60-second takeaway

Liquid AI did not turn LFM2.5-2.6B into an agent with one broad fine-tuning run. It first trained a general instruction model, branched that checkpoint into domain specialists, distilled their capabilities back into one student on the student's own outputs, then trained the consolidated model inside real agent harnesses.

The result is a 2.69B-parameter, 128K-context model aimed at local tool use and multi-step workflows. Liquid reports inference under 2.5 GB and 220 tokens per second on an Apple M5 Max, but those are vendor measurements, not results we have independently reproduced.

The most interesting part is the training architecture: optimize skills separately, combine them with dense on-policy supervision, then test the combined policy where agents actually operate.

Liquid AI released LFM2.5-2.6B on 4 August 2026 as a compact text model for on-device agents. It has 2.69 billion parameters, a 131,072-token context window and open model weights under Liquid AI's LFM license.

The headline is local deployment. The deeper story is how Liquid tried to fit several competing capabilities into one small model without asking one training stage to solve everything at once.

The official pipeline has four post-training stages:

1. two rounds of supervised fine-tuning;
2. separate domain-specialist teachers;
3. multi-domain on-policy distillation; and
4. reinforcement learning inside agent harnesses.

A public Qwen chat summarized the design neatly as: train several small experts, distill them into one student on the student's own rollouts, then polish the result with agentic reinforcement learning.

That is a useful mental model. Liquid AI's release article supplies the details.

Start with the base model

Before post-training, LFM2.5-2.6B-Base was pretrained on about 34 trillion tokens. Liquid also extended the existing tokenizer to a 128,000-token

vocabulary to improve support for non-Latin scripts, then added a dedicated mid-training phase to extend context to 128K.

The architecture contains 30 layers: 22 double-gated short-convolution blocks and eight grouped-query-attention blocks. This hybrid design is central to Liquid's efficiency pitch. Short convolutions handle much of the sequence processing, while attention layers retain the ability to connect information over longer distances.

Pretraining produces a capable next-token predictor, not yet a reliable assistant or agent. The four-stage post-training process is what shapes that base model into the released checkpoint.

Stage 1: two rounds of supervised fine-tuning

Supervised fine-tuning, or SFT, teaches the model from examples of desired inputs and outputs.

Liquid used two consecutive SFT stages. The first covered a broad range of domains. The second placed more weight on priority capabilities such as reasoning, tool use and agentic tasks. Across both stages, Liquid says the training mix was about seven times larger than the SFT mix used for LFM2.5-8B-A1B.

The mix included:

- tool-use examples;
- web-search tasks;
- software-engineering work;
- agent trajectories; and
- broader instruction and reasoning data.

The resulting checkpoint had two jobs. It became the student that would eventually ship, and it became the common starting point for all the specialist teachers.

That shared origin matters later. A teacher that behaves too differently from its student can provide technically rich but hard-to-follow supervision. Here, every specialist starts from the same SFT checkpoint, so its output distribution should remain closer to the student's.

Stage 2: train one specialist per domain

Liquid copied the SFT checkpoint into separate training branches. Each branch received a focused SFT pass on a reweighted domain mix, followed by reinforcement learning with verifiable rewards, or RLVR.

The disclosed teacher domains were:

- instruction following;
- mathematics;
- knowledge and hallucination control;
- code;
- tool use; and
- long context.

RLVR uses outcomes that can be checked, such as whether an answer is mathematically correct, code passes a test, or a structured response satisfies a contract. It gives each teacher room to specialize without updates from unrelated domains pulling the same weights in another direction.

This is the training recipe's divide-and-conquer step. A math teacher can concentrate on mathematical rewards. A tool-use teacher can learn when and how to call functions. A long-context teacher can optimize behavior over large inputs.

The teachers are intermediate training assets. They do not need to become separate products, and the final agent does not route live requests among six models. Their purpose is to create stronger, domain-specific supervision for one general student.

Stage 3: distill the teachers on the student's own rollouts

Multi-domain on-policy distillation, abbreviated MOPD by Liquid, brings the specialist capabilities back together.

Traditional off-policy distillation often trains a student on answers generated by a teacher. Those answers show what the teacher does well, but they may not cover the mistakes and awkward states the student reaches when generating independently.

On-policy distillation changes the source of the trajectory:

1. sample a prompt;
2. let the student generate its own response;
3. route the prompt to the matching domain teacher; and
4. use the teacher's token-level feedback to update the student.

