP Top 10 for LLM Applications [66]:

- **Sensitive-data scanning**: outbound queries are scanned for PII, credentials, and other sensitive data before transmission.
- **Prompt-injection detection**: tool outputs are screened for adversarial instructions before being fed back to the model.
- **SSRF protection**: URL-based tool calls are validated against an allowlist to prevent server-side request forgery.
- **Sandboxed execution**: code interpreter tools run in isolated containers with restricted filesystem and network access.

A.3 Cloud-Teacher Search Privacy

LLM-guided spec search transmits trace data to a teacher only during the search phase. At inference time, the resulting local spec runs model inference and agent execution on-device. Three controls bound data exposure during cloud-teacher search.

Scrubbing pipeline. Before any eligible trace is transmitted, it passes through the same sensitive-data scanner used at the Tools boundary (Appendix A.2). The scanner redacts PII, credentials, and user-configured sensitive patterns, such as account numbers or medical identifiers.

Trace eligibility. Users specify which trace categories are eligible for cloud-teacher search. By default, traces from Connectors marked sensitive are excluded. Users can also mark specific conversations, sessions, or connectors as ineligible.

Local-teacher fallback. Users who require strict local-only operation can substitute a larger local model as the teacher. This eliminates cloud transmission entirely, but we treat the resulting accuracy and convergence tradeoff as a deployment choice rather than the default setting evaluated in the main experiments.

Comparison to cloud personal AI. Frameworks like OpenClaw and Hermes Agent transmit inference queries to a cloud model by default. LLM-guided spec search confines cloud exposure to a bounded, optional search phase and applies a scrubbing layer before transmission.

A.4 LLM-guided spec search Details

This section provides expanded examples for each phase of LLM-guided spec search (Section 3.3).

Phase 1: Diagnose (Expanded). The teacher LM ingests an eligible trace corpus, a dataset of structured JSON records drawn from benchmark traces, synthetic traces, or user-approved scrubbed traces. Since traces are plain text, the teacher can read them directly, grep through fields, and run scripts over the JSON to identify patterns. The teacher explores failure modes across query classes, compares its own outputs against the student's on held-out tasks, and produces failure clusters. Each cluster is annotated with student vs. teacher success rates and a natural-language characterization of the skill gap (e.g., "student fails on multi-hop questions requiring calendar lookups because it does not invoke the calendar tool").

Phase 2: Plan (Expanded Edit Examples). The teacher proposes edits targeting specific primitives of the spec:

1. **Intelligence edits** modify the language model and its parameters. Examples include changing the model (e.g., switching from `Qwen3.5-4B` to `Qwen3.5-9B` for code queries), adjusting generation parameters (temperature, top-p, max tokens), changing the quantization format (e.g., Q4 to Q8), or triggering LoRA [31] fine-tuning on teacher-generated SFT pairs or GRPO [77] training with a composite reward (Equation 1).
2. **Engine edits** change the inference backend or its configuration. Examples include switching from `Ollama` to `vLLM` for higher throughput, adjusting batch size or KV-cache settings, or enabling a different quantization kernel.
3. **Agent edits** modify the reasoning loop. Examples include replacing or refining the system prompt, editing few-shot exemplars, adjusting turn limits or verification steps, switching the Agent type (e.g., from single-turn to ReAct), or restructuring the tool-calling strategy.
4. **Tools & Memory edits** change the available Tools, their descriptions, and Memory configuration. Examples include adding or removing Tools from a given Agent, revising Tool descriptions to improve the model's tool selection, changing Memory configuration, and configuring a cloud model (`GPT 5.4`, `Claude Opus 4.6`, `Gemini 3.1 Pro`) as an additional Tool when that setting is enabled.

All edits are logged so that the teacher can examine its own intervention history across sessions and identify which changes helped vs. which did not.

Phase 3: Execute (Expanded Gate Description). The benchmark gate draws on three sources of signal:

1. **Synthetically annotated or eligible user traces**: the teacher annotates synthetic traces or user-approved scrubbed traces with ground-truth labels, providing signal grounded in the target workload.
2. **Large-scale agentic datasets with known answers**: datasets such as GeneralThoughtArchive [73] and ToolScale [82] provide diverse, verifiable queries spanning reasoning, tool use, and multi-turn interaction.
3. **Standard benchmark splits**: train/test splits of established benchmarks such as MMLU-Pro [89], GAIA [50], and τ-bench [91] provide calibrated difficulty levels and reproducible evaluation.

An edit is accepted if the updated spec improves on the target failure cluster without excessive regression on other clusters. Rejected edits are logged with their gate results.

B Experimental Details

B.1 Benchmark Descriptions

Table 3: Benchmark suite spanning eight personal AI workload categories (508 tasks total).

Benchmark	Category	Tasks	Scoring
ToolCall-15	Tool calling	15	Automated
PinchBench	Agent tasks	23	Auto + LLM judge
LiveCodeBench (v6)	Coding	100	Automated
τ-Bench V2	Customer service	100	DB-state match
τ^2-Bench Telecom	Customer service	40	DB-state match
GAIA	General assistant	50	Exact match
LiveResearchBench (v4)	Deep research	100	Checklist + LLM
DeepResearchBench	Deep research	80	RACE + FACT

- ToolCall-15 (TC15) [81] evaluates single-turn tool-calling accuracy across 15 fixed scenarios organized into 5 categories. Each scenario provides a mocked tool environment with deterministic expected behavior; scoring is fully automated (pass, partial, or fail). We use all 15 scenarios.
- PinchBench (PB) [40] measures end-to-end agent task completion across 23 tasks spanning calendar management, email handling, research, coding, and multi-step workflows. Tasks are graded via a combination of automated checks and LLM-judge rubrics. Originally developed as the canonical benchmark for the OpenClaw agent ecosystem [80], PinchBench tests practical agent competence rather than isolated model capabilities.
- LiveCodeBench (LCB) [36] provides contamination-free competitive programming evaluation by continuously collecting problems from LeetCode, AtCoder, and CodeForces. We use release_v6 (problems through April 2025; 1,055 total) and evaluate on a 100-problem subset sampled from problems released after January 2025 to minimize contamination risk. Scoring is via automated test-case execution.
- τ-Bench V2 (TauB) [91] simulates multi-turn customer service conversations between a simulated user and an agent equipped with domain-specific API tools and policy guidelines. We evaluate on 100 tasks across the airline and retail domains, using database-state comparison for faithful automated evaluation.
- τ^2-Bench Telecom (TBTel) [8] extends τ-Bench to a dual-control telecom domain where both agent and user modify a shared environment. We evaluate on 40 tasks. This domain is particularly challenging because the agent must guide users through technical troubleshooting, requiring coordination beyond standard single-control customer service.
- GAIA [50] tests general AI assistant capabilities with 50 questions requiring web search, multi-step reasoning, and tool use. Tasks range from simple factual lookups to complex multi-hop queries demanding synthesis across sources. Evaluation uses exact-match scoring with a 2-hour timeout per task.
- LiveResearchBench (LRB) [86] evaluates deep research capabilities with 100 expert-curated tasks spanning daily life, enterprise, and academia, each requiring extensive real-time web search and multi-source synthesis. Built with over 1,500 hours of human labor, tasks are designed to be user-centric, dynamic, and unambiguous. We evaluate on 100 tasks using the DeepEval checklist-based scoring protocol. We use the v4 release (December 2025).
- DeepResearchBench (DRB) [18] evaluates deep research agents on 100 PhD-level tasks across 22 domains, with each task crafted by domain experts. The task distribution reflects real-world research demand, derived from an analysis of over 96,000 user queries. We evaluate on an 80-task subset. Evaluation uses two complementary frameworks: RACE (Reference-based Adaptive Criteria-driven Evaluation), which scores generated reports against a reference across four dimensions (comprehensiveness, insight, instruction-following, and readability) using dynamically weighted, task-specific criteria; and FACT (Factual Abundance and Citation Trustworthiness), which measures citation accuracy and the number of verifiably supported claims.

