Mobile AI Factory: Building Specialized Language Models for Your Mobile Device

Evan Li*, Krishna Deshpande*, Wen Cheng* and Shyamnath Gollakota
Paul G. Allen School of Computer Science and Engineering, University of Washington
{evanly, kdeshpan, wencheng, gshyam} @cs.washington.edu
*co-primary authors

Abstract

Small language models (SLMs) running on mobile devices can keep user data private and work without connectivity. But they perform well only when fine-tuned for the target task and sized for the target device, a process that today requires repeated expert tuning. We present *Mobile AI Factory*, an agentic system that builds a fine-tuned, quantized SLM for a given task and phone. Given a task specification and a target phone, an orchestrator LLM chooses whether to tune hyperparameters, mine real data, or generate synthetic examples, while fixed rules roll back changes that hurt accuracy. The factory searches from small to large models across base models and quantizations, adapts its accuracy goal as it searches, and uses real measurements of the target phone across inference engines and backends. We evaluate Mobile AI Factory on ten tasks and four Android phones. On an unseen testset, its on-device models outperform two five-shot frontier models on five tasks, by up to 0.34, and match them on three more. Its search comes within 0.03 of exhaustively training all 27 model variants on an entity extraction task, while using 19× less GPU time. Phone measurements also matter: the best configuration changes across phones. Without phone measurements, Claude Opus 5.5, given the full hardware and model specifications, misses the app's requirements in 25% of easy and moderate cases and in 60% of hard ones. Code is available at: https://github.com/evanly-gh/SLM_Factory.

1 Introduction

Small language models that run directly on mobile devices could enable numerous applications. For example, a keyboard app could rewrite a hastily typed message into a more professional response, an email client could summarize messages and draft replies, and a voice assistant could help users with motor impairments fill out forms and navigate applications. Running these models on the device, rather than in the cloud, has two key advantages. First, sensitive data, like a patient's medication schedule or a customer's bank details, can remain on the device. Second, applications can operate in scenarios where connectivity is limited or intermittent, such as translating a conversation on a hiking trail or providing navigation when a vehicle loses network access.

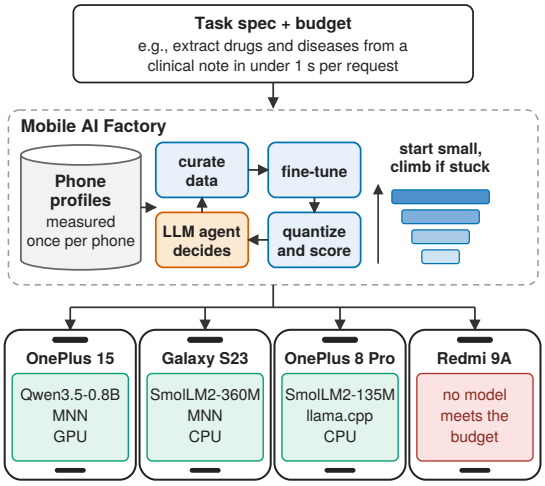


Figure 1: Mobile AI Factory. It takes a task specification and a budget, and builds a specialized model for a phone.

Smartphones now include capable CPUs and GPUs, and mobile inference engines such as llama.cpp and MNN can run quantized SLMs at practical speeds [12]. The harder problem is building the models themselves. Each application needs a model that is trained for its task and small enough to run on the target phone. Even with training data available, developers still need to fine-tune a base model, iterate on accuracy, decide on the model size and quantization level, and test the resulting model on the target phone.

This process must be manually repeated by a human expert for every phone model, which makes it impractical for all but the largest companies. The Facebook app alone runs on more than ten thousand smartphone and tablet models spanning many distinct SoCs [56]. New phones are released every year, and roughly two new open foundation models are released every week [38]. A configuration that performs well on one phone may perform poorly, or fail entirely, on another. For example, in our measurements, llama.cpp GPU prefill is 5.4× faster than CPU prefill on the OnePlus 15, but 7.4× slower on the OnePlus 8 Pro. Similarly, Qwen3-4B at 8-bit precision decodes 12.6–13.8 tokens/s on the OnePlus 15, but only 0.7–4.8 tokens/s on the OnePlus 8 Pro.

We ask the following question: given a task specification and a target phone, can we automatically produce a fine-tuned, quantized SLM that meets both the task's accuracy

goal and the phone's resource budget? The developer inputs the specification, which includes any available data sources, the evaluation metric, a verifier for the model's outputs, optionally an accuracy goal, and the app's resource requirements, such as prefill and decode speed, memory, and energy. This is difficult because we cannot know in advance which model will meet these requirements while achieving the best accuracy. Further, we may not know how large a model a task needs until we train it.

We present *Mobile AI Factory*, an agentic system that builds specialized SLMs. Its core is a loop that curates training data, fine-tunes and evaluates a model, then decides what to try next. An orchestrator LLM chooses one intervention per iteration, while fixed rules handle rollback, budgets, and model scaling. Three components support this loop: a teacher model that generates synthetic data, a held-out-data agent that prepares a held-out set before training and shares only aggregate results with the orchestrator, and a phone profiler that benchmarks models across quantization levels, runtimes, and backends once per phone, reusing the results across tasks to identify configurations that meet app requirements.

At its core, our factory uses three key techniques.

• **Agentic search with rollback and adaptive accuracy goals.** We model each training attempt as a pipeline of dataset, hyperparameters, and learning strategy. At each iteration, the orchestrator chooses one intervention: tuning hyperparameters (LoRA rank, scaling, learning rate, weight decay, and number of epochs), mining more real data (if available) from the task's sources, or having the teacher generate synthetic examples aimed at the failure categories that hurt accuracy most. A contamination check removes any row that also appears in the held-out test set. Since most interventions do not help, any change that lowers accuracy is rolled back to the best model so far, and the orchestrator tries something different. Finally, a fixed accuracy target may be too easy for some tasks and unreachable for others. Instead we set the goal to the teacher model's five-shot score on the task (zero-shot where five examples do not fit its context), with a floor of 0.8. If the factory reaches the goal quickly or with little effort on the current model, the orchestrator can choose to raise it and continue its search. This lets it keep improving on tasks where the goal was set too low.

• **Tiered search from small to large models.** A model's size depends on both its parameter count and its quantization: a 2B model at 4 bits is smaller than a 16-bit 0.8B model. We group all combinations of base model and quantization into N buckets by size, and start with the smallest bucket. Within a bucket, the orchestrator picks which combination to train. If accuracy stops improving before reaching the goal, the factory moves to the next bucket. This avoids two extremes: letting the orchestrator pick one combination from the whole pool, which gives no guarantee of a small model, and training every combination, which is too expensive.

• **Profiling the target phone.** Whether a model meets the app's requirements cannot be reliably determined from its specifications; it must be measured on the phone. The phone profiler measures prefill and decode speed, peak memory, and energy on the target phone for all combinations of base model, quantization, runtime (llama.cpp or MNN), and backend (CPU or GPU). Each phone is profiled once, and the profile is reused across all tasks. The profile also determines how to run the final model, since the best runtime and backend differ across phones, models, and quantization levels. The factory can then search only over models that meet the app's requirements on that phone, and the model it outputs is far more likely to run within them.

We implement Mobile AI Factory using Claude Sonnet 5 as the orchestrator. Each run uses two NVIDIA L40S GPUs: one trains and scores the model, and the other hosts a local teacher model (Qwen3.6-35B-A3B) that generates synthetic data. The model pool contains nine base models from 135M to 4B parameters, from the SmolLM2, Gemma 3, Qwen3, and Qwen3.5 families, each at 4-bit, 8-bit, and 16-bit precision, for 27 combinations. We group these into five buckets by size on disk, with boundaries at 300 MB, 750 MB, 1.5 GB, and 2.5 GB. We profile four Android phones, ranging from a current flagship to a budget phone whose GPU the inference engines cannot use due to the lack of OpenCL: the OnePlus 15, Galaxy S23, OnePlus 8 Pro, and Redmi 9A. We evaluate the factory on ten tasks: biomedical entity extraction, fine-grained entity extraction, calendar event extraction, function calling, command parsing, 151-way intent classification, spam detection, emotion detection, grammatical error correction, and dialogue summarization.

Our results are as follows:

• On an unseen test set, the factory's small models outperform both Claude Sonnet 5 and DeepSeek v4 Flash, given five examples, on five tasks, by 0.024 to 0.342, and match them on three more. The gains are largest where the output format is strict: on biomedical entity extraction, a 517 MB model scores 0.828, while Claude scores 0.652 with five examples.

• With the phone in the loop, the factory returns models whose profiled cost on the phone meets every limit. On a Galaxy S23 whose limits rule out the factory's best model for call summaries, it returns a model within 0.013 of it.

• Phone measurements cannot be replaced by reasoning. Given every specification but no real measurements, Claude Opus 5.5, with high reasoning effort enabled, misses the app's requirements in a quarter of easy and moderate cases, and in 60% of hard ones. It misjudges run-time memory and the backend, neither of which appears on a spec sheet.

• After the first fine-tune, search adds a median of 8 points on the smallest models and 2 points on the largest. On biomedical entity extraction, bucketed search comes within 3 points of training all 27 combinations at a nineteenth of the GPU time. When a goal is reached within a few iterations, the orchestrator raises it; in seven of the nine runs where a raise led to more training, the model improved, by 3.6 points.

Contributions. (1) We present Mobile AI Factory, to our knowledge the first agentic system that builds a fine-tuned, quantized SLM for a given task and a given phone. In contrast to optimizing accuracy on a single model [3], our factory searches from small to large models and uses measurements from the target phone. (2) We profile four Android phones across 364 configurations (104 per phone with a GPU, 52 on the Redmi 9A) of model, quantization, runtime, and backend, and show that the best configuration differs across phones, models, and quantization levels. (3) We evaluate the factory on ten tasks and show that, on unseen test sets, its on-device models outperform two five-shot frontier models on five tasks and match them on three more, and that, with the phone in the loop, it returns models whose profiled cost on the phone meets every limit.

2 Related Work

Agentic and Automated Fine-Tuning. Classical AutoML automates model selection and hyperparameter search over predefined spaces [6, 14, 16, 30, 35, 49], but cannot change the training data. MLE-bench [11] and Kaggle-style agents [25, 43] evaluate agents on end-to-end modeling tasks. AutoML-Agent [50] proposes multiple LLM agents to go from a task description to a working model, but not one optimized to a particular device. Prompt optimizers [2, 28, 60] instead improve the prompts given to a fixed model.

Over the past year, researchers have begun using agents to fine-tune language models [31]. PostTrainBench [45] studies whether agents can post-train base language models under a fixed compute budget. Pioneer Agent [3], the closest work, acquires data, builds evaluation sets, and trains small language models by jointly searching over data, hyperparameters, and learning strategy. Other work uses agents to curate or process training data for a fixed student model [23, 37], or to generate synthetic data aimed at the student's weaknesses [62]. Tools such as Slemify [4] automate a fixed pipeline of synthetic data generation, fine-tuning, and quantization for models served on CPU servers.

These systems share a key assumption: the model they produce will run on a server, so accuracy is the only objective. On a phone, this assumption breaks in four ways. First, a search that maximizes accuracy favors the largest model. Our factory instead treats the base model and its quantization level as a single choice and searches from small to large, moving to a larger model only when the current one stops improving. Second, a model evaluated at full precision may not be able to run on the phone; 4-bit quantization can significantly lower the accuracy: on the same weights, MNN's default 4-bit export loses up to 0.19 (§3). Our factory therefore scores every model after quantization. Third, the same model can meet the app's requirements on one phone and fail them on another, since the best runtime and backend differ across phones, and even a frontier model given full specifications picks a configuration that fails the app's limits in 60% of hard cases (§5.3). Our factory uses a measured profile of the target phone to decide which models to search and how the final model runs. Finally, tasks differ widely in difficulty, so a fixed accuracy target is either unreachable or trivial. Our factory sets its goal from a teacher model and raises it when the goal is met easily.

