Turn your coding agents into a continual-learning lab

Infrastructure for thousands of task-specialised models that learn continuously from every interaction.

---

1 Enabling a continual learning system

We believe the future of AI is not a handful of giant generalist models, but **thousands of task-specialised models that learn continuously from every interaction**. In that future, **enterprises will build a continual-learning system for every use-case** they have - each one a model that keeps getting better at its specific job. Realising that future requires a new layer of infrastructure: the **autoresearch** infrastructure for continuously learning models - the abstractions that let a system discover what to learn, train on it, and ship it autonomously and safely. **The SDK builds that layer.** The SDK abstracts away all the complexities related to training models and makes coding agents and you focus only on two parts - **data** and the **algorithm**

To make that autoresearch work, you need to both **standardise** and **centralise** the entire history of experiments, so that coding agents have the context they need. That is what this SDK does: it lets coding agents - and you - focus on the training experiments themselves (finding the right **data** and **recipe**), while the SDK handles the standardisation, the centralisation, and the supply of the right context to you and your coding agents. Without this standardisation a coding agent just sees a pile of ad-hoc training scripts: every experiment looks different, so it cannot reason across them or build on what worked. Standardisation is what makes the history of experiments legible enough for an agent to learn from.

The end state is a closed loop. A deployed model produces **production traces**; a lightweight **agent decides** whether a given trace is worth learning from and, if so, triggers **autoresearch**; the autoresearch loop runs a series of educated experiments and trains a candidate checkpoint; **deployment evals** then judge whether that checkpoint is genuinely better than what is in production - and only then is it **deployed**. The freshly deployed model produces new traces, and the loop continues, so the system improves with every interaction. What we release today is **Phase 1**: the autoresearch loop - everything the rest of this document describes.

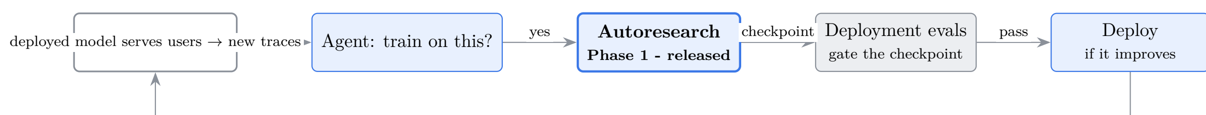

Figure 1: The continual-learning system. A deployed model produces traces; an agent decides whether to learn from them and triggers **autoresearch** (this SDK), which trains a candidate checkpoint; deployment evals gate whether it ships. The blue **autoresearch** stage - and only that stage - is what we release as Phase 1.

1.1 Why two agents, and why autoresearch?

The design rests on a deliberate split of work across **two agents**.

The **first agent is a light, fast decision-maker**. It watches incoming production traces and decides whether a batch is worth learning from. Because it gates the expensive step, it must be cheap and quick - and it needs *no* context of previous experiments. If it says yes, it hands off to the second stage, where the real budget is spent.

That second stage is autoresearch - a heavier, budgeted search for the right training recipe. We frame continual learning as autoresearch on purpose: there is **no single algorithm** that enables continual learning across all data; different data demands different recipes. So a capable agent must autonomously run experiments, recover from training collapses, and decide which checkpoint is finally worth deploying. That is exactly what the autoresearch loop does - which is why training a model with this SDK is, by construction, a building block of the larger continual-learning system.

2 Why do we need it?

We believe in a world where there will be thousands of custom models, each specialised for a particular task. Training a model that sits on the Pareto frontier of the accuracy-versus-cost curve is a cumbersome task: it comes down to deciding what data goes into training and which recipe is used. We believe auto-research is the answer to this problem: coding agents run a series of educated experiments and finally train a model that meets the targeted requirements.

Tinker-protocol-compatible training infrastructure is increasingly making it easy to train models, abstracting away the model infrastructure and the complexities of training large models. However, engineers and researchers still have to focus on a lot of plumbing code and experiment management. With this SDK, launching an experiment is now as simple as writing a YAML file.

3 Design principles

- **Separate data from recipe.** We believe that to train a frontier model you need to choose two main things: the **data** and the **training recipe**. All of our abstractions are built around this, with a clear separation between the two. It lets you edit and run experiments across different training data and algorithms, while abstracting away all the plumbing around each experiment.
- **Any algorithm or data ablation.** The abstractions are general enough to express any training algorithm and any data ablation, and every run is scored against a benchmark, so each variation you try is linked directly to downstream task performance.
- **Capture context for auto-research.** To enable auto-research, you need to provide the right context to your coding agents. Our main class is the `Experiment` class: for every experiment you can store a hypothesis (the reason for running it) and a conclusion (its result).
- **Single line experiment configs enables auditability** Launch experiments with a single-line, the SDK handles all the management.
- **Built for multi-turn, agentic workloads.** Workloads are moving from single-turn chatbots to more agentic, multi-turn workflows. We therefore support multi-turn reinforcement learning inside custom agent harnesses, so you can run ablations on both the algorithm and the harness together.