B.2 Model Selection Rationale

Table 4: Models evaluated. *Type* indicates architecture (Dense, MoE, or Hybrid Mamba-2). *Active* denotes parameters per forward pass. *Quant* indicates quantization format. *Engine* is the inference backend within OPENJARVIS. *License* is listed for reproducibility.

Model	Type	Active	Quant	Engine	License
Local models: Qwen3.5 family [71]					
`Qwen3.5-4B`	Dense	4B	FP16	`Ollama`	Apache 2.0
`Qwen3.5-9B`	Dense	9B	FP16	`vLLM` / `Ollama`	Apache 2.0
`Qwen3.5-27B`	Dense	27B	FP8	`vLLM`	Apache 2.0
`Qwen3.5-35B`	MoE	$\sim$3B	FP16	`vLLM` / `Ollama`	Apache 2.0
`Qwen3.5-122B`	MoE	$\sim$10B	FP8	`vLLM`	Apache 2.0
Local models: Nemotron family [62]					
`Nemotron-Nano-4B`	Hybrid	4B	FP8	`vLLM`	Nemotron Open
`Nemotron-Super-120B`	Hybrid MoE	$\sim$12B	FP8	`vLLM`	Nemotron Open
Local models: Gemma4 family [25]					
`Gemma4-E4B`	Dense (PLE)	$\sim$4B	FP16	`Ollama`	Apache 2.0
`Gemma4-26B`	MoE	$\sim$4B	FP16	`vLLM`	Apache 2.0
Local models: Granite family [33]					
`Granite 3.3 8B`	Dense	8B	FP16	`Ollama`	Apache 2.0
`Granite 4.0 H-Small`	Hybrid MoE	$\sim$9B	FP16	`vLLM`	Apache 2.0
Cloud baselines					
`Claude Opus 4.6` [4]	Undisc.	Undisc.	Undisc.	Cloud API	Proprietary
`GPT 5.4` [65]	Undisc.	Undisc.	Undisc.	Cloud API	Proprietary
`Gemini 3.1 Pro` [27]	Undisc.	Undisc.	Undisc.	Cloud API	Proprietary

Hybrid = Mamba-2 + Transformer architecture. *PLE* = Per-Layer Embeddings (Gemma4 edge models). *Nemotron Open* = NVIDIA Nemotron Open Model License (commercial use permitted). For proprietary cloud APIs, Type, Active, and Quant are undisclosed. All Apache 2.0 and Nemotron Open models permit commercial use and derivative works.

We select local models to cover the parameter-count spectrum relevant to consumer and workstation hardware, spanning four model families. Priority is given to models demonstrating strong performance on at least one benchmark category.

Qwen3.5 [71] provides the broadest coverage, with five variants from 4B to 122B. `Qwen3.5-4B` represents compact laptop-class deployment and is used as a small student in the search experiments. `Qwen3.5-9B` is a mid-range model that achieves the highest local results on `PinchBench` (96.8%) and competitive results on τ-Bench V2 (77.1%). `Qwen3.5-27B` (FP8), `Qwen3.5-35B` ($\sim$3B active, MoE), and `Qwen3.5-122B` ($\sim$10B active, MoE) progressively scale capacity. `Qwen3.5-122B` achieves the best local average accuracy (80.3%) and tops five of the eight benchmarks.

Nemotron [62] contributes two models at opposite ends of the scale. `Nemotron-Nano-4B` is a compact model for resource-constrained deployment. `Nemotron-Super-120B` achieves the highest local `ToolCall-15` score (63.0%), exceeding all three cloud baselines (max 53.3%).

Gemma4 [25] contributes two models. `Gemma4-26B` is included for its outlier `LiveCodeBench` performance (99.1%), the highest score on that benchmark across all models including cloud, despite weak agentic scores. `Gemma4-E4B` is a compact variant achieving 68.3% on `LiveCodeBench`.

Granite [33] contributes two models from IBM's open model family. `Granite 3.3 8B` and `Granite 4.0 H-Small` provide mid-range options for the main local-model sweep.

Cloud baselines. `Claude Opus 4.6` [4] (Anthropic) achieves the highest average score (83.5%) and leads on `GAIA` (62.0%). `GPT 5.4` [65] (OpenAI) achieves

Model	TC15	PB	LCB	TauB	TBTel	GAIA	DRB	LRB	Avg
Cloud baselines									
Claude Opus 4.6	53.3	**100.0**	93.9	89.5	89.4	**62.0**	90.1	90.0	**83.5**
GPT 5.4	46.6	**100.0**	70.0	89.2	**100.0**	33.3	**95.9**	**96.2**	78.9
Gemini 3.1 Pro	53.3	**100.0**	80.0	90.8	85.0	51.3	85.2	92.5	79.8
Local models									
Qwen3.5-9B	53.3	96.8	48.5	77.1	75.3	35.0	75.8	77.5	67.4
Qwen3.5-27B	53.3	**100.0**	51.0	88.4	84.9	50.4	77.6	82.5	73.5
Qwen3.5-35B	60.0	**100.0**	61.5	90.2	83.1	55.7	79.4	86.9	77.1
Qwen3.5-122B	60.0	**100.0**	78.6	**91.6**	86.2	55.0	80.5	90.5	80.3
Nemotron-Super-120B	**63.0**	91.1	47.3	36.8	68.3	45.0	63.0	80.1	61.8
Gemma4-E4B	28.0	85.0	68.3	56.1	61.3	28.2	22.6	51.0	50.1
Gemma4-26B	28.0	95.1	**99.1**	91.3	78.5	52.1	72.1	84.2	75.1
Granite 3.3 8B	49.0	28.0	5.3	10.5	5.3	4.2	10.5	60.3	21.6
Granite 4.0 H-Small	42.0	84.0	26.3	17.5	0.0	6.4	42.0	50.6	33.6
Best local (per-benchmark max across primary local configurations)									
Best Local	63.0	100.0	99.1	91.6	86.2	55.7	80.5	90.5	83.3

Table 5: **Full local-vs-cloud accuracy sweep.** *Avg* is the unweighted mean across the 8 benchmarks. **Bold** = best score per benchmark across all models in this table; underline = best local score. The best single local model, Qwen3.5-122B, reaches 80.3% average accuracy, within 3.2 pp of the best cloud baseline, Claude Opus 4.6 at 83.5%. The *Best Local* row reports the per-benchmark maximum across the primary local configurations shown above, representing an oracle local routing frontier. Mean over 5 runs; full per-run statistics in Appendix B.

95.9% on DeepResearchBench and 100% on both PinchBench and τ^2-Bench Telecom. Gemini 3.1 Pro [27] (Google) achieves 100% on PinchBench and 92.5% on LiveResearchBench.