Hardware-Aware Model Design. Hardware-aware neural architecture search includes device cost in the search objective, either by measuring latency on real phones [48] or by estimating it with predictors and lookup tables [10, 55]. Other work specializes networks for different devices [9] or for microcontrollers [33]. nn-Meter [63] predicts DNN inference latency, and [20, 51] search quantization and compression policies with hardware feedback. These methods design architectures or compression policies for a given dataset, rather than adapting pretrained language models to a new task. In contrast, we fine-tune pretrained language models for a new task and treat quantization as part of the search. We also measure each configuration on the phone instead of predicting it, since run-time memory and the best backend differ in ways a predictor or a specification sheet may miss.

Recent work also gives LLM agents feedback from hardware, to write embedded firmware [32, 46, 58, 61, 64] or generate inference engines [26]. These systems optimize the software around a model, or preserve a model that already performs the task. Our factory instead builds the task capability itself, for language models on phones.

Adapting Models to Devices. NestDNN [15] builds multi-capacity models that trade accuracy for resources at runtime, LegoDNN [19] scales models block by block to fit each device, and AdaptiveNet [54] adapts a deployed model to diverse edge environments. These systems start from a model that already performs the task and adjust its size. Our factory starts from a task specification, builds the task capability through fine-tuning and data, and chooses the model, its precision, and its runtime from measurements of each phone.

On-Device LLM Inference and Measurement. A number of engines run quantized language models on phones, including llama.cpp [12], MNN [24, 53], MLC-LLM [40], and ExecuTorch [39]. Research systems further reduce latency using mobile NPUs [57] and activation sparsity [59], and

MobileLLM [36] designs sub-billion-parameter models for phones. Prior work also measures LLM performance, memory, and energy across phones and frameworks [29, 42], profiles models on real devices in device farms [44], and selects a backend for each device at runtime [7, 39]. These systems run or measure an existing model. In contrast, our factory builds the model, and uses per-phone measurements to decide how it runs on each target phone.

Training Data and Quantization. Prior work shows that data generated by a teacher model lets small models perform well beyond their size [1, 41, 52], and that a small set of high-quality examples can outperform a larger one [66]. Small models can outperform much larger models [21, 65]; our factory automates this process and targets it at a phone. Later work targets this data at the student's failures [27, 62], filters it for quality [5], and warns that self-generated data can narrow a model's outputs [47]. Our factory uses synthesis as an intervention, anchored to real training examples and filtered by programmatic checks before training. Similarly, models are typically fine-tuned with LoRA [22] and quantized after training [13, 17, 34]. In contrast, our factory makes quantization part of the search, choosing between a larger model at lower precision and a smaller one at higher precision, and scores both after quantization. Unlike systems that fine-tune on the phone [18], our factory runs on a server.

3 Motivation for hardware profiling

Deploying a small language model on a phone requires choosing an inference engine, a backend (CPU or GPU), and a quantization. These choices depend heavily on the phone.

Measurement setup. We benchmark 26 of the 27 model variants that Mobile AI Factory searches over: nine base models from 135M to 4B parameters, each at 4, 8 and 16 bits, except the 16-bit Qwen3.5-4B, which exceeds the RAM of two of our four phones. We run them on four phones that span five years: a OnePlus 15 (Snapdragon 8 Elite Gen 5, 16 GB RAM), a Galaxy S23 (Snapdragon 8 Gen 2, 8 GB), a OnePlus 8 Pro (Snapdragon 865, 12 GB) and a Redmi 9A (MediaTek Helio G25, 4 GB). Each variant runs on llama.cpp [12] and MNN [24], on the CPU and the GPU; the Redmi 9A's GPU lacks OpenCL, so it runs on the CPU only.

We follow the protocol of [8]: a 256-token prompt, then 256 generated tokens, four CPU threads, and three repetitions after one warm-up. Every run starts with the battery temperature at or below 28 °C (32 °C on phones that idle above it), with the CPU clocks uncapped, on battery, in airplane mode with Wi-Fi left on for adb, and with the screen off; the OnePlus 8 Pro stays awake with only its panel off, since it caps its CPU when it sleeps. Energy is the whole phone's battery energy, read from the fuel gauge every 100 ms, minus idle power. At 4 bits, the profiles use Q4_0 for llama.cpp, since

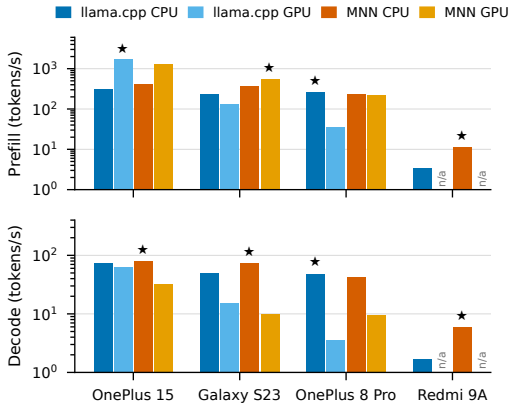


Figure 2: Qwen3-0.6B at 4 bits on four phones, two inference engines and two backends (CPU/GPU). The star marks the fastest option on each phone.

the pinned build's OpenCL backend has no Q4_K_M kernel, and MNN's default export; the factory scores accuracy on the Q4_K_M and HQQ exports of the same weights (§4.3.3). We attempted 364 configurations (104 on each phone with a GPU, 52 on the Redmi 9A), and 288 ran to completion. Repeated runs differ by 0.1–5% within a session (up to 38% for MNN's CPU prefill on the OnePlus 15), while GPU decode throughput varies by 10–20% between sessions, small next to differences of up to 94×.

Key Findings. Our findings are as follows:

- **The same model runs 42× slower on one phone than on another.** Qwen3-0.6B on llama.cpp's CPU backend at 4 bits generates 72 tokens/s on the OnePlus 15, 50 on the S23, 48 on the OnePlus 8 Pro, and 1.7 on the Redmi 9A (Fig. 2). Prompt processing spreads further, from 309 tokens/s to 3.3, a 94× gap. A gap between a flagship and a budget phone is expected. More surprising is how much the three flagships disagree on everything else.

- **The best backend on one phone can be the worst on another.** For Qwen3-0.6B with llama.cpp, GPU prefill is 5.4× faster than CPU prefill on the OnePlus 15, 1.7× slower on the S23, and 7.4× slower on the OnePlus 8 Pro (Fig. 3a). A developer tuning on the OnePlus 15 would put prefill on the GPU, and make it more than seven times slower on the OnePlus 8 Pro. The best backend also depends on the stage. On the OnePlus 15, the best GPU configuration prefills a median 3.4× faster than the best CPU one, while the CPU decodes faster for 14 of the 18 model–precision pairs below 2B parameters. On the S23, MNN decodes SmolLM2-135M at 224 tokens/s on the CPU but 9 on the GPU, a 25× gap.

- **Engine and precision rankings flip.** MNN's CPU backend decodes Qwen3-0.6B 1.1× faster than llama.cpp on the OnePlus 15, 1.5× on the S23 and 3.4× on the Redmi 9A, but slightly slower on the OnePlus 8 Pro (Fig. 3b). Moving from

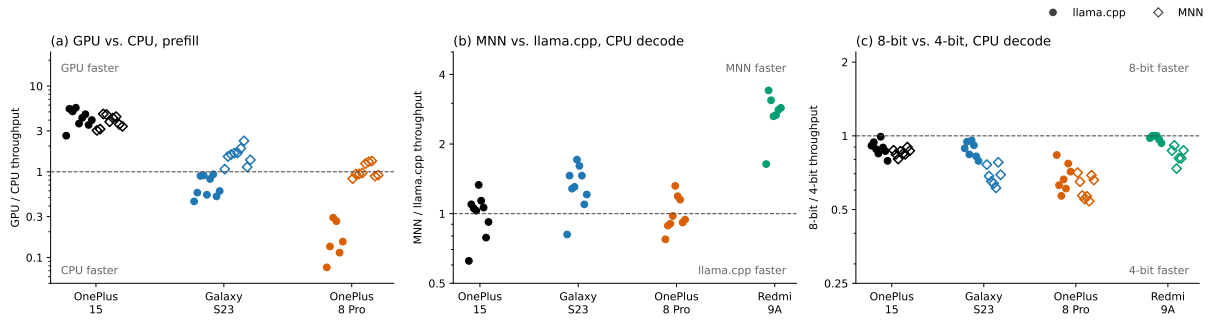


Figure 3: Each point is one model; the dashed line marks a tie. (a) GPU prefill throughput relative to the CPU on the same phone. (b) MNN decode throughput relative to llama.cpp on the CPU, at 4 bits. (c) 8-bit decode throughput relative to 4-bit on the CPU. Whether a rule holds depends on the phone.

Weights	llama.cpp Q4_K_M	MNN default	MNN HQQ-32
SmolLM2-360M, fine-tuned	0.767	0.659	0.747
Qwen3.5-0.8B, fine-tuned	0.864	0.850	0.859
Qwen3-1.7B, fine-tuned	0.886	0.851	0.875
Qwen3-4B, fine-tuned	0.915	0.897	0.918
SmolLM2-360M, zero-shot	0.114	0.104	0.119
Qwen3.5-0.8B, zero-shot	0.372	0.303	0.320
Qwen3-1.7B, zero-shot	0.538	0.351	0.557
Qwen3-4B, zero-shot	0.743	0.708	0.745

Table 1: Macro-F1 on CLINC150 for the same weights under three 4-bit recipes. All three are labelled 4-bit. Run-to-run noise is about ±0.015.

Phone	Model	Engine, backend	F1
Budget: 1 s			
OnePlus 15	Qwen3.5-0.8B, 4-bit	MNN, GPU	0.835
Galaxy S23	SmolLM2-360M, 4-bit	MNN, CPU	0.744
OnePlus 8 Pro	SmolLM2-135M, 4-bit	llama.cpp, CPU	0.719
Redmi 9A	none (fastest option takes 10 s)		
Budget: 4 s			
OnePlus 15	Qwen3.5-2B, 8-bit	llama.cpp, GPU	0.859
Galaxy S23	Qwen3.5-2B, 8-bit	MNN, GPU	0.859
OnePlus 8 Pro	Qwen3.5-0.8B, 4-bit	MNN, CPU	0.835
Redmi 9A	none		

Table 2: The most accurate deployment of the entity-extraction model for each phone. When several engines and backends reach the same accuracy, we list the fastest.

4 to 8 bits with llama.cpp on the CPU costs at most 21% of decode speed on the OnePlus 15 and the S23, but 16–43% on the OnePlus 8 Pro (Fig. 3c). On the S23's GPU, Qwen3-0.6B even decodes faster at 8 bits (26.5 tokens/s) than at 4 (14.9), likely because llama.cpp keeps a 4-bit file's output layer at 6 bits and the S23's OpenCL kernel for that format is slow.

- **Inference engines can change accuracy.** Both engines offer 4-bit formats, but MNN's default export rounds every linear layer to the nearest 4-bit value in blocks of 64 weights, while llama.cpp's Q4_K_M searches a scale and offset for each block of 32 and keeps the most sensitive layers at 6 bits. On the same weights, scored on 151-way intent classification (Table 1), MNN's default drops a fine-tuned SmolLM2-360M from 0.767 to 0.659, and a zero-shot Qwen3-1.7B from 0.538 to 0.351. HQQ with 32-weight blocks brings six of the eight cells within noise of llama.cpp or above it. Since MNN decodes 4-bit models fastest on three of our four phones, a factory has to score accuracy on the artifact each engine will run, not on a generic quantized checkpoint.
- **Spec sheets do not reveal which configurations will run.** Of the 364 configurations, 76 did not run. Weight size predicts 35 of these. The other 41 failed for reasons a spec sheet does not show: 20 MNN 16-bit GPU runs crashed, 10 swapped although their weights fit in RAM, 8 hit GPU-driver gaps, and 3 failed to load. The OnePlus 8 Pro's driver lacks OpenCL symbols the newer llama.cpp needs, which rules out llama.cpp on the GPU for every Qwen3.5 model. RAM is a poor guide too: the OnePlus 8 Pro has more RAM than the S23, yet runs Qwen3-4B at 8 bits at only 0.7–4.8 tokens/s, against 12.6–13.8 on the OnePlus 15. File size also underestimates memory. On the S23, llama.cpp's GPU backend holds a median 2.8× the model's file size, and up to 8.1× for the smallest model, since it allocates GPU buffers for weights, KV cache and workspace while keeping a CPU copy of the weights.