- **Standardise without losing flexibility.** When a coding agent gathers context across hundreds of experiments, standardisation helps a lot with understanding each one. For this reason we have chosen highly customisable yet general data models, giving you flexibility while keeping the experiments standardised.

4 The three pillars at a glance

A single `ExperimentConfig` (one YAML file) drives one run. Conceptually the run is a left-to-right pipeline: the **Data** surface produces typed rows, the **Training** surface consumes them and emits checkpoints, and the **Evaluation** surface scores those checkpoints, with validation looping back to steer model selection.

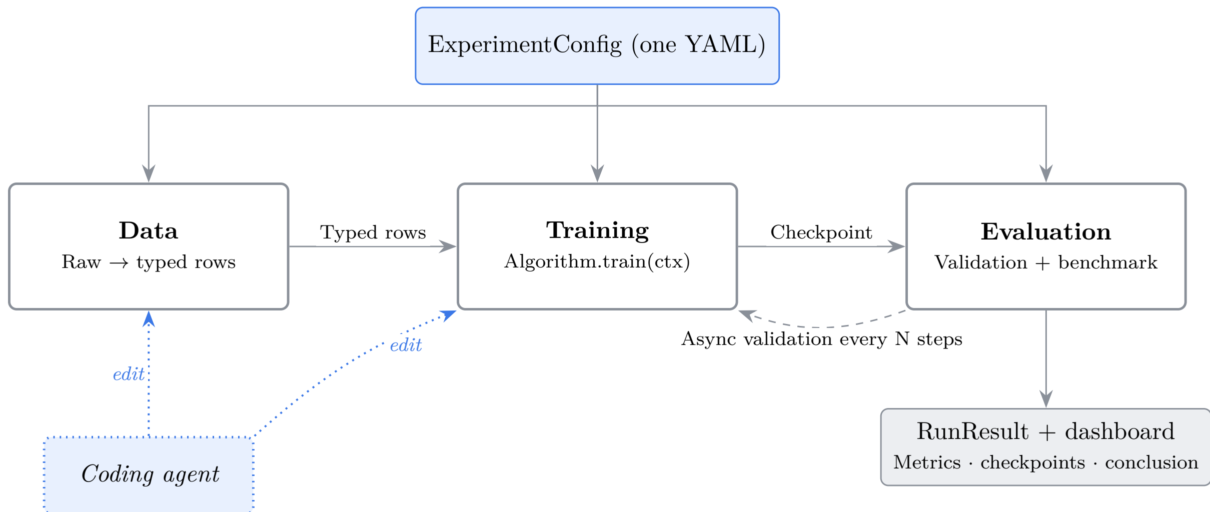

Figure 2: Top-level dataflow. The user authors one config (blue); the SDK (grey) runs Data → Training → Evaluation and records a RunResult. Evaluation's in-loop validation feeds back into training to drive model selection, while the benchmark is scored once at the end. The dotted arrows mark the two extension points a **coding agent** can author: a `transforms()` on the data surface, and a `StepBuilder` on the training surface.

5 The Experiment

Everything in the SDK is organised around the **Experiment**, the unit of scientific work. An experiment is two things together: an **experiment config** (a single YAML file that declares the data, the recipe, and how to evaluate) and a short **SDK script** that launches it. That is all it takes to run.

Because an experiment is a scientific act, it carries its reasoning with it. Every experiment records a **hypothesis** (the reason you are running it) and, once finished, a **conclusion** (what you learned). This is what keeps the history legible: a coding agent (or a teammate) reading back over hundreds of past experiments sees not just numbers, but the question each one asked and the answer it reached.

One experiment need not be a single run. From the same config you can fan out into many **runs** (a sweep over learning rates, datasets, or algorithms) and repeat each across several seeds to measure variance. The SDK runs them all, scores each against your chosen **success metric**, and selects the best to form the experiment's conclusion. You describe the variations declaratively; the SDK handles the orchestration, the dashboard, and the comparison.

Each run is where the two choices come together: a choice of **data** and a choice of **recipe** (the training algorithm and how it is evaluated). The rest of this document walks through those two surfaces in turn: first how the data is shaped, then how training and evaluation run.

6 Data surface

Standardize *anything* into a few typed shapes, then hand those to the algorithm. Loading, caching, and conversion are the SDK's job; the user only declares a source and (optionally) custom transforms.