The teacher therefore supervises behavior drawn from the student's current policy. In plain language, the student receives correction on the path it actually took, not only a polished answer produced by somebody else.

Liquid says this dense, routed supervision helped the student converge quickly while integrating the specialists. The plausible benefit is reduced exposure mismatch: training spends more time on states the student is likely to encounter at inference.

MOPD does not make capability conflicts disappear by definition. Different teacher signals can still pull behavior in competing directions, and aggregate benchmark gains do not prove every domain survived intact. The important architectural choice is that specialization and integration are separated into distinct phases with a common starting checkpoint.

Stage 4: train inside real agent harnesses

After distillation, Liquid applied multi-turn agentic reinforcement learning. This stage moved beyond single answers and placed the model inside environments where it had to research, write, code, analyze data, manage documents and use tools across several steps.

For each rollout, the training system sampled a task and selected a corresponding harness. Liquid names Hermes Agent and OpenClaw among the harnesses used. Each attempt ran in its own sandbox with access to the harness's actual system prompts, tools and interaction conventions.

The optimization method was group relative policy optimization, or GRPO. Liquid describes an outcome reward with three parts:

- an LLM-as-a-judge rubric;
- programmatic checks; and
- a hard safety gate.

This is more grounded than training only on static tool-call examples. The model must choose actions, consume observations and continue until the environment produces an outcome.

It also introduces an evidence limit. LLM judges and programmatic checks measure the properties encoded in their rubrics and tests. They do not automatically establish that the agent is reliable across every harness, tool schema or real-world workflow.

How Liquid connected reinforcement learning to black-box harnesses

The infrastructure behind the final stage is unusually relevant for teams building agent systems.

Liquid separated four jobs:

Component	Job
Training engine	Optimize the model with FSDP
Rollout engine	Generate actions from the latest policy with SGLang
RL framework	Use verl to launch rollouts, collect rewards and update weights
Sandbox service	Run the harness and task environment safely

A harness proxy treated each agent harness as a black box. It captured token-level trajectories without requiring Liquid to rewrite the harness itself. Liquid says it then reconstructed and validated samples with linear-trajectory consistency checks, token-mismatch checks and Rollout Routing Replay, or R3.

This solves a practical training problem. An agent harness may transform prompts, expose its own tools and interleave environment messages. The trainer needs the exact token trajectory associated with each action and observation, even when it does not own the harness internals.

The broader lesson is useful beyond LFM2.5: if a model will run through a harness in production, the harness is part of the behavior-shaping environment, not merely a wrapper added after training.

What the released model is designed to do

Liquid recommends LFM2.5-2.6B for:

- tool use and function calling;
- data extraction;
- retrieval-augmented generation;
- long-context workflows; and
- local, high-volume agent tasks.

The model card explicitly does not recommend it for agentic coding or knowledge-heavy tasks. That qualification matters. Small local models can be valuable executors without replacing larger models for every planning, coding or factual-recall workload.

LFM2.5-2.6B is also a reasoning model whose chat template begins assistant output with a <think> tag. It supports Python-like function calls between dedicated tool-call tokens, with JSON available when requested in the system prompt.

Official checkpoints include the native Transformers format, GGUF for llama.cpp, MLX for Apple Silicon and ONNX for cross-platform runtimes.

How strong are the published results?

Liquid reports that LFM2.5-2.6B competes with models several times larger on instruction following, tool use and agent benchmarks. In its published table, the 2.6B model beats the listed Gemma and Qwen comparison models on Multi-IF and IFStruct, and records the highest ToolSandbox score. Larger Qwen models retain advantages on several math, coding and other agent evaluations.

Liquid also reports:

- 220 output tokens per second on an Apple M5 Max;
- 113 tokens per second on an AMD Ryzen AI Max+ 395;
- about 30 tokens per second on a phone;
- under 2.5 GB of memory for the CPU measurements; and
- almost 15,000 output tokens per second at high concurrency on one H100 SXM5.

These numbers are useful for deciding whether to run a local benchmark. They are not a universal speed ranking. Hardware, quantization, prompt length, output length, concurrency, runtime and sampling settings all affect throughput.

The H100 result used SGLang 0.5.16, BF16, 1,024 input tokens, up to 256 output tokens and three runs per concurrency level. The benchmark table also used different generation settings for some tasks and models. That is enough detail to interpret the vendor result, but not enough to assume the same numbers on another deployment.