Implications for Mobile AI Factory. To see what this means for a developer, we take the biomedical entity-extraction task and its accuracy for the 26 profiled variants, and ask: if a request (a 256-token prompt and a 48-token answer) must finish within a fixed time, what is the most accurate model each phone can run? Under a 1 s budget, each phone needs a different model, on a different engine and backend, and accuracy ranges from 0.719 to 0.835 (Table 2). Under a 4 s budget, the OnePlus 15 and the S23 both reach the best accuracy any variant achieves on this task, through different engines.

These measurements shape three decisions in Mobile AI Factory. First, the configuration cost has to be measured on each phone; the measurements do not depend on the task, so

we run them once per phone and reuse them for every model the factory builds. Second, accuracy has to be measured on the exact artifact the target engine will run. Third, the right model size cannot be fixed before the phone is known.

4 Mobile AI Factory Design

Our factory takes two inputs: a task specification and a target phone. It returns a fine-tuned, quantized model for that phone, together with the inference engine and backend.

4.1 Agentic Architecture Overview

Fig. 4 shows our two-stage system. The setup stage, run once per task, looks up the target phone's measured profile, removes every configuration that exceeds the developer's budgets on that phone, and picks the inference engine and backend for each remaining variant. The search loop runs until the goal is met or the budget runs out.

The orchestrator LLM decides which intervention to try next, which model to try within a tier, and whether to raise the accuracy goal. It does not decide when to stop, when to roll back, or when to move to a larger model; those are deterministic rules. We split the roles this way for two reasons. First, a run calls the orchestrator most often when it is stuck, so the rule that notices it is stuck should not depend on the orchestrator. Second, most of the orchestrator's ideas do not work: in the run with the most iterations (nested intent parsing, 109 evaluations, 104 of them interventions), 88% of the interventions made the model worse and were rolled back. The rules ensure steady progress.

Each run uses three models: the orchestrator, a local teacher that writes synthetic training data, and the small model being built for the phone.

4.2 Setup Stage

The setup stage finds the configurations the phone can run, writes a brief for the teacher model, and builds the evaluation set.

4.2.1 Phone Profiles and Feasibility. We measure each phone once, using the protocol in §3. A configuration is one of the model variants run on one engine (llama.cpp or MNN) and one backend (CPU or GPU). For each configuration, the profile stores prefill and decode throughput, peak memory, and energy per token; configurations that crash, run out of memory, or are rejected by the GPU driver are stored as failures.

The developer gives budgets for a typical request: latency, memory and energy. The factory computes each configuration's latency from its measured throughput and the request's prompt and output lengths, and drops configurations that exceed a budget or failed on this phone. For each surviving variant, it keeps the engine and backend that decode fastest on that phone.

The factory groups the feasible variants into five tiers by their size on disk, with boundaries at 300 MB, 750 MB, 1.5 GB and 2.5 GB. A 2B model at 4 bits (1.3 GB) thus shares the third tier with a 0.6B model at 16 bits (1.4 GB). Of the 27 variants, 6, 5, 5, 4 and 7 fall in tiers 1 to 5. A phone profile only removes variants; it does not change the tier boundaries.

4.2.2 Task Analysis and the Brief. The orchestrator reads a sample of training rows and writes a short brief describing the task, what a correct answer should look like, and the mistakes most likely to make an answer incorrect. It never sees the held-out set. The brief is included in every prompt that asks the teacher to generate or check a training row.

4.2.3 Evaluation Setup. The factory draws up to 1,000 rows, balanced across labels for classification tasks, and freezes them as the held-out set. The set is scored on every iteration, so we cap it. Four firewalls check for overlap with the held-out set, by exact match after normalization, at four points: (1) when a dataset is loaded, where training rows that repeat a held-out row are dropped and any overlap that remains stops the run; (2) on every row mined or generated later; (3) again after quality control, which can rewrite rows; and (4) through the label set, which is fixed from the held-out rows so that no later step, including the teacher, can add a label. The trainer also keeps its own validation split for choosing checkpoints, so the held-out set is never seen during training. The held-out set guides every search decision, so its scores can read slightly high; §A measures this on unseen test sets.

To label difficulty, the factory runs the smallest and the largest base models over the held-out set, without fine-tuning. A row is *easy* if both answer it correctly, *medium* if only the larger one does and *hard* if neither does. This measures the gap the tiered search aims to close. A fixed accuracy target such as 0.9 may be trivial for some tasks and out of reach for others, so the factory scores the teacher, prompted with five real training examples, on the held-out set through the task's own scorer and sets the goal to that score, with a floor of 0.80 and a cap of 0.99. The run starts in the lowest tier.

4.3 Search Loop

The loop decides which model to train (§4.3.1), what to change (§4.3.2) and whether a change helped (§4.3.3).

4.3.1 Small-to-Large Search. The factory starts in the cheapest tier and moves up only when that tier stops improving.

Escalation rule. The factory escalates when the score has not risen by more than 0.02 over the last 15 evaluations, or after 30 evaluations in one tier without reaching the goal. Both counts include rolled-back attempts. On escalation, the

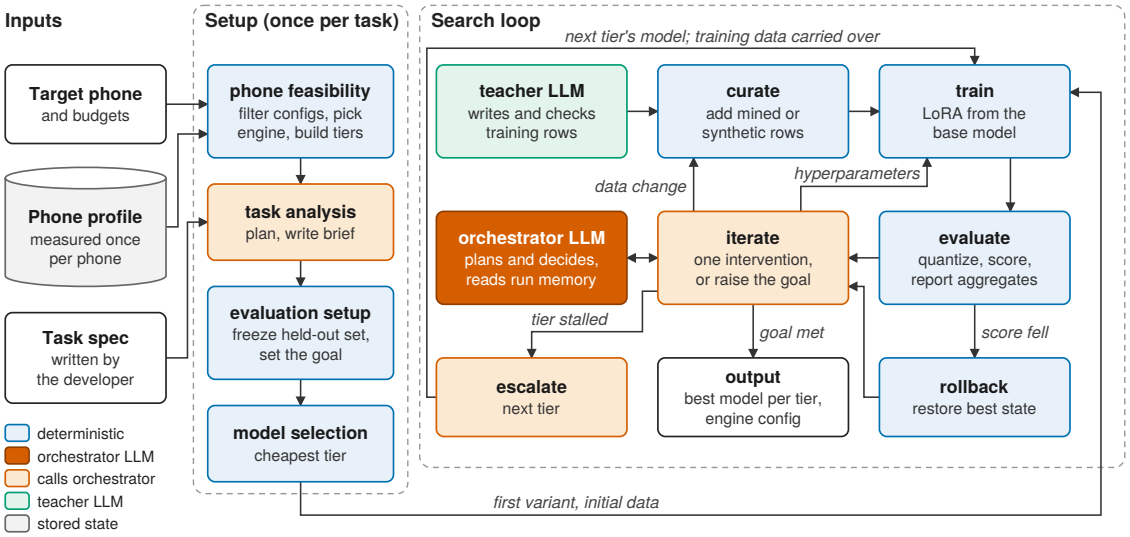


Figure 4: Mobile AI Factory. A setup stage reads the phone's measured profile and builds a frozen evaluation set; a loop then curates data, trains, scores the quantized artifact and decides what to do next. Dark orange is the orchestrator LLM, and light orange nodes call it. Blue nodes, including rollback and the decision to escalate, are deterministic code.

orchestrator picks one variant in the next tier. The run keeps its training data, and the rest of the search starts over.

Increasing the goal. When the run reaches its goal, the factory stores the result and asks the orchestrator whether to raise the goal, by at least 0.005 above the highest goal so far and to at most 0.99. A raised goal sends the run back into the loop. The orchestrator may lower the goal when failures look like a model size limit, but only after a raise, and never below the starting goal.

Output. The run returns the best model in each tier it visited, with its engine and backend. Since the search climbs from the cheapest tier, the developer reads off the cheapest one that meets their bar. If no tier does, the run returns its best model and why it stopped: tiers, steps or time ran out.

4.3.2 The Orchestrator and Its Interventions. In each iteration, the orchestrator proposes exactly one change.

Input and output. The orchestrator sees the evaluator's report (the overall score, accuracy on easy, medium and hard rows, and the most common failure categories from the task's own scorer, such as wrong entity boundaries or a call to an undeclared function), and the run memory, built from the search graph, which lists every attempt, whether it was kept, and what has failed since the last improvement. It returns a JSON object, naming the change, its parameters and a hypothesis. The factory validates it, and replaces a hyperparameter setting already tried on the same data with an untried neighbor.

Hyperparameters. A hyperparameter intervention holds the data fixed and adjusts up to five values: LoRA rank, the ratio of α to rank, weight decay, learning rate and epochs. Batch shape follows from the GPU and dropout is fixed at zero.

Data. A data change adds rows and keeps the best hyperparameters so far. Rows come from two sources. *Mining* first reads further into the task's own sources, and searches the web for new datasets only when those are exhausted, giving up after two empty rounds. *Targeted synthesis* has the teacher write 25–600 rows per failure category, in proportion to its failure count, and drops a category whose count does not fall. Synthesis passes three gates. (1) The teacher must score at least 0.80 on the task with five real examples, the way synthesis prompts it, or synthesis is off for the run; our main runs bypass this gate on seven tasks (§5.1). Examples matter: on biomedical entity extraction the teacher scores 0.13 without them and 0.32 with five. (2) Each row passes a programmatic check where one exists, for example that every extracted entity appears verbatim in its text. (3) The teacher can check each row against the brief (§4.2.2), using only real examples as references; our runs turn this check off where a strict programmatic check exists and, on the other tasks, record its verdicts without dropping rows. The training set only grows; rows leave only through quality control, the firewall, or the rollback of the change that added them.

Rollbacks. Three mechanisms limit each failure to one iteration. Training always restarts from the base model, so rollback exactly restores the previous behavior. Control then returns to the orchestrator, which must try a different change rather than repeat the same regression. A watchdog stops

runs that stop making progress, like three consecutive synthesis rounds that add nothing or a training set that shrinks by 25% (or 1,000 rows) in one step. We require repeated or large failures so the watchdog does not trigger on noise; we added it after one run spent eight hours on eight data changes that added no examples.

4.3.3 Scoring Each Iteration. The setup stage freezes a held-out evaluation set that training cannot access (§4.2.3). The loop adds three rules.

- Each iteration merges the adapter, exports 4- and 8-bit models to their deployment format, loads them in the target engine, and scores them on the held-out set; 16-bit models are scored in Transformers. For MNN, we use 4-bit HQQ with 32-weight blocks because its default quantization loses up to 0.19 accuracy (§3).
- The evaluator returns scores, failure counts, and the fraction of parseable outputs, but not the evaluation rows. This separates wrong answers from malformed outputs. Any decision that quotes a held-out row is rejected.
- The base model is the first candidate, and any fine-tuned model that scores worse is discarded, so the loop can conclude that fine-tuning does not help. A failed evaluation is recorded as missing, not zero. An early run used zero and incorrectly credited fine-tuning with a gain of 0.85.

4.4 Implementation Details

Mobile AI Factory's loop is a LangGraph state machine. The orchestrator is Claude Sonnet 5, called through its API, and the teacher is Qwen3.6-35B-A3B, served locally with vLLM. Training uses LoRA adapters on 4-bit base models, through Unsloth, PEFT and Transformers, with loss computed on the answer tokens only and the checkpoint chosen on a validation split. Each run uses two NVIDIA L40S GPUs on a SLURM cluster: one trains and scores the model, and the other serves the teacher. Scoring exports each 4- or 8-bit model with llama.cpp or MNN's own tools and runs it in that engine on the same GPU. A run can take several days, longer than a cluster job is allowed to run, so the run state is checkpointed to SQLite after every step, and a job nearing its time limit checkpoints and requeues itself.