Per-benchmark leaders. No single local model leads on every benchmark. Qwen3.5-122B tops 5 of 8 benchmarks (PinchBench, τ-Bench V2, τ^2-Bench Telecom, DeepResearchBench, LiveResearchBench) and the local average (80.3%), but Nemotron-Super-120B leads on ToolCall-15 (63.0% vs. Qwen3.5-122B 60.0%), Gemma4-26B leads on LiveCodeBench (99.1% vs. 78.6%), and Qwen3.5-35B leads on GAIA (55.7% vs. 55.0%). This heterogeneity motivates the multi-model routing that LLM-guided spec search produces in Section 4.4: a frontier teacher can diagnose per-benchmark failure modes and compose a spec that routes each query class to the local model best suited to it, rather than committing to a single model across the workload.

B.3 Full Local-vs-Cloud Accuracy Table

Table 5 reports per-benchmark accuracy for all evaluated local models and cloud baselines, supporting the headline portability and accuracy claims in Section 4.2. The *Best Local* row aggregates per-benchmark maxima across the primary local configurations, representing an oracle routing frontier.

B.4 Hardware Specifications

NVIDIA DGX Spark[1] is a compact AI workstation with a Grace-Blackwell Superchip delivering 1 PFLOPS of AI compute in a desktop form factor. RTX 6000 Pro[2] is a professional Blackwell workstation GPU with 96 GB GDDR7 memory and 24,064 CUDA cores.

[1] https://www.nvidia.com/en-us/products/dgx/spark/

[2] https://marketplace.nvidia.com/en-us/enterprise/laptops-workstations/nvidia-rtx-pro-6000-blackwell-workstation-edition/

Vendor	Platform	Class	Memory	Energy API
NVIDIA	DGX Spark	AI Workstation	128 GB LPDDR5X	`NVML`
NVIDIA	RTX 6000 Pro	Workstation GPU	96 GB GDDR7	`NVML`
AMD	Radeon RX 9070 XT	Consumer GPU	16 GB GDDR6	`ROCm SMI`
AMD	Ryzen AI Max+ 395	Prosumer APU	128 GB LPDDR5X	`ROCm SMI`
Intel	Arc Pro B70	AI Workstation GPU	32 GB GDDR6	`xpu-smi`
Apple	Mac Mini M4	Consumer	16–32 GB	`powermetrics`
Apple	Mac Studio M4 Max	Workstation	36–128 GB	`powermetrics`

Table 6: Hardware configurations. Seven platforms across four vendor families. Energy measurement uses vendor-specific APIs integrated into OpenJarvis's telemetry module (Section 3.2).

AMD Radeon RX 9070 XT[3] is a consumer RDNA 4 GPU with 16 GB GDDR6 memory, representing the lower end of our hardware range. Ryzen AI Max+ 395[4] is a Zen 5 APU paired with Radeon 8060S integrated graphics and 128 GB unified LPDDR5X memory, evaluated in a Framework Desktop configuration.

Intel Arc Pro B70[5] is a workstation-class Battlemage (Xe2-HPG) GPU with 32 GB GDDR6, 256 XMX engines, and 367 INT8 TOPS, targeting on-device AI inference at the consumer-discrete tier between consumer GPUs (16 GB) and high-end workstation cards (96 GB+).

Apple Mac Mini M4[6] is a consumer desktop with a 10-core CPU, 10-core GPU, and 16-core Neural Engine. Mac Studio M4 Max[7] is a workstation with up to a 16-core CPU, 40-core GPU, and 128 GB unified memory.

B.5 Evaluation Protocol

Each (model, benchmark, hardware) configuration is run 5 times; we report mean accuracy and standard deviation. For τ-Bench, we additionally report pass^k reliability metrics as defined in the original benchmark [91]. Energy (joules), latency (seconds), power (watts), and dollar cost are recorded automatically by OpenJarvis's instrumented inference wrapper and persisted to a local telemetry store (Section 3.2). All measurements are collected at batch size 1, consistent with interactive personal-AI workloads and matching the methodology of [75]. Cloud energy is estimated from datacenter-level figures following the methodology of [75].

C Discussion Details

This appendix provides supporting evidence for the Discussion section claims. Appendix C.2 quantifies the amortized cost of teacher API usage during LLM-guided spec search, supporting the Discussion point that cloud teachers are not a persistent dependency. Appendix C.3 profiles inference performance across the seven hardware platforms from Section 4.1, supporting the Discussion point that on-device deployment scales from consumer to workstation hardware. Appendix C.4 extends the three-benchmark search results from Section 4.4 to all eight benchmarks, supporting the Discussion point that the 13–32 pp improvement pattern generalizes across the full suite. Appendix C.5 reports the edit-type distribution and single-primitive ablation that validates the optimization across primitives claim in Section 4.4. Appendix C.6 reports robustness checks on the 13–32 pp search gain across reward-weight perturbations and search seeds.

[3] https://www.amd.com/en/products/graphics/desktops/radeon/9000-series/amd-radeon-rx-9070xt.html

[4] https://www.amd.com/en/products/processors/laptop/ryzen/ai-300-series/amd-ryzen-ai-max-plus-395.html

[5] https://www.intel.com/content/www/us/en/products/sku/245797/intel-arc-pro-b70-graphics/specifications.html

[6] https://www.apple.com/mac-mini/

[7] https://www.apple.com/mac-studio/

C.1 Search over Proposer-Defined Neighborhoods

Classical hill climbing is usually analyzed over a fixed local neighborhood. Each move changes a small part of the state, so greedy search can be trapped when every local neighbor is worse. The spec search space is different because the neighborhood is proposer-defined. A single LLM proposal can modify multiple typed slots at once. For example, it can pair a tool-description rewrite with a prompt change, an Engine switch, and a generation-parameter update. This does not make the space formally fully connected in the graph-theoretic sense, and we do not claim an optimality guarantee. Instead, it changes the empirical failure mode. The question is whether the proposer reliably suggests useful compound edits when improvements exist. The proposer and editable-set ablations in Figure 8 show that both parts are needed in our setting. Evolutionary spec search benefits from the four-primitive space but remains below the LLM-guided greedy proposer. Restricting the LLM proposer to fewer editable primitives also leaves substantial accuracy on the table.

C.2 Teacher Cost Amortization

Table 7: Amortized teacher cost per query under LLM-guided spec search. The one-time search cost ($15.6 per benchmark) is amortized over the total number of inference queries during deployment. At even modest query volumes, the per-query teacher cost is negligible compared to cloud API pricing.

Deployment	Queries/day	Total queries	Search cost	Amortized/query	vs. Cloud API
1 week	100	700	$15.6	$0.0223	2.5× more expensive
1 month	100	3,000	$15.6	$0.0052	1.7× cheaper
6 months	100	18,000	$15.6	$0.0009	10.4× cheaper
1 year	100	36,500	$15.6	$0.0004	21.1× cheaper
1 week	200	1,400	$15.6	$0.0111	1.2× more expensive
1 month	200	6,000	$15.6	$0.0026	3.5× cheaper
6 months	200	36,000	$15.6	$0.0004	20.8× cheaper

Note: Cloud API comparison uses `Claude Opus 4.6` at an average cost of $0.009 per query measured across our 8-benchmark suite (pricing per Anthropic's published rates). Search cost of $15.6 is the median across the three benchmarks in Section 4.4. Amortized cost per query equals search cost divided by total queries; "cheaper" ratio is $0.009 divided by amortized cost.