The raw data is a `jsonl` file. A **transform** is the piece a coding agent writes to convert that raw data into one of the typed formats (`ChatMessagesRow` / `PromptExample` / `HarborTask`), which is then consumed by the training loop ahead. This separation is what lets you run **data experiments**: the same raw data, but different *actual* training data depending on the transform (for example, thinking vs. non-thinking traces, with or without system prompts, or different templating) without ever touching the source.

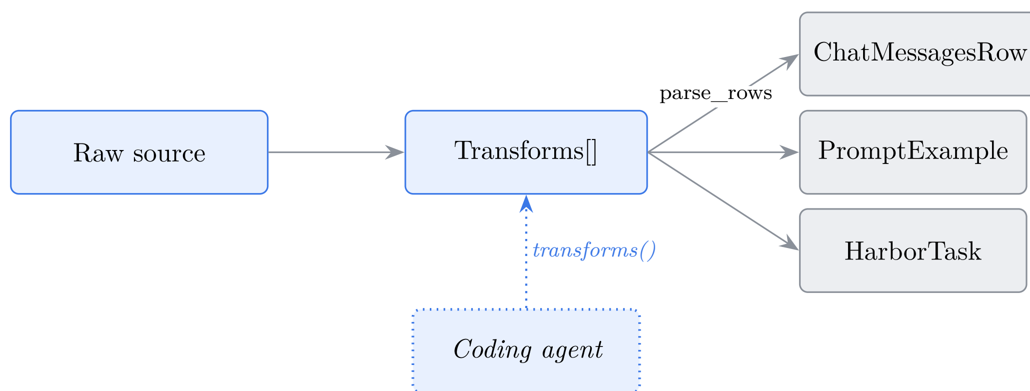

Figure 3: Data pipeline. A declared source is passed through ordered `Transforms` and converted (`parse_rows`) into one of three typed shapes. The shape chosen depends on the algorithm: chat SFT consumes `ChatMessagesRow`, SDFT consumes `PromptExample`, RL consumes `HarborTask`.

7 Training & Evaluation

You focus on the **algorithm**; the SDK does the plumbing. To define a new training method you implement a single function, `build_batch`. This function is the algorithm: it does the math for one optimisation step and returns both the batch of data and the **loss function** to apply to it, which the training loop then uses to update the model. Everything around it is handled for you: allocating the backend, the forward/backward and optimiser steps, batching, checkpointing, and running validation every N steps. A reference algorithm is only a few dozen lines.

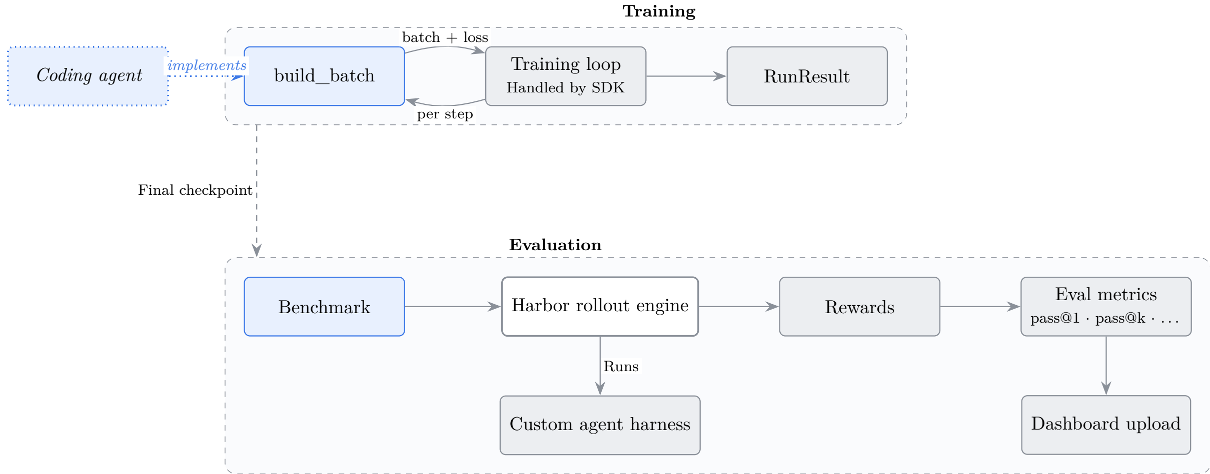

Figure 4: Training and evaluation share one engine. *Training* (top): a coding agent implements `build_batch` (dotted), the one function that is the algorithm. Each step it returns a batch plus the loss function, and the SDK-managed training loop applies it (forward/backward → optimizer → checkpoint), yielding a RunResult. *Evaluation* (bottom): the final checkpoint drives the benchmark through the harbor rollout engine, which runs multi-turn rollouts via a custom agent harness; each rollout yields a reward, the metrics aggregate them (pass@1 / pass@k / . . .), and only *eval* rollouts are uploaded.