5 Evaluation

5.1 Setup

Tasks. We use ten tasks. Five are bound by their output format: biomedical entity extraction (BC5CDR), calendar extraction to JSON, nested intent parsing (TOPv2), entity extraction over 33 fine-grained types (MultiCoNER) and function calling (BFCL). In four, the label cannot be read off the input: 151-way intent classification (CLINC150), spam detection, 28-way emotion classification (GoEmotions) and grammatical error correction (BEA-19). In the last, dialogue summarization (DialogSum), the base model can already do the task and fine-tuning mainly teaches the answer style. Each task starts from 2,600–5,000 real training rows. Its held-out set has 1,000 rows, except for DialogSum (500), Calendar JSON (535) and MultiCoNER (871).

Factory configuration. The model pool holds the 27 variants from §4.2.1: nine base models from 135M to 4B parameters, each at 4, 8 and 16 bits, from three families: SmolLM2 (135M, 360M), Gemma 3 (270M) and Qwen (Qwen3 at 0.6B, 1.7B and 4B; Qwen3.5 at 0.8B, 2B and 4B). Every run in this section uses the full pool and the five size tiers of §4.2.1, and scores every 4- and 8-bit model as a llama.cpp file (Q4_K_M or Q8_0) on the training GPU; 16-bit models are scored in Transformers. §5.3 then runs the factory with the phone profiles. The orchestrator is Claude Sonnet 5 and the teacher is Qwen3.6-35B-A3B. Every run starts from LoRA rank 16, α 32, learning rate 2×10^{-4}, weight decay 0.01 and 3 epochs.

Goal settings. Every task runs with the factory's default goal, the teacher's five-shot score with a floor of 0.80 (§4.2.3); for intent classification, whose five examples do not fit the teacher's context, its zero-shot score. To show how far the factory can go when a developer asks for more, we also ran intent classification (CLINC150) and spam detection with the goal pinned at 0.99, so that the search continues through every tier. Table 3 reports these runs in their own block. The per-tier analyses below (§5.2.1–§5.2.5) use the ten *main runs*: the default-goal runs for eight tasks and these two pinned runs, which cover the full ladder. We also enabled synthesis on every task, including seven where the teacher scored below the factory's 0.80 gate (§4.3.2), so we can measure its contribution on all ten (§5.2.2).

Baselines. For every task we score Claude Sonnet 5, DeepSeek v4-Flash, the teacher, and the untrained student. We called both APIs with reasoning disabled, via their batch interfaces.

For each tier of a run, we report three scores: the untrained model, its *first fine-tune* on the training data the run holds when it reaches that tier, and its *best* score after search. The *search gain* is best minus first fine-tune: what the iterative search bought on top of plain fine-tuning. A run's *best model* is the highest-scoring model across all tiers.

Since a run can take days, each configuration is a single run. All scores are on the held-out set. Two models trained identically on the same data differ by up to 0.015 on this set, so we treat smaller differences as ties. The effects we report are larger: on the seven tasks the factory wins under its default goal, its margin over the better frontier model is 0.037–0.357, over the better of zero- and five-shot.

Below, we evaluate the factory first phone-agnostic (§5.2) and then phone-aware (§5.3).

Task	Metric	Claude 0-shot / 5-shot	DeepSeek 0-shot / 5-shot	Untrained	First FT	Factory	Factory model	MB
Output-format bound								
BC5CDR	span F1	0.165 / 0.606	0.128 / 0.508	0.000	0.705	**0.830**	Qwen3.5-0.8B Q4	517
Calendar JSON	arg. match	0.514 / 0.634	0.505 / 0.669	0.036	0.535	**0.811**	Qwen3.5-0.8B Q4	517
TOPv2	exact match	0.241 / 0.242	0.145 / 0.085	0.003	0.534	**0.599**	Qwen3-4B Q8	4,082
MultiCoNER	micro F1	0.491 / 0.587	0.428 / 0.521	0.284	0.604	**0.653**	Qwen3-4B Q8	4,082
BFCL	arg. match	0.879 / 0.875	**0.896** / 0.876	0.818	0.834	0.852	Qwen3.5-4B Q4	2,655
Label not directly readable from the input								
CLINC150	macro F1	0.921 / **0.933**	0.907 / 0.904	0.114	0.770	0.884	SmolLM2-360M Q4	258
SMS Spam	minority F1	0.954 / **0.961**	0.856 / 0.832	0.235	0.878	0.913	SmolLM2-360M Q4	258
GoEmotions	Ekman F1	0.506 / 0.481	0.448 / 0.436	0.378	0.598	**0.598**	Qwen3-4B Q8	4,082
BEA-19	ERRANT $F_{0.5}$	0.476 / 0.506	0.518 / 0.528	0.435	0.536	**0.565**	Qwen3-4B Q8	4,082
In distribution								
DialogSum	ROUGE-L	0.334 / 0.424	0.327 / 0.374	0.272	0.482	**0.482**	Qwen3-4B Q8	4,082
Goal pinned at 0.99								
CLINC150	macro F1	0.921 / 0.933	0.907 / 0.904	0.766	0.901	**0.937**	Qwen3-4B Q8	4,082
SMS Spam	minority F1	0.954 / **0.961**	0.856 / 0.832	0.000	0.707	**0.961**	Qwen3-1.7B Q4	1,223

Table 3: Factory models against frontier models, on each task's own scorer and the same frozen held-out set. Frontier models are given no examples (0-shot) or five worked examples (5-shot). The three student columns describe the factory's best model: untrained, after its first fine-tune, and after search. Q4 and Q8 are 4- and 8-bit weights; MB is the model's file size. Bold marks the best score in each row; on the last two rows, the factory and Claude are within our 0.015 noise floor. GoEmotions is scored as macro F1 over its Ekman emotion groups.

5.2 Evaluating the Factory's SLMs

Table 3 compares the factory's best models with two frontier models, given no examples or five. Against the better of the two, the factory wins seven tasks under the default goal, by 0.037 to 0.357. With the goal pinned at 0.99, it matches Claude on intent classification and spam detection, within our noise floor. On unseen test sets the factory never saw (§A), 5 of these wins hold and 3 tasks are ties.

Our local teacher, which sets each run's goal, behaves the same way. Without and with five examples, it scores 0.132 and 0.321 on biomedical extraction, 0.204 and 0.624 on calendar extraction, 0.023 and 0.018 on nested parsing, 0.431 and 0.538 on fine-grained entity extraction, 0.866 and 0.856 on function calling, 0.864 and 0.850 on spam detection, 0.423 and 0.432 on emotion classification, 0.501 and 0.509 on grammar correction, and 0.284 and 0.339 on dialogue summarization. On intent classification it scores 0.877.

On tasks frontier models already do well, the margins are small. With the default goal, the factory stops as soon as a model meets the goal, so these runs end early, on the smallest tier with a 258 MB model: intent classification at 0.884 after 14 iterations and 9.6 hours, and spam detection at 0.913 after five iterations and 0.7 hours, both below Claude. With the goal set to 0.99, the search continues, and the factory matches Claude. On these tasks, matching a frontier model takes a 1.7B or 4B model, and a search that is allowed to keep going. The goal, not the factory, sets this ceiling.

On function calling (BFCL), the untrained 4B model scores 0.818, and fine-tuning and search together add 0.034, of which the search after the first fine-tune adds 0.018. It ends 0.027 below Claude and 0.044 below DeepSeek. Function calling is a format models are trained heavily on, so there is little left for fine-tuning to teach.

5.2.1 Decomposing the Gains. Table 3 reports, for the best model on each task, the untrained score, the score after the first fine-tune, and the final score. The first fine-tune accounts for a median of 82% of the total improvement over the untrained model. On biomedical entity extraction, it raises the score from 0.000 to 0.705, and the search adds another 0.125. On emotion classification and dialogue summarization, the first fine-tune is already the best result. When a run moves to a larger model, it carries over the training data collected so far, so the larger model's first fine-tune already includes data added during the search at smaller tiers.

To measure search on its own, we compare each tier's best score with that tier's first fine-tune, across the ten main runs. This gives 42 pairs, after excluding two tiers whose 270M model produced parseable output on fewer than 20% of rows.

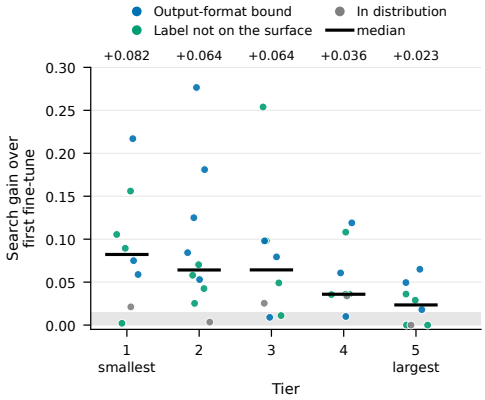


Figure 5: What search adds over each tier's first fine-tune. Each dot is one tier of one task's run. The grey band is the 0.015 improvement threshold.

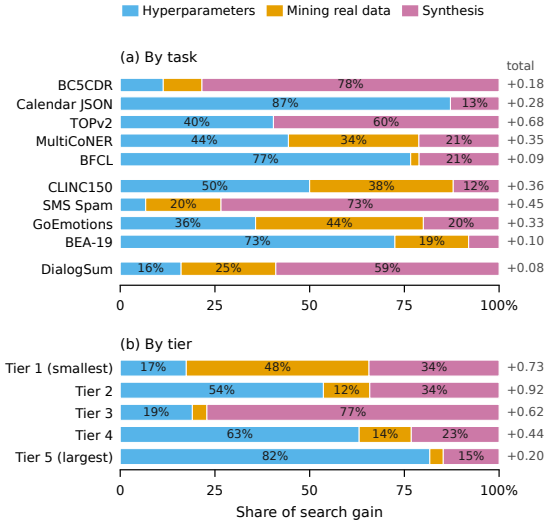


Figure 6: Search gains split across interventions, (a) per task and (b) per tier. Each bar sums the gains of the kept steps of each kind; the number on the right is the total.

Search improved on the first fine-tune by more than 0.015 in 34 of them, and by a median of 0.051. The gain shrinks as the model grows (Fig. 5). On the smallest tier, search adds a median of 0.082 and closes 21% of the remaining gap to a perfect score; on the largest, it adds 0.023 and closes 9%.

5.2.2 Contribution of Each Intervention. Fig. 6 shows that across all tiers, hyperparameter changes carry 41% of search gains, synthesis 40% and mining 19%. One tier inflates synthesis: on spam detection, synthetic rows repaired a 1.7B model whose first fine-tune had collapsed to 0.707. Without it, synthesis carries 35%. Which intervention matters depends on the task (Fig. 6a): synthesis carries most of the gain on four tasks, hyperparameters on three, and all three contribute on the rest.

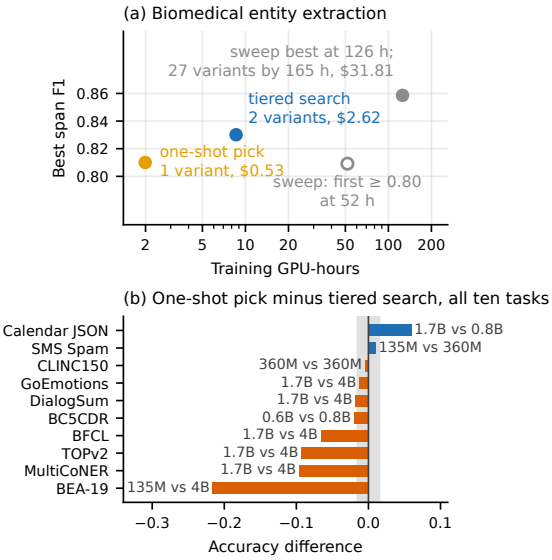


Figure 7: (a) Best score against GPU-hours for three search strategies on biomedical entity extraction, with orchestrator spend. (b) One-shot pick compared against the best model from the tiered search on each task, labelled with [picked model vs tiered winner]. The grey band is the noise floor.