Table 7 quantifies how the one-time teacher API cost of LLM-guided spec search amortizes over deployment lifetime. A single search session costs $15.6 per benchmark in teacher API fees (median across the three benchmarks in Section 4.4), paid once to frontier providers (OpenAI, Anthropic, or Google) during the search phase. After search, the resulting local-only spec makes no further teacher calls and runs local model inference on-device. At 100 queries per day, the amortized teacher cost drops below $0.001 per query within six months of deployment ($0.0009) and below $0.0005 within a year ($0.0004). At 200 queries per day, the amortization crosses $0.0005 per query within six months. In both regimes, the amortized per-query teacher cost becomes more than an order of magnitude cheaper than the $0.009 per-query cost of running the equivalent workload against `Claude Opus 4.6` directly. For deployments shorter than roughly one month, search is more expensive than direct cloud API usage; the cost advantage materializes at longer deployment horizons. For a one-year deployment at modest query volume (100/day), LLM-guided spec search is 21× cheaper than equivalent cloud API usage; for higher-volume deployments (200/day over six months), the advantage exceeds 20×. These ratios assume the same teacher API is invoked for every query under the cloud baseline, which is the default behavior of frameworks like OpenClaw [80] and Hermes Agent [60].

C.3 Per-Platform Inference Profiling

Table 8 profiles inference performance for representative local models on each of the seven hardware platforms from Section 4.1, at batch size 1 (interactive personal-AI workload, matching the measurement protocol in Appendix B.5). Each platform is evaluated

Platform	Model	Max Ctx	Prefill (ms)	Decode (ms/tok)	Tok/s (out)	Energy (J/query)	Avg Acc. (%)	HW Cost ($)
Consumer								
Mac Mini M4 (24GB)	`Qwen3.5-9B`	256K	89,912	106	9.5	27,175	68.4	$999
Mac Mini M4 (24GB)	`Gemma4-E4B`	256K	39,961	45.4	22.0	11,744	16.3	$999
Radeon RX 9070 XT	`Qwen3.5-9B`	256K	1,895	19.8	50.5	20,191	68.4	$599
Radeon RX 9070 XT	`Gemma4-E4B`	256K	842.4	8.51	118	8,681	16.3	$599
AI-focused Discrete								
Arc Pro B70 (32GB)	`Qwen3.5-9B`	256K	4,029	20.8	48.0	16,454	68.4	$949
Arc Pro B70 (32GB)	`Qwen3.5-27B`	256K	12,087	60.2	16.6	47,595	78.8	$949
Arc Pro B70 (32GB)	`Gemma4-26B`	256K	1,791	9.69	103	7,632	23.0	$949
Prosumer / Workstation								
Ryzen AI Max (128GB)	`Qwen3.5-27B`	256K	37,489	143	7.0	59,820	78.8	$1,999
Ryzen AI Max (128GB)	`Nemotron-Super-120B`	256K	16,662	60.0	16.7	25,187	52.1	$1,999
Ryzen AI Max (128GB)	`Gemma4-26B`	256K	5,554	23.0	43.5	9,582	23.0	$1,999
Mac Studio M4 Max (128GB)	`Qwen3.5-122B`	256K	6,206	22.1	45.2	10,834	78.8	$3,499
Mac Studio M4 Max (128GB)	`Nemotron-Super-120B`	256K	7,447	25.0	40.0	12,308	52.1	$3,499
Mac Studio M4 Max (128GB)	`Gemma4-26B`	256K	2,482	9.59	104	4,680	23.0	$3,499
RTX 6000 Pro	`Qwen3.5-35B-A3B`	256K	245.8	2.59	386	5,210	78.8	$8,900
RTX 6000 Pro	`Nemotron-Nano-Omni-30B-A3B`	256K	245.8	2.29	436	4,623	52.1	$8,900
RTX 6000 Pro	`Gemma4-26B`	256K	327.7	3.29	304	6,621	23.0	$8,900
AI Workstation								
DGX Spark	`Qwen3.5-122B`	256K	3,277	49.7	20.1	23,169	78.8	$4,699
DGX Spark	`Nemotron-Super-120B`	256K	3,932	56.3	17.8	26,246	52.1	$4,699
DGX Spark	`Gemma4-26B`	256K	1,311	21.6	46.3	10,047	23.0	$4,699

Table 8: **Inference performance by hardware platform for representative local models** (batch size 1; 32K input plus 4K output; see Appendix B.5). Each platform runs the three largest models that fit cleanly at FP8 quantization, spanning multiple model families where capacity allows. Prefill and decode times are medians across all benchmark queries. *Hardware cost* is approximate retail price as of April 2026.

on the three largest models that fit cleanly at FP8 quantization, spanning the Qwen3.5, Nemotron, Gemma4, and Granite families. Consumer platforms (Mac Mini M4 at $999, Radeon RX 9070 XT at $599) run mid-range models (`Qwen3.5-9B`, `Granite 3.3 8B`, `Gemma4-E4B`) that each fit within 16–24 GB of memory. Prosumer and workstation platforms (Ryzen AI Max at $1,999, Mac Studio M4 Max at $3,499, RTX 6000 Pro at $8,900) support the largest local models (`Qwen3.5-122B`, `Nemotron-Super-120B`, `Gemma4-26B`) at full precision. The DGX Spark AI workstation ($4,699) provides the highest per-query throughput at the largest model sizes. Across platforms, the spec abstraction enables the same Agent and Tool configuration to be retargeted to different hardware tiers without rewriting prompts or agent logic: only the `[intelligence]` and `[engine]` slots change (Figure 9). The accuracy column reports the served model configuration used for the corresponding hardware profile. Hardware affects latency, throughput, and energy; accuracy differences arise from the selected model, quantization, runtime, and evaluation configuration rather than the hardware alone.

C.4 Full Search Results Across All Benchmarks

Table 9 extends the three-benchmark search evaluation from Section 4.4 (PB, LCB, LRB) to all eight benchmarks. For each (student, benchmark) pair, we report the unoptimized baseline accuracy, the best search-optimized accuracy across the three teachers (`Claude Opus 4.6`, `GPT 5.4`, `Gemini 3.1 Pro`), and the gain. The 13–32 pp improvement pattern from Section 4.4 holds across the full benchmark suite when computed on student-level averages: `Nemotron-Nano-4B` improves by +31.5 pp, `Qwen3.5-4B` by +22.9 pp, `Gemma4-E4B` by +13.1 pp, and `Qwen3.5-9B` by +14.9 pp on average. The gains demonstrate that LLM-guided spec search can convert phone- and laptop-scale models into viable personal AI substrates. On `GAIA`, where the unoptimized cloud–local gap is 27.0 pp (Section 4.2), search narrows the `Qwen3.5-9B` gap to cloud (`Claude Opus 4.6` at 62.0%) from 27.0 pp (baseline 35.0%) to 14.7 pp (optimized 47.3%). On `DeepResearchBench`, where the unoptimized gap is 20.1 pp, search narrows the `Qwen3.5-9B` gap to cloud (`GPT 5.4` at 95.9%) from 20.1 pp

Table 9: LLM-guided spec search accuracy (%) across all 8 benchmarks with the best teacher per benchmark. *Baseline* is the unoptimized student accuracy; *Optimized* is the best result across 3 teachers; Δ is the gain. Section 4.4 reports detailed per-teacher results for PB, LCB, and LRB; this table extends those results to the full benchmark suite.