Rollouts run through harbor. For RL and other on-policy methods, `build_batch` produces its data by rolling out the current model. Those rollouts run through the harbor framework, which owns the multi-turn rollout engine: concurrency, retries, and trajectory collection. You never write a rollout loop. If a task needs bespoke behaviour, you can plug in your **own agent harness** (any harbor agent) and train a model against it while reusing everything else.

Rewards are your own verifier functions. A verifier scores a single rollout into a number: that number is the reward during RL and the score during evaluation. Writing one is a single registered function with the signature `(model_output, expected, params) -> float`. The SDK ships the common cases (`exact_match`, `contains`, `regex_match`, `tool_calls_match`), and you register your own the same way, so reward design stays in plain Python.

Validation and benchmarks are just RL tasks. A validation set and a test benchmark have the same shape as an RL training task: a set of tasks, where each task is an instruction plus a verifier. Validation runs in-loop every N steps to steer model selection; the benchmark is held out and scored once at the end. Both go through the same harbor engine, and you choose which **metrics** to report (for example `pass@1`, `pass@k`, or mean reward) from the metric registry, or register your own.

8 The workspace: `.evsys`

Think of `.evsys/` as a scratch workspace the SDK creates to do its work in: it holds both the cached data a run needs and the artefacts a run produces.

Datasets and benchmarks are referenced by name or id, never by a raw file path. The first time a run needs one, the SDK pulls it and caches it under `.evsys/` as immutable JSONL, recorded with its version and lineage. Later runs read straight from the cache, so the same id always resolves to the same bytes and experiments stay reproducible and comparable across time and machines. Raw rows are materialised through the run's transform into typed rows on the way in, and the cache is written atomically (a manifest tracks the row count and a completion flag), so a half-finished pull is never mistaken for a real dataset.

The workspace also holds what a run produces: the training rollouts and the run logs are written here on disk. Training rollouts stay local (only evaluation rollouts are uploaded to the dashboard), so the workspace doubles as the on-disk record of everything that happened during a run.

9 Registries: the pluggability layer

Every `kind:` in the YAML resolves through a registry to a class. The pattern is uniform: **implement a protocol → register under a `kind` → reference it in YAML**. No subclassing the library, no fork.

- **algorithm** - `@register_algorithm`, the `run.algorithm.kind`: the training recipe, i.e. the math for one optimisation step (`sft`, `rl`, `sdft`).
- **backend** - `@register_backend`, the `run.backend.kind`: where the model trains (`tinker`, `local`, `mock`).
- **transform** - `@register_transform`, in `data.transforms[].kind`: shapes raw rows into the format the algorithm consumes.
- **data_store** - `@register_data_store`, the `data_store.kind`: where datasets are pulled from and cached.
- **log_store** - `@register_log_store`, the `log_store.kind`: where metrics, hyperparams, and artifact pointers go.
- **metric** - `@register_metric`, in `eval.metrics[]` / `validation.metrics[]`: how rewards aggregate into a score (`pass@1`, `pass@k`, . . .).
- **verifier** - `@register_verifier`, in verifier specs / the RL reward: scores a single rollout.
- **inference_client** - `@register_inference`, the `eval.inference.kind`: how completions are generated at eval time.

10 Using the SDK

There are two ways to drive it.

Directly. Write an experiment config and a short launch script, or use the `evsys` command line. `evsys run config.yaml` validates the config, expands the runs, trains, scores, and records everything to the dashboard, while `evsys list` and `evsys schema` let you discover the available algorithms, backends, verifiers, and metrics together with their parameters.

As a Claude Code plugin. Point Claude at the SDK from inside your own repository (claude -plugin-dir <path-to-evsys-sdk>) and it loads our **training-decider** subagent. The agent reads the full history of past experiments, their hypotheses, conclusions, and metrics, proposes the next educated experiment, scaffolds the config and any custom verifier, metric, or transform code, launches it, and writes back a conclusion. This closes the auto-research loop: your coding agent becomes the researcher, and the SDK gives it the standardised context and the controls to act.

Summary

The **Experiment** carries a hypothesis, expands into one or more **training runs**, and produces a conclusion against a `success_metric`. Each run is the three pillars in sequence: the **Data** surface turns raw sources through ordered `Transforms` into typed rows that carry only data;

the **Training** surface is one contract, `Algorithm.train(ctx) -> RunResult`, with an optional `TrainingLoop/StepBuilder` toolkit over any tinker-compatible `Backend`; the **Evaluation** surface scores checkpoints through the shared harbor rollout engine (a custom agent harness running multi-turn rollouts, scored by verifiers into metrics) with async validation steering selection and only eval rollouts uploaded. Everything is a protocol registered under a `kind` across eight registries, so users extend the system without forking it.