Mining carries 48% of the gain on the smallest tier, synthesis 77% on the third, and hyperparameters 82% on the largest (Fig. 6b). This does not mean small models need data and large ones need tuning. First, mining runs out early, so later tiers fall back on the other two. Second, training data carried to a larger model improves its first fine-tune, which falls outside the search. Third, a gain goes to whichever step happened to be kept, so a different order could split the same total differently.

On seven tasks the teacher scored below 0.80, which by default turns synthesis off (§4.3.2). We allowed synthesis on those runs anyway. They account for 67% of all the gain credited to synthesis, mostly on nested parsing and biomedical extraction. The same runs suggest the gate is too strict. It tests whether the teacher can do the task, but the teacher's job is to write training rows. On nested parsing, the teacher scores 0.02 with five examples, yet 90% of the rows it writes pass the task's check, and synthesis carried 60% of that task's search gain. Writing a correct example is easier than solving a new one, so a better gate would test the teacher as a writer.

5.2.3 Tiered Search versus Baselines. We compare the tiered search with two baselines. An *exhaustive sweep* trains all 27 variants with the same loop; we ran it on biomedical entity extraction only, given the compute cost. A *one-shot pick* has the orchestrator choose the cheapest variant it expects to reach the goal, and searches only that model; we ran it on

all ten tasks. We report GPU-hours on the training GPU and orchestrator spend; the teacher's GPU is counted in §5.2.5.

The exhaustive sweep took 165 GPU-hours and $31.81, and first reached the 0.80 goal after 52 hours (Fig. 7a). The tiered search reached 0.830 in 8.6 hours for $2.62, a nineteenth of the GPU time. The sweep's best model, a 2B at 8 bits, scored 0.859. But the tiered search stopped as soon as it met its goal, on a 0.8B model whose best in the sweep was 0.835. Across ten tasks with the default goal, the one-shot pick used 60 GPU-hours and $11.68, against the tiered search's 286 hours and $45.39. In return, the tiered search scored higher on 8 of 10 tasks, by more than noise on six, and by 0.066–0.216 on four, a median gain of 0.019. Its winning model is a 4B on six tasks; the one-shot pick never chose one (Fig. 7b).

5.2.4 Raising the Goal. When a run meets its goal, the orchestrator can raise it. Fig. 8 shows that the orchestrator used both speed and margin. It raised the goal 8 of 9 times when the goal was reached within three iterations, but stopped 9 of 10 times after six iterations. In nine runs, a raise led to further training, and seven of those runs ended higher, by 0.036 on average. On runs that stayed on one model, the best score arrived within 1–7 evaluations of the raise. All nine extensions together took 14.6 hours on the training GPU. In two further runs, the raised goal was already below the score just reached, so the run finished at once. Only once did a raise change the model: in an earlier run of biomedical extraction, the higher goal moved the run from a 0.8B to a 2B, for a gain of 0.020. The gains come from three tasks: spam detection, calendar extraction and biomedical extraction. In the other main runs, the search kept climbing toward the goal through the largest tier (on two by design, with the goal pinned at 0.99), so raising the goal never came up.

5.2.5 Cost of the Orchestrator. Across the ten main runs, the orchestrator cost $57.37, or $2.62–8.46 per task. Nearly all of it, 98%, goes to the decision made each iteration, at a median of $0.064. The cost is the same across tasks and tiers, since the prompt summarizes the run in the same way whatever model it trains, so a run's bill grows with its number of iterations, not with model size. This is small next to training. Each run held two GPUs, one for training and one for the teacher, for 663 GPU-hours in total.

5.3 Phone-Aware Factory Deployment

We evaluate phone-aware deployment using profiles from four phones: a OnePlus 15, Galaxy S23, OnePlus 8 Pro, and Redmi 9A. On the one phone shared with prior work [8], our measurements match within 10% for GPU prefill and within a median of 5% for decode energy.

We deploy our ten tasks in several settings (§C). Each deployment specifies limits on memory, latency, throughput

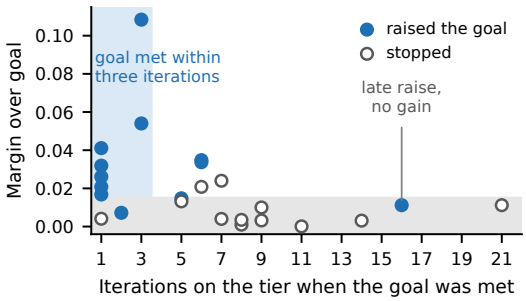


Figure 8: Every decision to raise the goal or stop, by how many iterations the tier had taken to meet the goal and by how much it beat it. The grey band is the noise floor, set at 0.015.

and energy. The deployment type sets memory and latency, the task determines required prefill and decode speeds, and an allotted share of daily battery use sets the per-request energy budget. Across four phones, this gives 152 requirements. We classify a requirement as *easy* if at least 60% of configurations satisfy it, *moderate* if 30–60% do, and *hard* if fewer than 30% do, and study 20 of each.

Fig. 9a,b shows two representative runs, with the phone in the loop. Fig. 9a shows biomedical entity extraction on a OnePlus 8 Pro. The requirement allows 5,800 MB of memory, needs at least 5 tokens/s of decode and 3 tokens/s of prefill, and caps each request at 15.7 J. The phone's profile kept 14 of the 27 variants. The run took 8.2 hours and $3.05 of API calls. Fig. 9b builds a call summarizer for a Galaxy S23. The requirement allows at most 1,024 MB, needs 350 tokens/s of prefill to start answering quickly, and allows 90 J per request. Only 11 of the 27 variants meet this. Without the phone, the factory's best model is a 4B at 8 bits, at 0.482 ROUGE-L. It needs 4,082 MB and cannot run here. With the phone, the factory searched only the 11 variants that fit and returned Qwen3-0.6B at 8 bits, on MNN's CPU backend, at 0.469. That is 0.013 below the model that does not fit, inside our noise floor. The run took 33.2 hours and $2.38.

This experiment also asks whether a configuration can be chosen without measuring the phone. Each requirement is constructed so that at least one configuration meets it, and every pick, from either method, is checked against the same measured profile. A method that selects from measurements meets every requirement by design; we include it as a reference that shows a feasible configuration exists. The question is how often a selection made without measurements finds one.

Comparing all the phone runs with the phone-agnostic runs, the factory's most accurate model meets 16 of 20 easy requirements, but only 6 moderate and 3 hard ones (Fig. 9(c)); the rest require a smaller model. The best configuration also

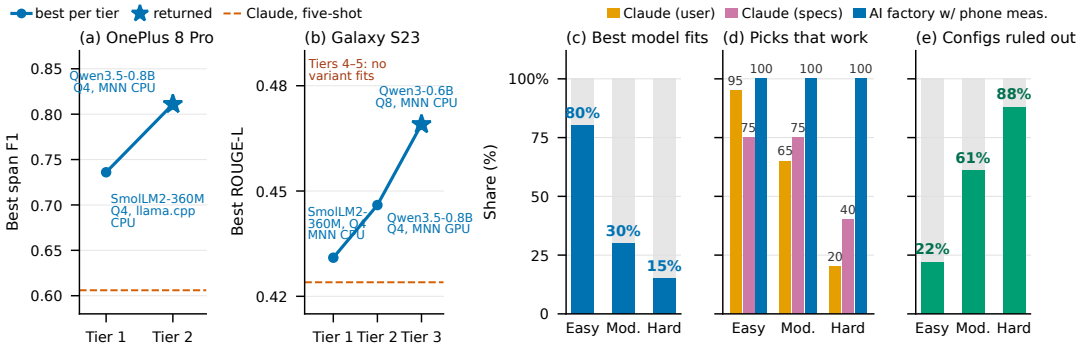


Figure 9: Phone-aware factory deployment. (a, b) Two representative end-to-end runs: biomedical entity extraction on a OnePlus 8 Pro (moderate requirement) and call summaries on a Galaxy S23 (hard requirement); labels give each tier's model, precision, engine and backend. (c–e) Across 60 requirements: (c) how often the factory's most accurate model for the task meets the requirement on the phone; (d) how often a configuration picked without measurements meets the requirement. The measured-selection bar is a reference: it meets every requirement by construction; (e) the share of the phone's configurations that measurements rule out before any training.

Task	CPU	GPU	Task	CPU	GPU
BC5CDR	−.0004	−.0017	CLINC150	−.0004	−.0004
Calendar	.0000	−.0019	SMS Spam	−.0160	−.0119
TOPv2	.0000	+.0030	GoEmo.	−.0062	−.0079
MultiCoN.	+.0005	−.0005	BEA-19	−.0007	+.0015
BFCL	+.0010	+.0010	DialogSum	−.0014	−.0022

Table 4: Accuracy of each factory model on the phone's CPU and GPU, minus its accuracy on the server.

depends on the phone. Of 18 deployments tested on multiple phones, 15 require different configurations on different phones. The same task uses a 4B 4-bit model on the S23 GPU, a 4B 8-bit model on the OnePlus 15 CPU, and a 2B model on the Redmi 9A.

We gave Claude Opus 5.5 with high reasoning effort, a stronger model than the factory's orchestrator, the same 60 requirements under two settings. In one, it received all information available without measurement: phone specs, model size and precision, engines, backends, and the four limits. In the other, it saw only the phone name and a plain-language app description. Given the specifications, Claude Opus 5.5 picked a configuration that misses the requirement in 25% of easy, 25% of moderate and 60% of hard cases; given only an app description, in 5%, 35% and 80% (Fig. 9(d)).

Given the spec sheet, Claude underestimated run-time memory and often misjudged prefill speed, which depends on the backend (§3). It chose the right inference engine in 92% of cases, since MNN was usually best, but the right CPU/GPU backend in only 75%. In 11 of 60 cases, the right model at the right precision failed because Claude chose the wrong backend (8 cases) or engine (3 cases). With only a user-style description, it did better on easy requirements but worse on moderate and hard ones: 15 of its 16 misses on hard requirements chose a larger model or a higher precision than fits, while on easy ones it often underused the phone. In 10 successful cases, the phone could have run a model with 2–5.6× more parameters.

Before training anything, the measurements rule out a median of 22% of a phone's configurations for easy requirements, 61% for moderate ones and 88% for hard ones (Fig. 9e). On hard requirements, the largest model that fits is 0.8B or smaller in 19 of 20 cases, so the factory never needs to train a model of 1.7B parameters or more. The tighter the requirement, the more the measurements save training time.

Finally, the factory scores every model on a server GPU. While it uses the same engine as the phone, the hardware and the kernels are different. So as a final step, we copy a factory-trained checkpoint for each task to a OnePlus 15, and score it there on the same held-out rows, once on the phone's CPU and once on its GPU. On the GPU, every transformer layer and the KV cache run on the GPU; token embedding lookup, tokenization and sampling still run on the CPU. Table 4 shows the difference from the server. On the CPU, eight tasks differ by at most 0.0014, and two of them give the same score. On the GPU, seven differ by at most 0.0022. The largest differences on both are on spam detection, 0.016 on the CPU and 0.012 on the GPU, at our noise floor, and on emotion classification, 0.006 and 0.008. Spam detection moves the most because its score is the F1 of the spam class alone, so a few messages near the decision boundary can change it. The phone's CPU and GPU also agree with each other, to within 0.0041 on every task.

6 Discussion and Limitations

A developer still provides the task specification: data sources, the metric, and a verifier for model outputs. Our results suggest that the verifier matters most. Rejecting malformed outputs made synthetic data usable, while tasks where few generated rows survived quality control saw little benefit from synthesis. Building this harness is the most hand-engineered part of the pipeline. An agent that decomposes a task into its type, such as classification, extraction or generation, and assembles a matching scorer and verifier would make the factory far more autonomous. So would an early check of synthetic-data survival rates.

We also restrict the factory to a predefined set of decoder models from 135M to 4B. This is practical when developers prefer specific model vendors, but the restriction is not fundamental. The factory could instead download candidate base models on demand, and once a task is known to be classification, add encoder models of 1–50M parameters, far smaller than any model in our pool. Training is LoRA fine-tuning only; preference optimization, reinforcement learning or distillation from a larger model are natural additions. Synthesis writes a few hundred rows per round from a local 35B teacher. This keeps each round cheap and targeted, but may cap what synthesis can add; larger rounds from a stronger teacher are worth exploring.