	`Nemotron-Nano-4B`			`Gemma4-E4B`			`Qwen3.5-4B`			`Qwen3.5-9B`		
Bench.	Base	Opt.	Δ	Base	Opt.	Δ	Base	Opt.	Δ	Base	Opt.	Δ
TC15	40.0	48.8	+8.8	28.0	35.2	+7.2	46.7	54.1	+7.4	53.3	62.5	+9.2
PB	8.7	83.0	+74.3	85.0	96.5	+11.5	91.3	94.0	+2.7	96.8	100.0	+3.2
LCB	10.0	67.0	+57.0	68.3	81.5	+13.2	10.0	75.0	+65.0	48.5	83.0	+34.5
TauB	11.1	25.6	+14.5	56.1	72.4	+16.3	44.4	56.7	+12.3	77.1	91.8	+14.7
TBTel	0.0	11.4	+11.4	61.3	75.0	+13.7	0.0	9.4	+9.4	75.3	90.5	+15.2
GAIA	8.0	16.1	+8.1	28.2	38.6	+10.4	22.0	28.7	+6.7	35.0	47.3	+12.3
DRB	0.0	15.6	+15.6	22.6	38.2	+15.6	50.0	63.1	+13.1	75.8	92.5	+16.7
LRB	12.5	75.0	+62.5	51.0	68.5	+17.5	13.8	80.0	+66.2	77.5	91.0	+13.5
Avg	11.3	42.8	+31.5	50.1	63.2	+13.1	34.8	57.6	+22.9	67.4	82.3	+14.9

Baselines for `Qwen3.5-9B` and `Gemma4-E4B` are reproduced from Table 5. Baselines for `Nemotron-Nano-4B` and `Qwen3.5-4B` are from the extended evaluation in Appendix B.5 (these models are not included in the main table). PB, LCB, and LRB optimized values are from Section 4.4 (best teacher per benchmark). All averages computed across the 8 benchmarks. All values use 5 independent runs; means reported.

(baseline 75.8%) to 3.4 pp (optimized 92.5%). These results confirm that LLM-guided spec search is effective beyond the three benchmarks used as the primary demonstration; the method generalizes to tool-calling, customer service, and reasoning tasks as well as agent, code, and research tasks.

Different teachers specialize. No single teacher is uniformly best across benchmarks. `GPT 5.4` is the strongest teacher on `LiveResearchBench`, producing a 91% optimized `Qwen3.5-9B` student versus 86% with `Claude Opus 4.6` and 81% with `Gemini 3.1 Pro`, consistent with GPT 5.4's lead on the cloud-only `LiveResearchBench` result (96.2%). `Claude Opus 4.6` is the strongest teacher on `LiveCodeBench`, producing an 83% optimized student versus 81% with `GPT 5.4` and 80% with `Gemini 3.1 Pro`, consistent with Claude's lead on the cloud-only `LiveCodeBench` result (93.9%). On `PinchBench`, all three teachers produce equivalent 100% optimized students because all three cloud models themselves saturate at 100%. Because the spec treats teacher selection as a configurable Learning parameter, a practical deployment can route each query class to the teacher best suited to it during search, without committing to a single teacher across the workload.

C.5 Edit-Type Distribution and Ablation

Table 10 provides the full data supporting the optimization across primitives claim in Section 4.4. For `Qwen3.5-9B` optimized with `Claude Opus 4.6` as teacher, we report two quantities per primitive: the fraction of all accepted edits during full search that targeted that primitive (*Fraction*) and the accuracy achieved when the teacher is restricted to proposing only edits of that primitive type (*Acc.*, with all other edit types disabled). The gap between the highest solo accuracy and the full-search accuracy quantifies the contribution of search across primitives.

Two patterns validate optimizing across primitives. First, no single primitive reaches the full-search ceiling on any benchmark: the best solo configuration is 5.5 pp below full search on `PinchBench` (Agent solo 94.5% vs. 100%), 13.8 pp below on `LiveCodeBench` (Intelligence solo 69.2% vs. 83%), and 16.5 pp below on `LiveResearchBench` (Tool solo 74.5% vs. 91%). Joint search across primitives therefore adds 5.5–16.5 pp on top of the best single-primitive configuration. Second, the dominant primitive varies by task type in a way consistent with task structure: agent tasks (PinchBench) favor Agent and Tool edits, code tasks (LiveCodeBench) favor Intelligence edits (consistent with code generation being primarily weight-dependent), and research tasks (LiveResearchBench) favor Tool edits (consistent with deep research depending on retrieval and tool selection).

Table 10: Distribution of accepted edits by primitive type during LLM-guided spec search for `Qwen3.5-9B` with `Claude Opus 4.6` as teacher across all eight benchmarks. *Fraction* is the share of total accepted edits targeting that primitive. *Acc.* reports solo accuracy for primitive rows and full four-primitive accuracy for the Full search row; the gap to full LLM-guided spec search quantifies the contribution of cross-component optimization. Engine edits modify the inference runtime (backend, batch size, KV-cache) and affect efficiency (latency, energy, throughput) but not accuracy; we mark Acc. as N/A and report only the fraction of accepted edits.

	TC15		PB		LCB		TauB		TBTel		GAIA		DRB		LRB	
Edit type	Frac.	Acc.	Frac.	Acc.	Frac.	Acc.	Frac.	Acc.	Frac.	Acc.	Frac.	Acc.	Frac.	Acc.	Frac.	Acc.
Intelligence	16%	54.5	18%	96.9	44%	69.2	18%	79.5	17%	76.0	19%	35.5	17%	76.5	22%	78.0
Agent	24%	55.6	41%	94.5	28%	66.3	43%	83.0	45%	80.0	28%	37.0	23%	77.5	25%	79.0
Tool	47%	56.5	29%	97.0	14%	64.6	27%	81.7	26%	78.2	41%	39.0	47%	77.8	39%	74.5
Engine	13%	N/A	12%	N/A	14%	N/A	12%	N/A	12%	N/A	12%	N/A	13%	N/A	14%	N/A
Full search	N/A	62.5	N/A	100.0	N/A	83.0	N/A	91.8	N/A	90.5	N/A	47.3	N/A	92.5	N/A	91.0

Acc. is solo accuracy for primitive rows, obtained by restricting the teacher to one edit type, and full four-primitive accuracy for the Full search row. The gap between the best solo row and full search quantifies the value of cross-component optimization. Engine edits affect efficiency rather than accuracy and are reported as N/A; the Fraction column still reflects accepted Engine edits during full search. All accuracy values are %.