What Qwen's explanation got right

The public Qwen answer captured the pipeline's central logic accurately:

- start from a general SFT checkpoint;
- branch it into skill specialists;
- let the student generate the trajectories used for distillation;
- combine specialist knowledge into one deployable model; and
- finish with training for interactive tool use.

Its strongest contribution was compression. It turned a dense diagram into a memorable explanation: optimize skills separately, teach one student on its own behavior, then train the combined model in agent settings.

The official source adds boundaries that should remain attached to that summary. There were two SFT stages, the specialists used RLVR, prompts were routed by domain during MOPD, and the final agent stage used GRPO plus multiple reward mechanisms inside sandboxed harnesses.

A useful analogy: specialists training a general manager

One way to understand the whole process is to imagine how a company might train a new general manager.

The manager begins with a broad education. They know something about many subjects, but they are not yet ready to run every part of a company. This is like the shared SFT checkpoint. It can follow instructions and handle a range of tasks, but its skills are still uneven.

The company then develops specialists from the same starting point. One becomes strong in finance, another in operations, another in software and another in research. Each person can spend more time on the rewards and

mistakes that matter in one field. This resembles Liquid's domain teachers for mathematics, code, tool use, long context and other skills.

The next step is not to put every specialist in the manager's office forever. It is to transfer what they know into the manager.

This is where multi-domain on-policy distillation matters. The manager first attempts a real piece of work. The company then sends that attempt to the specialist who understands the subject. The specialist corrects the path the manager actually took. In the model, the student generates a response, a router selects the matching teacher, and that teacher supplies detailed token-level guidance.

That is closer to an apprenticeship than a textbook. A textbook contains correct answers to problems chosen in advance. A mentor sees the learner's own work and can focus on the mistakes that learner really makes. On-policy distillation uses the same basic idea: teach on the student's current behavior instead of relying only on polished answers produced by the teacher.

After learning from the specialists, the manager still needs experience running the company. Decisions unfold over time. An early choice changes the information, tools and options available later. Results also need to be measured. This is the role of agentic reinforcement learning. Liquid placed the combined model inside agent harnesses, gave it multi-step tasks and scored the outcomes with rubrics, programmatic checks and a safety gate.

With enough useful correction and practice, some decisions become fast. A seasoned manager may appear to act on intuition because they no longer need to work through every old lesson in public. In a model, something similar can happen in a narrow computational sense. Expensive training, feedback and search are compressed into the model's weights, allowing it to produce a response quickly at runtime. This is sometimes described as amortized reasoning: do more work during learning so that later decisions cost less.

The analogy has limits. The specialists do not all debate every prompt. Liquid routes each prompt to the matching teacher during training. The released model also does not call those teachers when it answers a user. Their guidance has been distilled into one set of weights.

A model is not a human manager either. A deployed model with frozen weights does not keep gaining experience from each task unless a separate system records that experience and trains or updates it later. Fast model behavior does

not prove consciousness, human understanding or real-world judgment. Quantitative rewards can also teach the wrong lesson when they measure a poor proxy for the desired result.

The deeper pattern still holds: develop skills separately, combine them by correcting the generalist's own work, then test the combined policy in an environment where actions have consequences. That sequence is the clearest reason the four training stages belong together.

The practical takeaway for local-agent builders

LFM2.5-2.6B is interesting because its training recipe matches a real deployment tension. A local agent must be small and fast, but it also needs instruction following, tool syntax, planning, long-context behavior and enough domain competence to finish useful work.

Liquid's answer was not to rely on one undifferentiated training mix. It built competence in branches, recombined those branches on student-generated behavior, then optimized the combined policy in the harness layer where mistakes become visible.

That does not establish that LFM2.5-2.6B will outperform a larger model on your workload. It makes the model a credible candidate for a bounded local trial.

The next useful test is straightforward: run the exact tools, prompts, context lengths and failure conditions your agent will face. Measure task completion, incorrect tool calls, loops, latency, memory and recovery behavior. A small model that completes routine work reliably can be more valuable than a larger model whose extra capability is consumed by cost or latency.

References

- Liquid AI: LFM2.5-2.6B - Deploy Agents Everywhere
- LiquidAI/LFM2.5-2.6B model card
- LiquidAI/LFM2.5-2.6B-Base model card
- Public Qwen explanation of the training process