Our phone profiles cover CPUs and GPUs. Recent work shows that mobile NPUs can process prompts an order of magnitude faster while offering little decode advantage over CPUs [8]. Adding NPUs would require only extending the hardware profile, and would make backend selection even more important. We also measure each configuration alone, from a cool start, at one prompt length, with four threads, and with the phone otherwise idle. A deployed model shares the phone with other apps, other models and changing thermal state, so sustained and concurrent use, and longer prompts, may change which configurations meet a requirement. Profiling a phone is slow, largely because it must cool between runs, but this cost is paid once and shared across tasks. Future work could measure only a subset of configurations on a new phone and predict the rest, while validating those predictions before the factory relies on them.

7 Conclusion

We present Mobile AI Factory, an agentic system that builds a fine-tuned, quantized small language model for a given task and phone. We showed that the right model and configuration change from phone to phone, and that even a frontier model cannot make this choice without measurements.

Taking a step back, ML and agent researchers mainly optimize metrics and rarely think about mobile devices. Mobile systems researchers, on the other hand, build inference engines and take the model as given. Mobile AI Factory suggests a near-future in which agents build many phone-specific small, specialized models, and rebuild them as new phones and open base models are released. The same approach could extend beyond phones to wearables and embedded devices.

References

[1] Marah Abdin, Jyoti Aneja, Hany Awadalla, Ahmed Awadallah, Ammar Ahmad Awan, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Jianmin Bao, Harkirat Behl, Alon Benhaim, Misha Bilenko, Johan Bjorck, Sébastien Bubeck, Martin Cai, Qin Cai, Vishrav Chaudhary, Dong Chen, Dongdong Chen, Weizhu Chen, Yen-Chun Chen, Yi-Ling Chen, Hao Cheng, Parul Chopra, Xiyang Dai, Matthew Dixon, Ronen Eldan, Victor Fragoso, Jianfeng Gao, Mei Gao, Min Gao, Amit Garg, Allie Del Giorno, Abhishek Goswami, Suriya Gunasekar, Emman Haider, Junheng Hao, Russell J. Hewett, Wenxiang Hu, Jamie Huynh, Dan Iter, Sam Ade Jacobs, Mojan Javaheripi, Xin Jin, Nikos Karampatziakis, Piero Kauffmann, Mahoud Khademi, Dongwoo Kim, Young Jin Kim, Lev Kurilenko, James R. Lee, Yin Tat Lee, Yuanzhi Li, Yunsheng Li, Chen Liang, Lars Liden, Xihui Lin, Zeqi Lin, Ce Liu, Liyuan Liu, Mengchen Liu, Weishung Liu, Xiaodong Liu, Chong Luo, Piyush Madan, Ali Mahmoudzadeh, David Majercak, Matt Mazzola, Caio César Teodoro Mendes, Arindam Mitra, Hardik Modi, Anh Nguyen, Brandon Norick, Barun Patra, Daniel Perez-Becker, Thomas Portet, Reid Pryzant, Heyang Qin, Marko Radmilac, Liliang Ren, Gustavo de Rosa, Corby Rosset, Sambudha Roy, Olatunji Ruwase, Olli Saarikivi, Amin Saied, Adil Salim, Michael Santacroce, Shital Shah, Ning Shang, Hiteshi Sharma, Yelong Shen, Swadheen Shukla, Xia Song, Masahiro Tanaka, Andrea Tupini, Praneetha Vaddamanu, Chunyu Wang, Guanhua Wang, Lijuan Wang, Shuohang Wang, Xin Wang, Yu Wang, Rachel Ward, Wen Wen, Philipp Witte, Haiping Wu, Xiaoxia Wu, Michael Wyatt, Bin Xiao, Can Xu, Jiahang Xu, Weijian Xu, Jilong Xue, Sonali Yadav, Fan Yang, Jianwei Yang, Yifan Yang, Ziyi Yang, Donghan Yu, Lu Yuan, Chenruidong Zhang, Cyril Zhang, Jianwen Zhang, Li Lyna Zhang, Yi Zhang, Yue Zhang, Yunan Zhang, and Xiren Zhou. 2024. Phi-3 Technical Report: A Highly Capable Language Model Locally on Your Phone. arXiv:2404.14219 [cs.CL] https://arxiv.org/abs/2404.14219

[2] Lakshya A. Agrawal et al. 2026. GEPA: Reflective Prompt Evolution Can Outperform Reinforcement Learning. *arXiv preprint arXiv:2507.19457* (2026).

[3] Dhruv Atreja, Julia White, Nikhil Nayak, Kelton Zhang, Henrijs Princis, George Hurn-Maloney, Ash Lewis, and Urchade Zaratiana. 2026. Pioneer Agent: Continual Improvement of Small Language Models in Production. arXiv:2604.09791 [cs.AI] https://arxiv.org/abs/2604.09791

[4] AWS Samples. 2026. Slemify: Generate, train, and validate small specialist models. https://github.com/aws-samples/sample-slemify. Accessed: 2026-10-20.

[5] Hritik Bansal et al. 2025. Smaller, Weaker, Yet Better: Training LLM Reasoners via Compute-Optimal Sampling. In *International Conference on Learning Representations (ICLR)*.

[6] James Bergstra and Yoshua Bengio. 2012. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research* 13 (2012), 281–305.

[7] Cactus Compute, Inc. [n. d.]. Cactus: AI Inference Engine for Phones & Wearables. https://github.com/cactus-compute/cactus. Accessed: 2026-10-20.

[8] Guanyu Cai, Ruiming Tian, Lang Yang, Zhouhong Ren, Jinliang Yuan, Lingkun Li, and Jiliang Wang. 2026. Is Your NPU Ready for LLMs? Dissecting the Hidden Efficiency Bottlenecks in Mobile LLM Inference.

arXiv:2607.05475 [cs.AR] https://arxiv.org/abs/2607.05475

[9] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. 2020. Once for All: Train One Network and Specialize it for Efficient Deployment. In *International Conference on Learning Representations*.

[10] Han Cai, Ligeng Zhu, and Song Han. 2019. ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware. In *International Conference on Learning Representations*. https://arxiv.org/pdf/1812.00332.pdf

[11] Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Aleksander Madry, and Lilian Weng. 2025. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering. In *The Thirteenth International Conference on Learning Representations*. https://openreview.net/forum?id=6s5uXNWGIh

[12] LLAMA CPP contributors. 2023. llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp. Accessed 2026.

[13] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[14] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *International Conference on Machine Learning (ICML)*.

[15] Biyi Fang, Xiao Zeng, and Mi Zhang. 2018. NestDNN: Resource-Aware Multi-Tenant On-Device Deep Learning for Continuous Mobile Vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 115–127. doi:10.1145/3241539.3241559

[16] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and Robust Automated Machine Learning. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[17] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. In *International Conference on Learning Representations (ICLR)*.

[18] Jiaxiang Geng, Lunyu Zhao, Yiyi Lu, and Bing Luo. 2026. MobileFineTuner: A Mobile-Native Framework for On-Device LLM Fine-Tuning in Real-World Embedded AI Applications. arXiv:2512.08211 [cs.LG] https://arxiv.org/abs/2512.08211

[19] Rui Han, Qinglong Zhang, Chi Harold Liu, Guoren Wang, Jian Tang, and Lydia Y. Chen. 2021. LegoDNN: Block-Grained Scaling of Deep Neural Networks for Mobile Vision. In *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking (MobiCom)*. 406–419. doi:10.1145/3447993.3483249

[20] Yihui He, Ji Lin, Zhijian Liu, Hanrui Wang, Li-Jia Li, and Song Han. 2018. AMC: AutoML for Model Compression and Acceleration on Mobile Devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*. 784–800.

[21] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. 2023. Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes. In *Findings of the Association for Computational Linguistics: ACL 2023*. 8003–8017.

[22] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. LoRA: Low-Rank Adaptation of Large Language Models. In *International Conference on Learning Representations (ICLR)*.

[23] Wei Huang, Anda Cheng, Yinggui Wang, Lei Wang, and Tao Wei. 2026. LLM-AutoDP: Automatic Data Processing via LLM Agents for Model Fine-tuning. *Proc. VLDB Endow.* 19, 5 (May 2026), 794–807. doi:10.14778/3796195.3796196

[24] Xiaotang Jiang, Huan Wang, Yiliu Chen, Ziqi Wu, Lichuan Wang, Bin Zou, Yafeng Yang, Zongyang Cui, Yu Cai, Tianhang Yu, Chengfei Lv, and Zhihua Wu. 2020. MNN: A Universal and Efficient Inference Engine. In *Proceedings of Machine Learning and Systems (MLSys)*.

[25] Zhengyao Jiang, Dominik Schmidt, Dhruv Srikanth, Dixing Xu, Ian Kaplan, Deniss Jacenko, and Yuxiang Wu. 2025. AIDE: AI-Driven Exploration in the Space of Code. arXiv:2502.13138 [cs.AI] https://arxiv.org/abs/2502.13138

[26] Keisuke Kamahori, Shihang Li, Simon Peter, and Baris Kasikci. 2026. VibeServe: Can AI Agents Build Bespoke LLM Serving Systems? arXiv:2605.06068 [cs.AI] https://arxiv.org/abs/2605.06068

[27] Samuel Kessler, Menglin Xia, Daniel Madrigal Diaz, Dongge Han, Helia Heshemi, Saravan Rajmohan, Victor Ruehle, and Jordan T. Ash. 2026. Towards Active Synthetic Data Generation for Finetuning Language Models. arXiv:2512.00884 [cs.LG] https://arxiv.org/abs/2512.00884

[28] Omar Khattab et al. 2023. DSPy: Compiling Declarative Language Model Calls into Self-Improving Pipelines. *arXiv preprint arXiv:2310.03714* (2023).

[29] Stefanos Laskaridis, Kleomenis Katevas, Lorenzo Minto, and Hamed Haddadi. 2024. MELTing Point: Mobile Evaluation of Language Transformers. In *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking* (Washington D.C., DC, USA) *(ACM MobiCom '24)*. Association for Computing Machinery, New York, NY, USA, 890–907. doi:10.1145/3636534.3690668

[30] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. 2017. Hyperband: A Novel Bandit-Based Approach to Hyperparameter Optimization. *Journal of Machine Learning Research* (2017).

[31] Qizheng Li, Yifei Zhang, Xiao Yang, Xu Yang, Zhuo Wang, Weiqing Liu, and Jiang Bian. 2026. FT-Dojo: Towards Autonomous LLM Fine-Tuning with Language Agents. arXiv:2603.01712 [cs.AI] https://arxiv.org/abs/2603.01712

[32] Yiming Li, Yuhan Cheng, Mingchen Ma, Yihang Zou, Ningyuan Yang, Wei Cheng, Hai "Helen" Li, Yiran Chen, and Tingjun Chen. 2026. Skilled AI Agents for Embedded and IoT Systems Development. arXiv:2603.19583 [cs.SE] https://arxiv.org/abs/2603.19583

[33] Ji Lin, Wei-Ming Chen, Yujun Lin, John Cohn, Chuang Gan, and Song Han. 2020. MCUNet: tiny deep learning on IoT devices. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 982, 12 pages.

[34] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. 2024. AWQ: Activation-aware Weight Quantization for On-Device LLM Compression and Acceleration. In *Proceedings of Machine Learning and Systems*, P. Gibbons, G. Pekhimenko, and C. De Sa (Eds.), Vol. 6. 87–100. https://proceedings.mlsys.org/paper_files/paper/2024/file/42a452cbafa9dd64e9ba4aa95cc1ef21-Paper-Conference.pdf

[35] Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. 2024. Large Language Models to Enhance Bayesian Optimization. In *International Conference on Learning Representations (ICLR)*.

[36] Zechun Liu, Changsheng Zhao, Forrest Iandola, Chen Lai, Yuandong Tian, Igor Fedorov, Yunyang Xiong, Ernie Chang, Yangyang Shi, Raghuraman Krishnamoorthi, Liangzhen Lai, and Vikas Chandra. 2024. MobileLLM: optimizing sub-billion parameter language models for on-device use cases. In *Proceedings of the 41st International Conference on Machine Learning* (Vienna, Austria) *(ICML'24)*. JMLR.org, Article 1316, 24 pages.