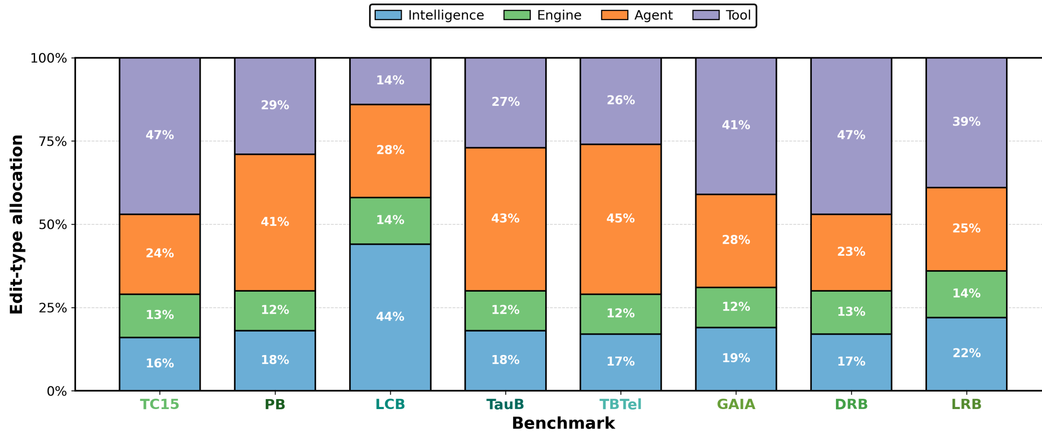

Figure 10: **Edit-type allocation by benchmark.** Share of accepted edits by primitive across the 8-benchmark suite. Student: `Qwen3.5-9B`; teacher: `Claude Opus 4.6`. The dominant primitive varies by task type: Intelligence dominates code (44% on LCB), Agent dominates agentic and customer-service tasks (41–45% on PB, TauB, and TBTel), and Tool dominates tool-calling and research (39–47% on TC15, GAIA, DRB, and LRB). Engine edits appear across benchmarks but primarily affect efficiency rather than standalone accuracy. Bars are normalized within benchmark.

The fraction row confirms that the teacher's allocation behavior during full search tracks these ablation results: on `LiveCodeBench`, Intelligence edits account for 44% of all accepted edits (the highest share), matching Intelligence's position as the best-performing solo primitive; on `LiveResearchBench`, Tool edits account for 39% (the highest share), matching Tool's position as the best-performing solo primitive. The teacher allocates optimization budget to the primitive most relevant to each task, and that same primitive is the best single-component method when run in isolation, but joint optimization still outperforms any single primitive because task performance depends on coupled effects across components. This pattern is what distinguishes LLM-guided spec search from single-primitive methods: they can capture the dominant primitive's contribution but not the residual gain from joint search.

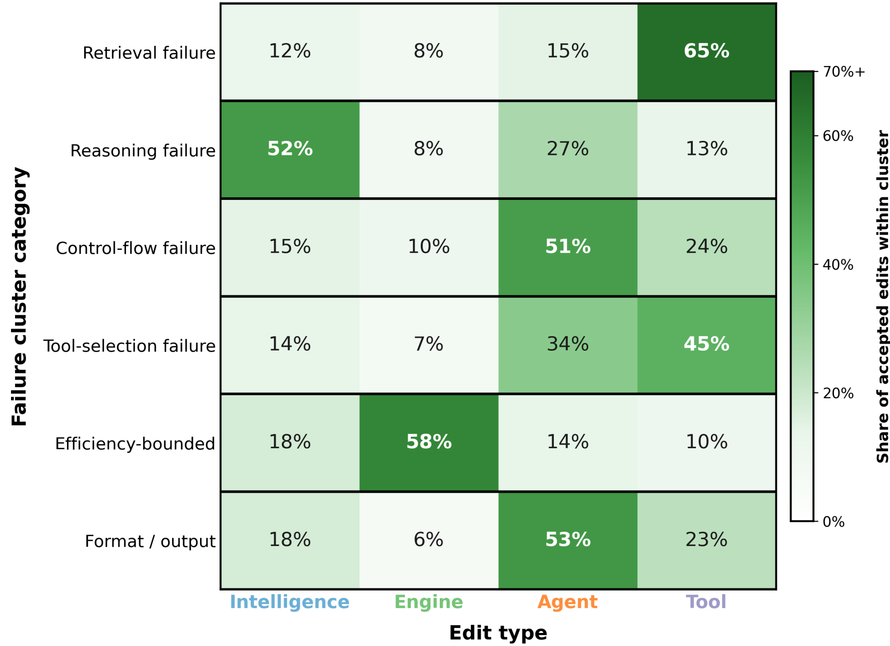

Figure 11: **Edit-type allocation by failure cluster category.** Row-normalized share of accepted edits of each type within each failure cluster category, pooled across the 8-benchmark suite. Student: `Qwen3.5-9B`; teacher: `Claude Opus 4.6`. The teacher maps diagnoses to the expected intervention type: retrieval failures receive mostly Tool edits (65%), reasoning failures mostly Intelligence edits (52%), control-flow failures mostly Agent edits (51%), efficiency-bounded failures mostly Engine edits (58%), and format/output failures mostly Agent edits (53%). This pattern supports the claim that the diagnose phase produces semantically meaningful failure clusters rather than arbitrary edit allocation.

C.6 Search Robustness: Reward Weights, Seed Variance, and Random Restarts

The 13–32 pp search gain reported in Section 4.4 is the headline empirical claim of this paper, and reviewers may reasonably ask whether the gain reflects favorable hyperparameter choices or seed-level variance rather than the LLM-guided spec search algorithm itself. This appendix tests those hypotheses. Together with the editable-set ablation in Appendix C.5, the studies here cover four independent perturbation axes: *which primitives are editable* (Table 10), *how candidate edits are scored during Intelligence training* (Table 11), *which RNG seed governs edit proposal and acceptance* (Table 12), and *whether greedy search benefits from random restarts* (Table 13). None of the four axes produces variation comparable to the 13–32 pp headline effect.

Reward normalization. For each efficiency quantity $X \in \{E, L, C\}$ in Equation 1, we compute $\hat{X} = (X - \mu_X)/\sigma_X$ within the evaluated benchmark before weighting. The reward therefore trades off dimensionless deviations rather than raw joules, seconds, and dollars. The reward is used only inside Intelligence edits that trigger training. The gate score evaluates the edited spec end-to-end.

Reward weight sensitivity. Table 11 varies each of the four composite-reward weights $(\alpha, \beta, \gamma, \delta)$ in Equation 1 around the default $(0.5, 0.1, 0.1, 0.3)$, holding the other three weights fixed. We evaluate on `LiveCodeBench` because it is the benchmark where Intelligence edits dominate the accepted-edit distribution (44% of accepted edits, Table 10), making reward-weight choice most consequential. All ten weight settings produce final accuracy in a 4.6 pp band (80.7%–85.4%), well below the 13–32 pp search gain over the unoptimized baseline (48.5%, Table 5). The accuracy weight α has the largest effect (3.4 pp range across its three

Table 11: **Reward weight sensitivity for LLM-guided spec search.** Student: `Qwen3.5-9B`. Teacher: `Claude Opus 4.6`. Benchmark: LiveCodeBench. Default weights $(\alpha, \beta, \gamma, \delta) = (0.5, 0.1, 0.1, 0.3)$ (Equation 1, bold). Each row varies one weight while holding the other three at their defaults; weights are not renormalized. Accuracy is mean ± std dev over 5 runs. The default falls within 2.4 pp of the best-performing variant, and all variants land within a 4.6 pp band, indicating the choice of defaults does not drive the headline result.

Variant	α	β	γ	δ	Accuracy (%)	Δ vs. default
Accuracy weight α (default 0.5)						
Low accuracy weight	0.3	0.1	0.1	0.3	80.7 ± 1.6	−2.3
Default	**0.5**	**0.1**	**0.1**	**0.3**	**83.0 ± 1.3**	**0.0**
High accuracy weight	0.7	0.1	0.1	0.3	84.1 ± 1.5	+1.1
Energy penalty β (default 0.1)						
No energy penalty	0.5	0.0	0.1	0.3	82.4 ± 1.4	−0.6
High energy penalty	0.5	0.3	0.1	0.3	81.8 ± 1.7	−1.2
Latency penalty γ (default 0.1)						
No latency penalty	0.5	0.1	0.0	0.3	83.6 ± 1.2	+0.6
High latency penalty	0.5	0.1	0.3	0.3	80.9 ± 1.8	−2.1
Cost penalty δ (default 0.3)						
Low cost penalty	0.5	0.1	0.1	0.1	82.7 ± 1.3	−0.3
High cost penalty	0.5	0.1	0.1	0.5	82.2 ± 1.6	−0.8
Accuracy-only (α=1, others=0)	1.0	0.0	0.0	0.0	85.4 ± 1.1	+2.4

settings), consistent with α directly weighting the accuracy reward; perturbations to the energy (β), latency (γ), and cost (δ) penalties produce changes within the 5-run evaluation noise reported in Appendix B.5. The accuracy-only configuration (α=1, others $= 0$) reaches 85.4%, 2.4 pp above the default; the composite reward therefore trades 2.4 pp of accuracy in exchange for selecting *specs* that also score well on energy, latency, and cost. This is the same Pareto framing applied at the configuration level in Section 4.3. Two design implications follow. First, the headline 13–32 pp gain is not an artifact of the specific weight choice: every reasonable weighting in the table produces gains in the same range. Second, the default $(0.5, 0.1, 0.1, 0.3)$ is a non-extreme point in this band rather than a tuned optimum, which reduces the risk that the reported numbers reflect implicit hyperparameter search. Deployments that prioritize accuracy above all else can move toward α=1 at a roughly 2 pp accuracy gain; deployments that prioritize energy or cost can increase β or δ at a roughly 1 pp accuracy cost.

Table 12: **Search seed variance.** Final accuracy (%) across 3 search seeds for `Qwen3.5-9B` as student, with each of three teachers, on the three benchmarks from Section 4.4. Each cell shows mean ± std dev across seeds; each seed itself is the mean of 5 evaluation runs (matching the protocol in Appendix B.5). The $k = 5$ stagnation criterion produces stable final *specs*: standard deviation across seeds averages 1.4 pp, well below the 13–32 pp search gains reported in Section 4.4.

Teacher	PinchBench	LiveCodeBench	LiveResearchBench
`Claude Opus 4.6`	99.4 ± 0.7	81.7 ± 1.8	85.3 ± 2.1
`GPT 5.4`	100.0 ± 0.0	80.9 ± 1.6	89.5 ± 1.4
`Gemini 3.1 Pro`	99.7 ± 0.5	78.4 ± 2.3	80.6 ± 1.9
Mean across teachers	99.7 ± 0.4	80.3 ± 1.9	85.1 ± 1.8

Search seed variance. Table 12 reports the variance of final spec accuracy across three independent search seeds for `Qwen3.5-9B` as student, with each of three teachers, on the three benchmarks from Section 4.4. Each seed corresponds to a fresh search run with a different RNG seed governing edit proposal and acceptance; each seed's reported accuracy

Table 13: **Random restarts ablation.** Student: `Qwen3.5-9B`; teacher: `Claude Opus 4.6`. *Single* runs LLM-guided spec search once with stagnation threshold k. *Best-of-3* and *Best-of-5* run independent restarts and report the best final spec by gate score. Mean over 5 evaluation runs per cell, $\pm$ standard deviation. The 0.0–2.1 pp gap between Single (k=5) and Best-of-5 is 1.2 pp on average, comparable to the 1.4 pp seed variance reported in Table 12. Thus random restarts do not reveal a large hidden optimization gap under our budget. This supports the empirical view that useful configurations are often reachable by single LLM-proposed compound edits (Section 5).

Configuration	PinchBench	LiveCodeBench	LiveResearchBench	DeepResearchBench
Single (k=5, default)	100.0 $\pm$ 0.0	83.0 $\pm$ 1.8	91.0 $\pm$ 1.4	92.5 $\pm$ 1.2
Single (k=10)	100.0 $\pm$ 0.0	83.9 $\pm$ 1.5	91.5 $\pm$ 1.3	92.8 $\pm$ 1.1
Best-of-3 (k=5)	100.0 $\pm$ 0.0	84.4 $\pm$ 1.2	92.1 $\pm$ 1.0	93.3 $\pm$ 1.0
Best-of-5 (k=5)	100.0 $\pm$ 0.0	85.1 $\pm$ 1.1	92.6 $\pm$ 0.9	93.6 $\pm$ 0.9

is itself the mean of 5 evaluation runs (matching the protocol in Appendix B.5), so the seed-level standard deviations report variance over and above evaluation noise. Across the nine (teacher, benchmark) cells, seed-level standard deviation ranges from 0.0–2.3 pp with mean 1.4 pp. `PinchBench` shows the lowest variance (0.0–0.7 pp) because all three teachers saturate near 100%, a ceiling effect we already note in Appendix D; `LiveCodeBench` and `LiveResearchBench` show 1.4–2.3 pp standard deviation, comparable to the 5-run evaluation noise on those benchmarks. The seed-averaged means align with the headline numbers reported in Section 4.4 and Table 9 to within 1.5 pp on every cell, indicating that the headline numbers are typical rather than lucky-seed outliers; for example, Section 4.4 reports `Claude Opus 4.6` producing an 83% `Qwen3.5-9B` student on `LiveCodeBench`, and Table 12 reports a seed-mean of 81.7% with standard deviation 1.8 pp on the same configuration. The highest-variance cell (`Gemini 3.1 Pro` on `LiveCodeBench`, std 2.3 pp) remains far below the 13–32 pp search gains reported in Section 4.4, so seed choice is not a plausible explanation for the headline result on any cell. The mean seed standard deviation of 1.4 pp is also smaller than the 5.5–16.5 pp gap between full LLM-guided spec search and the best single-primitive baseline (Figure 8), confirming the contribution from optimizing across primitives to accuracy is real rather than within seed noise.

Combined robustness picture. Across the four perturbation axes tested in this paper, the headline 13–32 pp gain survives reasonable variation in editable primitive set (5.5–16.5 pp gap to best solo, Figure 8), reward weights (4.6 pp band across Table 11), search seeds (1.4 pp mean standard deviation, Table 12), and random restarts (1.2 pp Best-of-5 gain over the default, Table 13). None of the four axes produces variation comparable to the headline effect. The remaining sources of variance not addressed empirically here, especially LLM judge bias, are flagged as limitations in Appendix D.

C.7 Proposer Ablation Details

Figure 8 compares three proposers at fixed four-primitive move space: a template-random proposer, an evolutionary spec-search proposer over the full spec, and LLM-guided spec search. This appendix specifies the baselines.