[37] Yujie Luo, Xiangyuan Ru, Jingsheng Zheng, Jingjing Wang, Yuqi Zhu, Jintian Zhang, Runnan Fang, Kewei Xu, Ye Liu, Zheng Wei, Jiang Bian, Zang Li, and Shumin Deng. 2026. Exploring Autonomous Agentic Data Engineering for Model Specialization. arXiv:2605.30407 [cs.CL]

https://arxiv.org/abs/2605.30407

[38] Nestor Maslej, Loredana Fattorini, Raymond Perrault, Vanessa Parli, Anka Reuel, Erik Brynjolfsson, John Etchemendy, Katrina Ligett, Terah Lyons, James Manyika, Juan Carlos Niebles, Yoav Shoham, Russell Wald, and Jack Clark. 2024. *The AI Index 2024 Annual Report.* Technical Report. AI Index Steering Committee, Institute for Human-Centered AI, Stanford University.

[39] Meta AI. 2024. ExecuTorch: On-device AI across mobile, embedded and edge for PyTorch. https://github.com/pytorch/executorch. Accessed 2026.

[40] MLC team. 2023. MLC-LLM: Universal LLM Deployment Engine with ML Compilation. https://github.com/mlc-ai/mlc-llm. Accessed 2026.

[41] Subhabrata Mukherjee, Arindam Mitra, Ganesh Jawahar, Sahaj Agarwal, Hamid Palangi, and Ahmed Awadallah. 2023. Orca: Progressive Learning from Complex Explanation Traces of GPT-4. arXiv:2306.02707 [cs.CL] https://arxiv.org/abs/2306.02707

[42] Rithesh Murthy, Liangwei Yang, Juntao Tan, Tulika Manoj Awalgaonkar, Yilun Zhou, Shelby Heinecke, Sachin Desai, Jason Wu, Ran Xu, Sarah Tan, Jianguo Zhang, Zhiwei Liu, Shirley Kokane, Zuxin Liu, Ming Zhu, Huan Wang, Caiming Xiong, and Silvio Savarese. 2024. MobileAIBench: Benchmarking LLMs and LMMs for On-Device Use Cases. arXiv:2406.10290 [cs.CL] https://arxiv.org/abs/2406.10290

[43] Jaehyun Nam, Jinsung Yoon, Jiefeng Chen, Jinwoo Shin, Sercan Ö. Arik, and Tomas Pfister. 2025. MLE-STAR: Machine Learning Engineering Agent via Search and Targeted Refinement. *ArXiv* abs/2506.15692 (2025). https://api.semanticscholar.org/CorpusID:279464812

[44] Qualcomm Technologies, Inc. [n. d.]. Qualcomm AI Hub. https://aihub.qualcomm.com. Accessed: 2026-10-20.

[45] Ben Rank, Hardik Bhatnagar, Ameya Prabhu, Shira Eisenberg, Karina Nguyen, Matthias Bethge, and Maksym Andriushchenko. 2026. PostTrainBench: Can LLM Agents Automate LLM Post-Training? arXiv:2603.08640 [cs.SE] https://arxiv.org/abs/2603.08640

[46] Vishwam Raval, Mohamed Zeid, and Prasad Enjeti. 2025. Circuit-AI: A Self-Hosted AI-Agent Language Model Framework for Control Loop Implementation and Simulation. In *2025 IEEE International Communications Energy Conference (INTELEC)*. 91–96. doi:10.1109/INTELEC63987.2025.11214739

[47] Ilia Shumailov, Zakhar Shumaylov, Yiren Zhao, Nicolas Papernot, Ross Anderson, and Yarin Gal. 2024. AI Models Collapse When Trained on Recursively Generated Data. *Nature* 631 (2024), 755–759.

[48] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, and Quoc V. Le. 2019. MnasNet: Platform-Aware Neural Architecture Search for Mobile. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE Computer Society, Los Alamitos, CA, USA, 2815–2823. doi:10.1109/CVPR.2019.00293

[49] Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined Selection and Hyperparameter Optimization of Classification Algorithms. In *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*.

[50] Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. 2025. AutoML-agent: a multi-agent LLM framework for full-pipeline AutoML. In *Proceedings of the 42nd International Conference on Machine Learning* (Vancouver, Canada) *(ICML'25)*. JMLR.org, Article 2385, 48 pages.

[51] Kuan Wang, Zhijian Liu, Yujun Lin, Ji Lin, and Song Han. 2019. HAQ: Hardware-Aware Automated Quantization with Mixed Precision. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 8612–8620.

[52] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. 2023. Self-Instruct: Aligning Language Models with Self-Generated Instructions. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Toronto, Canada, 13484–13508. doi:10.18653/v1/2023.acl-long.754

[53] Zhaode Wang, Jingbang Yang, Xinyu Qian, Shiwen Xing, Xiaotang Jiang, Chengfei Lv, and Shengyu Zhang. 2024. MNN-LLM: A Generic Inference Engine for Fast Large Language Model Deployment on Mobile Devices. In *Proceedings of the 6th ACM International Conference on Multimedia in Asia Workshops* (Auckland, New Zealand) *(MMAsia '24 Workshops)*. Association for Computing Machinery, New York, NY, USA, Article 11, 7 pages. doi:10.1145/3700410.3702126

[54] Hao Wen, Yuanchun Li, Zunshuai Zhang, Shiqi Jiang, Xiaozhou Ye, Ye Ouyang, Ya-Qin Zhang, and Yunxin Liu. 2023. AdaptiveNet: Post-Deployment Neural Architecture Adaptation for Diverse Edge Environments. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking (MobiCom)*. doi:10.1145/3570361.3592529

[55] Bi-Chen Wu, Kurt Keutzer, Xiao-Liang Dai, Pei-Zhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, and Yang-Qing Jia. 2018. FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search. *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (2018), 10726–10734. https://api.semanticscholar.org/CorpusID:54461508

[56] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, et al. 2019. Machine Learning at Facebook: Understanding Inference at the Edge. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*.

[57] Daliang Xu, Hao Zhang, Liming Yang, Ruiqi Liu, Gang Huang, Mengwei Xu, and Xuanzhe Liu. 2025. Fast On-device LLM Inference with NPUs. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 445–462. doi:10.1145/3669940.3707239

[58] Ruiyang Xu, Jialun Cao, Mingyuan Wu, Wenliang Zhong, Yaojie Lu, Ben He, Xianpei Han, Shing-Chi Cheung, and Le Sun. 2025. EmbedAgent: Benchmarking Large Language Models in Embedded System Development. *arXiv preprint arXiv:2506.11003* (2025).

[59] Zhenliang Xue, Yixin Song, Zeyu Mi, Xinrui Zheng, Yubin Xia, and Haibo Chen. 2024. PowerInfer-2: Fast Large Language Model Inference on a Smartphone. arXiv:2406.06282 [cs.LG] https://arxiv.org/abs/2406.06282

[60] Chengrun Yang et al. 2024. Large Language Models as Optimizers. In *International Conference on Learning Representations (ICLR)*.

[61] Huanqi Yang, Mingzhe Li, Mingda Han, Zhenjiang Li, and Weitao Xu. 2026. AutoEmbed: LLM-driven Automated Software Development for Generic Embedded IoT Systems. In *Proceedings of the 2026 ACM/IEEE International Conference on Embedded Artificial Intelligence and Sensing Systems (SenSys '26)*. Association for Computing Machinery, New York, NY, USA, 362–376. doi:10.1145/3774906.3800466

[62] Kangning Zhang, Wenxiang Jiao, Kounianhua Du, Yuan Lu, Weiwen Liu, Weinan Zhang, and Yong Yu. 2025. LoopTool: Closing the Data-Training Loop for Robust LLM Tool Calls. arXiv:2511.09148 [cs.CL] https://arxiv.org/abs/2511.09148

[63] Li Lyna Zhang, Shihao Han, Jianyu Wei, Ningxin Zheng, Ting Cao, Yuqing Yang, and Yunxin Liu. 2021. nn-Meter: Towards Accurate Latency Prediction of Deep-Learning Model Inference on Diverse Edge Devices. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*. 81–93. doi:10.1145/3458864.3467882

[64] Zhihan Zhang, Alexander Le Metzger, Jiuyang Lyu, Chun-Cheng Chang, Jiayi Shao, Yujia Liu, Emmanuel Azuh Mensah, Edward Wang, Kurtis Heimerl, Gregory D. Abowd, Shwetak Patel, Natasha Jaques, and Vikram Iyer. 2026. Embedded Arena: Iterative Optimization via

Hardware Feedback. arXiv:2606.16190 [cs.AR] https://arxiv.org/abs/2606.16190

[65] Justin Zhao, Timothy Wang, Wael Abid, Geoffrey Angus, Arnav Garg, Jeffery Kinnison, Alex Sherstinsky, Piero Molino, Travis Addair, and Devvret Rishi. 2024. LoRA Land: 310 Fine-tuned LLMs that Rival GPT-4, A Technical Report. *arXiv preprint arXiv:2405.00732* (2024).

[66] Chunting Zhou et al. 2023. LIMA: Less Is More for Alignment. In *Advances in Neural Information Processing Systems (NeurIPS)*.

A Evaluation on Unseen Test Set

				Five-shot	
Task	Rows	Base	Tuned	Claude	DeepSeek
BC5CDR	4,000	0.000	**0.828**	0.652	0.560
Calendar	867	0.036	**0.818**	0.621	0.644
TOPv2	5,000	0.002	**0.562**	0.221	0.112
MultiCoNER	5,000	0.292	**0.619**	0.594	0.508
DialogSum	459	0.241	**0.451**	0.387	0.347
GoEmotions	4,000	0.393	**0.525**	**0.515**	0.466
BEA-19	3,000	0.411	**0.534**	0.504	**0.520**
CLINC (0.99)	4,000	0.704	**0.882**	**0.894**	0.833
BFCL	1,234	0.808	0.816	**0.870**	0.848
CLINC	4,000	0.111	0.822	**0.894**	0.833

Table 5: Every model on the unseen test set, which the factory never used for training or for choosing a model. Base is the untrained model, and Tuned is the factory's fine-tuned model. Frontier models get the same five examples as in Table 3. Bold marks the best score and any score within the 0.015 noise floor of it. CLINC is intent classification; (0.99) marks the run with the goal pinned at 0.99. DialogSum's rows have one reference summary instead of three, so all its scores are lower than in Table 3.

We evaluated the models on unseen test sets that the factory never used for training or model selection: the fine-tuned model, its untrained base, and both frontier models given the same five examples as in Table 3 (Table 5). Spam detection's held-out set took the whole test split, leaving only five unused spam messages, so we could not check it on unseen rows.

The fine-tuned model beats its untrained base on every task, by 0.13 to 0.83, except function calling, where the untrained 4B model is already strong and fine-tuning adds 0.008. Most fine-tuned scores are lower than on the held-out set. On nested parsing, entity extraction and function calling, they drop by 0.035–0.037 while the untrained model barely moves. On intent classification and grammar correction, the untrained model drops just as much, so the unseen test set data are simply harder.

Against frontier models given five examples, the factory's model wins five tasks on the unseen test set, by 0.024 to 0.342: biomedical extraction, calendar extraction, nested parsing, entity extraction and dialogue summarization. Three tasks

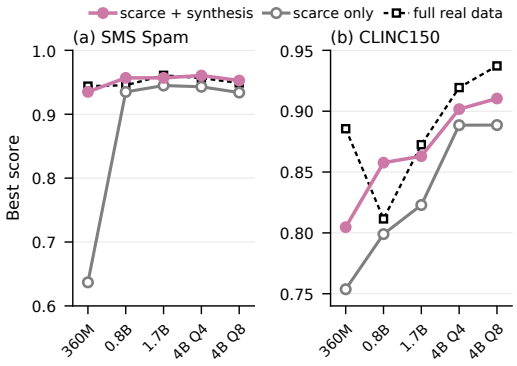


Figure 10: Best score per tier, starting from (a) 100 and (b) 151 real rows, with and without synthesis, against full real data. The x-axis gives each tier's model.

are ties, within the noise floor: emotion classification, grammar correction, and intent classification with the goal at 0.99. Two are losses: function calling, and intent classification with the default goal, which stops on a 360M model. Compared with the held-out set, grammar correction moves from a win to a tie, and intent classification at goal 0.99 moves from a tie in our favor to one against.

B Synthetic Data under Data Scarcity

With thousands of real rows, synthetic rows made up only 3–22% of each training set. So, we starved two tasks of real data: 100 rows for spam detection, and 151 for intent classification, one per intent. Each task ran twice from the same rows, with mining off and the goal set to 0.99, with and without synthesis data.

Synthesis won all ten tier comparisons (Fig. 10), eight of them by more than our 0.015 noise level and four of them by more than 0.03. Tiers within a run share a growing training set, so they are not independent. How much synthesis helped depended on what limited the small model. On spam detection, the 360M model scores 0.637 from 100 real rows and 0.935 with 362 generated rows added, close to its 0.944 on 4,045 mostly real rows. Without synthesis, only the 0.8B to 4B models reached that level. For the intent task, an example for each of 151 intents is too few for any model. Synthesis added 0.013–0.059 at every tier and ended at 0.910, within 0.027 of the full-data run. Without it, no model passed 0.889.

We reran every task with only 100 real rows (151 for intent classification, one per intent), mining off and synthesis enabled. Nested parsing kept its full-data score, but its loader always adds a fixed 674-row in-domain adaptation set, so we leave it out of the counts that follow. Four of the other nine tasks retained at least 97% of their full-data score. Calendar extraction kept 98%, but only on the 4B model; the 0.8B model fell 0.18 behind, suggesting real data matters more for smaller models. Grammar correction and summarization lost

7–10%, while entity and emotion tasks lost 16–30%. Teacher accuracy did not predict this loss. The two largest drops came on the two tasks where the fewest synthetic rows passed quality control: 3% for biomedical extraction and 24% for fine-grained entities, against more than 60% on every other task. Emotion classification points to a second limit: 99% of its rows passed, but with 28 labels and 100 rows it starts with under four examples per label, and it lost 16%. This pass rate is cheap to estimate before relying on synthesis.

C Hardware requirements

Tables 7–9 list the 60 requirements used in §5.3. Each deploys one of our ten tasks in a way an app would use it, on one phone. A requirement sets four limits: peak memory, a minimum prefill speed, a minimum decode speed, and a maximum energy per request. A configuration meets the requirement only if it meets all four.

The kind of deployment sets the memory budget and the response time (Table 6). A fixed memory budget is capped at 25% of the phone's RAM, which matters only on the 4 GB Redmi 9A. The task's real request then sets the speeds: prefill must process the request's input within the first-token time, and decode must finish the answer within the remaining time. The fixed part of each prompt, such as instructions, label lists and tool schemas, is computed once and cached, so a request prefills only its input. A summary the user reads as it streams instead needs at least 5 tokens/s. The energy budget is a share of the phone's battery divided by the deployment's requests per day. The same deployment therefore has different limits on different phones.

The tier of a requirement is the share of the phone's configurations that meet it: easy if at least 60% do, moderate if 30–60% do, and hard if fewer than 30% do. From the 38 deployments run on all four phones, we chose 20 requirements per tier, spread across tasks, phones and deployments, and kept only requirements that at least one configuration meets.

Deployment kind	Memory	First token	Answer
Keyboard	300 MB	0.3 s	1 s
Voice assistant	1,536 MB	0.5 s	2 s
Feature in an app	1,024 MB	0.5 s	3 s
Background service	512 MB	1 s	5 s
Dedicated app	40% of RAM	1 s	5 s
Overnight job	50% of RAM	6 h ÷ requests	

Table 6: Memory budget and response time by kind of deployment. An overnight job must finish all of a night's requests in 6 hours, at most 60 s each.

ID	Task	Phone	Deployment	Memory (MB)	Prefill	Decode	Energy (J)
E01	GoEmotions	S23	Keyboard mood insights	≤ 3,500	≥ 1.9	≥ 1	≤ 135
E02	SMS Spam	OP15	Scam-check app	≤ 6,000	≥ 42	≥ 1	≤ 101
E03	BEA-19	OP8P	English journal corrections	≤ 5,800	≥ 0.8	≥ 1	≤ 261
E04	DialogSum	S23	Overnight call summaries	≤ 3,500	≥ 5.83	≥ 2	≤ 1,350
E05	Calendar JSON	OP15	Planner app	≤ 6,000	≥ 40	≥ 22	≤ 101
E06	BFCL	OP15	Shortcuts app	≤ 6,000	≥ 35	≥ 9	≤ 50.6
E07	BC5CDR	OP15	Paper reader app	≤ 6,000	≥ 32	≥ 12	≤ 10.1
E08	MultiCoNER	OP15	Photo and notes library index	≤ 7,600	≥ 4.07	≥ 7	≤ 12.6
E09	TOPv2	OP15	Reminders and weather app	≤ 6,000	≥ 17	≥ 10	≤ 50.6
E10	GoEmotions	OP8P	Keyboard mood insights	≤ 5,800	≥ 1.9	≥ 1	≤ 157
E11	BEA-19	S23	English journal corrections	≤ 3,500	≥ 0.8	≥ 1	≤ 225
E12	DialogSum	OP8P	Overnight call summaries	≤ 5,800	≥ 5.83	≥ 2	≤ 1,566
E13	SMS Spam	OP15	First-run inbox scan	≤ 7,600	≥ 19.4	≥ 1	≤ 5.06
E14	BC5CDR	OP15	Paper-alert abstract index	≤ 7,600	≥ 2.96	≥ 5	≤ 25.3
E15	GoEmotions	OP15	Journaling app mood tags	≤ 6,000	≥ 57	≥ 2	≤ 25.3
E16	BEA-19	OP15	English-learning feedback	≤ 6,000	≥ 24	≥ 5	≤ 20.2
E17	DialogSum	OP15	Meeting-notes app	≤ 6,000	≥ 175	≥ 5	≤ 101
E18	BEA-19	OP15	Essay proofreading app	≤ 6,000	≥ 24	≥ 5	≤ 33.7
E19	GoEmotions	OP15	Keyboard mood insights	≤ 7,600	≥ 1.9	≥ 1	≤ 253
E20	DialogSum	OP15	Overnight call summaries	≤ 7,600	≥ 5.83	≥ 2	≤ 2,529

Table 7: Easy requirements: at least 60% of the phone's configurations meet all four limits. Prefill and decode are minimum speeds in tokens/s. OP15 = OnePlus 15, S23 = Galaxy S23, OP8P = OnePlus 8 Pro, R9A = Redmi 9A.

ID	Task	Phone	Deployment	Memory (MB)	Prefill	Decode	Energy (J)
M01	CLINC150	OP15	Offline assistant app	≤ 6,000	≥ 515	≥ 1	≤ 33.7
M02	GoEmotions	OP8P	Comment tone check	≤ 1,024	≥ 114	≥ 2	≤ 31.3
M03	SMS Spam	S23	Scam-check app	≤ 2,800	≥ 42	≥ 1	≤ 54
M04	BEA-19	R9A	English journal corrections	≤ 1,900	≥ 0.8	≥ 1	≤ 290
M05	BFCL	OP15	Voice-assistant actions	≤ 1,536	≥ 70	≥ 24	≤ 50.6
M06	Calendar JSON	S23	Planner app	≤ 2,800	≥ 40	≥ 22	≤ 54
M07	BC5CDR	OP8P	Paper-alert abstract index	≤ 5,800	≥ 2.96	≥ 5	≤ 15.7
M08	DialogSum	R9A	Overnight call summaries	≤ 1,900	≥ 5.83	≥ 2	≤ 1,737
M09	MultiCoNER	OP15	Search query understanding	≤ 1,024	≥ 44	≥ 14	≤ 10.1
M10	TOPv2	S23	Reminders and weather app	≤ 2,800	≥ 17	≥ 10	≤ 27
M11	BEA-19	OP8P	Email "fix grammar" button	≤ 1,024	≥ 48	≥ 8	≤ 31.3
M12	GoEmotions	R9A	Keyboard mood insights	≤ 1,900	≥ 1.9	≥ 1	≤ 174
M13	BFCL	OP15	Smart-home app commands	≤ 1,024	≥ 70	≥ 14	≤ 25.3
M14	SMS Spam	S23	First-run inbox scan	≤ 3,500	≥ 19.4	≥ 1	≤ 2.7
M15	MultiCoNER	OP8P	Photo and notes library index	≤ 5,800	≥ 4.07	≥ 7	≤ 7.83
M16	BC5CDR	OP15	Reference-manager highlight tags	≤ 1,024	≥ 64	≥ 18	≤ 25.3
M17	DialogSum	S23	Meeting-notes app	≤ 2,800	≥ 175	≥ 5	≤ 54
M18	TOPv2	OP8P	Reminders and weather app	≤ 4,600	≥ 17	≥ 10	≤ 31.3
M19	MultiCoNER	OP15	Read-later entity tags	≤ 6,000	≥ 22	≥ 9	≤ 5.06
M20	BFCL	S23	Shortcuts app	≤ 2,800	≥ 35	≥ 9	≤ 27

Table 8: Moderate requirements: 30–60% of the phone's configurations meet all four limits. Prefill and decode are minimum speeds in tokens/s.

ID	Task	Phone	Deployment	Memory (MB)	Prefill	Decode	Energy (J)
H01	CLINC150	S23	Voice-assistant intent router	≤ 1,536	≥ 1,030	≥ 3	≤ 27
H02	SMS Spam	OP15	Messaging app spam check	≤ 512	≥ 42	≥ 1	≤ 10.1
H03	Calendar JSON	OP8P	Voice “add to my calendar”	≤ 1,536	≥ 80	≥ 57	≤ 104
H04	BFCL	R9A	Smartwatch actions relay	≤ 512	≥ 35	≥ 9	≤ 17.4
H05	GoEmotions	S23	Keyboard tone check	≤ 300	≥ 190	≥ 8	≤ 5.4
H06	BEA-19	OP15	English-learner keyboard	≤ 300	≥ 80	≥ 29	≤ 5.06
H07	MultiCoNER	OP8P	Long-press smart actions	≤ 512	≥ 22	≥ 9	≤ 15.7
H08	TOPv2	R9A	Reminders app quick add	≤ 950	≥ 34	≥ 16	≤ 34.7
H09	DialogSum	S23	Dialer call summary	≤ 1,024	≥ 350	≥ 5	≤ 90
H10	BC5CDR	OP8P	Paper reader app	≤ 4,600	≥ 32	≥ 12	≤ 6.26
H11	CLINC150	OP15	Banking app assistant	≤ 1,024	≥ 1,030	≥ 2	≤ 506
H12	TOPv2	R9A	Smartwatch command relay	≤ 512	≥ 17	≥ 10	≤ 23.2
H13	Calendar JSON	S23	Calendar quick add	≤ 1,024	≥ 80	≥ 34	≤ 54
H14	DialogSum	OP15	Chat catch-up	≤ 1,024	≥ 350	≥ 5	≤ 50.6
H15	SMS Spam	OP8P	Messaging app spam check	≤ 512	≥ 42	≥ 1	≤ 6.26
H16	BEA-19	R9A	English-learning feedback	≤ 1,500	≥ 24	≥ 5	≤ 13.9
H17	BFCL	S23	Voice-assistant actions	≤ 1,536	≥ 70	≥ 24	≤ 27
H18	GoEmotions	OP15	Keyboard tone check	≤ 300	≥ 190	≥ 8	≤ 10.1
H19	BC5CDR	OP8P	Reference-manager highlight tags	≤ 1,024	≥ 64	≥ 18	≤ 15.7
H20	MultiCoNER	R9A	Long-press smart actions	≤ 512	≥ 22	≥ 9	≤ 17.4

Table 9: Hard requirements: under 30% of the phone’s configurations meet all four limits. Prefill and decode are minimum speeds in tokens/s.