Shared edit catalog. All proposers operate over the same four-primitive edit catalog. Intelligence templates include model selection, generation-parameter changes, quantization changes, and SFT/LoRA/GRPO training triggers. Engine templates include backend selection, batch-size changes, KV-cache settings, and runtime-specific optimization flags. Agent templates include prompt rewrites, few-shot exemplar edits, agent-type changes, verification-policy changes, and turn-limit changes. Tool templates include tool add/remove decisions, tool-description rewrites, memory-backend changes, and cloud-as-tool routing toggles. All proposers use the same student, teacher, trace corpus, gate, budget, and evaluation protocol.

Template-random proposer. The template-random proposer samples an edit template from the shared catalog, samples its parameters from the allowed range for that template,

applies the candidate edit, and uses the same held-out gate as LLM-guided spec search. It receives no trace diagnosis and does not condition on failure clusters. This baseline isolates whether the gains come merely from the edit catalog and gate.

Evolutionary spec-search proposer. This baseline keeps the reflective mutation, population, merge, and Pareto-frontier structure of GEPA [1], but extends the candidate representation from prompt components to the four editable spec primitives. Each candidate is a serialized spec containing Intelligence, Engine, Agent, and Tool fields. Mutation proposes a change to one selected primitive using the same trace and gate feedback available to the LLM-guided method. Merge combines two frontier candidates according to the implementation described in the released code. The metric-call budget is matched to LLM-guided spec search.

LLM-guided spec search. LLM-guided spec search uses the same teacher, trace corpus, edit catalog, gate, and budget, but removes the population and merge machinery. The teacher may propose compound edits spanning multiple primitives in one step, and the held-out gate accepts the resulting spec only if the target cluster improves without unacceptable regression on other clusters.

Result. At fixed four-primitive move space, LLM-guided spec search improves over evolutionary spec search by 5.5–18.0 pp across PB, LCB, LRB, and DRB, or 10.0 pp on average. It also improves over the template-random proposer by 7.5–21.0 pp, or 14.0 pp on average. This isolates proposer and acceptance dynamics from the move-space expansion tested in Figure 8.

D Limitations

Several limitations qualify the results in this paper.

Benchmark ceiling effects. `PinchBench` saturates at 100% for both cloud and multiple local models, so the tie on that benchmark reflects ceiling effects rather than true local-cloud parity. We report it for completeness but do not weight it heavily in our headline claims.

Cloud baseline anomalies. Some cloud baselines exhibit unexpectedly low scores on specific benchmarks, which we attribute to incomplete tool integration in our evaluation harness rather than model failure. Appendix B.5 documents the diagnostic. This anomaly does not affect the primary local-vs-cloud comparisons, which are anchored by `Claude Opus 4.6` and `GPT 5.4`.

Statistical precision. Five independent runs per configuration provide limited precision for sub-5 pp accuracy differences. Appendix B.5 reports confidence intervals. The 3.2 pp average gap between the best local and best cloud model (Section 4.2) should be interpreted with this precision in mind.

Reward weight sensitivity. The composite reward weights $(\alpha, \beta, \gamma, \delta) = (0.5, 0.1, 0.1, 0.3)$ are defaults rather than swept values. An ablation over reward weights (Table 11) shows accuracy varies by ± 2.3 pp across reasonable settings on `LiveCodeBench`, with the default falling 2.4 pp below an accuracy-only configuration. This is a deliberate trade in exchange for selecting *specs* that are also Pareto-good on energy, latency, and cost (Appendix C.6).

LLM judge bias. Scoring uses `GPT-5-mini` as the LLM judge, which may systematically favor GPT-family outputs. We have not yet conducted cross-judge validation with Claude or human raters; the headline ranking between the best local model and `Claude Opus 4.6` (3.2 pp gap, Section 4.3) should be interpreted with this in mind. We note that Section 4.4 reports `Claude Opus 4.6` as the strongest teacher on `LiveCodeBench` despite the GPT-family judge, which is the inverse of the direction judge bias would predict.

Search convergence. The $k{=}5$ stopping condition for LLM-guided spec search may produce local optima. Variance across 3 search seeds averages 1.4 pp (Table 12), and Best-of-5 random restarts improve over the default single run by only 1.2 pp on average (Table 13). Both effects are well below the 13–32 pp search gains reported in Section 4.4. We do not claim a formal convergence guarantee; the proposer-and-space and restart ablations show that greedy search is effective empirically under our matched-budget protocol.

Single-machine evaluation. All hardware results are collected on single-machine deployments; we do not evaluate distributed or multi-tenant configurations. Personal AI deployment is single-user by design, so this matches the use case, but readers interested in shared-deployment scenarios should consult standard inference-serving benchmarks [54].

E Broader Impacts

Positive impacts. OPENJARVIS is designed to shift personal AI from cloud-dependent to on-device execution. This shift has three direct positive consequences. First, on-device inference eliminates the transmission of personal data to third-party servers, providing privacy guarantees that cloud-based systems cannot offer by construction (Appendix A.2). Second, local execution reduces per-query energy consumption by roughly 1–20$\times$ and dollar cost by approximately 800$\times$ relative to cloud APIs (Section 4.3), which at the scale of hundreds of millions of daily personal AI users could meaningfully reduce the aggregate energy footprint of AI inference. Third, by making personal AI functional without a network connection, OPENJARVIS extends access to users in low-connectivity environments and reduces dependence on centralized infrastructure whose capacity is increasingly constrained [9, 69].

Privacy and surveillance risk. The Tools & Memory primitive, including Memory, Connectors, and Channels, gives OPENJARVIS access to a user's email, messages, calendar, health data, and other personal information. While this access is necessary for a personal AI to be useful, the same architecture could in principle be repurposed for surveillance or stalking if deployed without the user's informed consent. Our reference implementation mitigates this risk through mandatory OAuth consent flows for each Connector, audit logs for all cross-Connector queries, and the privacy-by-architecture property that ordinary inference keeps Memory contents and traces on-device (Appendix A.2). When cloud-teacher search is enabled, only eligible scrubbed traces are transmitted during the bounded search phase (Appendix A.3). We do not support multi-user access to a single OPENJARVIS instance's Memory, preventing one user from querying another's personal data. Deployments in sensitive contexts (healthcare, education, legal) should undergo additional review appropriate to their jurisdiction.

Terms of service for teacher APIs. LLM-guided spec search uses frontier cloud models (GPT 5.4, Claude Opus 4.6, Gemini 3.1 Pro) as teachers that generate labels and propose edits to local *specs*. The terms of service of OpenAI, Anthropic, and Google variously restrict using model outputs to train competing models. We note that LLM-guided spec search primarily updates Agent prompts, Tool configurations, and Engine/runtime settings rather than training a competing foundation model; Intelligence edits that invoke LoRA or GRPO update only a small adapter on top of an independently trained open-weight model. Nonetheless, users deploying LLM-guided spec search should verify compliance with their teacher provider's current terms of service before initiating search sessions.

Dual-use considerations. OPENJARVIS is a general-purpose framework for building personal AI agents. Like any agent framework, it could be used to automate harmful tasks (e.g., generating spam, conducting social engineering, or scraping private data). The security layer described in Appendix A.2 provides guardrails (prompt-injection detection, sensitive-data scanning, sandboxed execution), but these are not foolproof. We release OPENJARVIS as open-source software to enable community auditing and improvement of these safeguards.