

EasyLocal – Guides

A C++23 framework for local search and metaheuristics

Contents

REST application adapter	4
Enabling the component	4
Generic app Blueprint	4
Run creation envelope	6
Routes and lifecycle	8
Status and result shape	9
Error mapping	10
Bounded completed-run retention	11
Concurrency model	11
Cooperative control and TextUI	12
Security boundary	13
HTTP integration test	13
Semantic search tracing	14
Composite-neighborhood provenance	15
Persistence	15
Overhead benchmark	21
Logging	22
External dependency policy	23
Public target model	23
Dependency resolution	23
CMake options	24
Installation and consumers	24
Architectural boundary	24
ConfigTOML	25
TextUI	25
REST	26
API stability	27
Pre-releases	27
Levels	28
Reproducibility	30
Supported toolchains	30
Roadmap	31
Asynchronous runs on a Session	31
The Pareto front in the frontends	31
A failed run in the trace and in the result	31
Python and Julia bindings	32
Shifting penalty	32
A name for every service	32
A tester on an app without a neighborhood	33
A launcher across SolutionManagers	33
Adaptive neighborhood selection	33
The fixed cost of a neighborhood union	33
Where a first improvement starts its scan	34
Candidate strategies of Tabu Search	34
Cooperative runners	35
Parallel search	35
Kicks, Iterated Local Search and Variable Neighborhood Descent	35
A modelling layer	36
Parameter tuning	36

Experiment configurations	37
Trace micro-benchmarks	37
Delta ablation of a neighborhood explorer	37
Parameters described by reflection (C++26)	38

REST application adapter

`EasyLocal::REST` is an optional Crow-based application adapter. It exposes a configured `EasyLocal` app as a generic Crow Blueprint while keeping HTTP, JSON, Crow, and server lifecycle completely outside `EasyLocal::Core`.

The design follows the same rule used by the `TextUI`:

one run owns freshly bound, mutable `EasyLocal` services; only the immutable `Input` may be shared across runs.

This keeps concurrency at the application boundary instead of making the bound app, `SolutionManager`, neighborhoods, algorithms, or `Runner` state internally synchronized.

Enabling the component

REST is disabled by default:

```
cmake -S . -B build/rest \
  -DEASYLOCAL_ENABLE_REST=ON
```

`EasyLocal` first looks for Crow 1.3 through `find_package`. If it is unavailable, a pinned Crow v1.3.3 plus standalone Asio 1.38.2 may be fetched only when `EASYLOCAL_FETCH_DEPENDENCIES=ON` is explicitly enabled.

Installed consumers request the component explicitly:

```
find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core REST)
target_link_libraries(my_server PRIVATE EasyLocal::REST)
```

A consumer requesting only `Core` neither creates `EasyLocal::REST` nor loads Crow/Asio.

Generic app Blueprint

The REST adapter intentionally does not own a Crow server. It produces one Blueprint per `EasyLocal` application. Multiple applications can therefore be mounted into the same Crow process under distinct prefixes:

```
auto assignment_api = easylocal::rest::blueprint(
    "/assignment", assignment_app, AssignmentCodec{});
auto tsp_api = easylocal::rest::blueprint(
    "/tsp", tsp_app, TspCodec{});

crow::SimpleApp server;
server.register_blueprint(assignment_api.crow_blueprint());
server.register_blueprint(tsp_api.crow_blueprint());

server.bindaddr("127.0.0.1")
    .port(8080)
    .multithreaded()
    .run();
```

Port, middleware, HTTP concurrency, logging, and process lifecycle remain normal Crow concerns.

A problem with the text hooks of [Apps and tools](#) (`read_input`, `write_solution`, `read_solution`) needs no codec: `blueprint(prefix, app, options)` serves it with `text_codec`, and the `Input` and the solutions

are JSON strings in the text of the hooks, the costs JSON numbers (the text of `cost::to_text` for a structured cost).

```
auto api = easylocal::rest::blueprint("/tsp", tsp_app);
// POST /tsp/runners/sa/runs {"input": "5\n0 2 9 10 7\n...", "initial_solution": "0 1 2
3 4"}
```

A codec supplies a JSON boundary of its own; each value it does not handle goes through the problem's text hook, as above:

```
struct Codec
{
    // Or read_input, from a JSON string.
    Input decode_input(const crow::json::rvalue& value) const;

    // Or write_solution, to a JSON string.
    crow::json::wvalue encode_solution(const Input& input, const Solution& solution)
    const;

    // Or a JSON number, or the text of cost::to_text.
    crow::json::wvalue encode_cost(const Cost& cost) const;

    // Optional, or read_solution. Called only when the request contains
    // initial_solution.
    Solution decode_initial_solution(const Input& input, const crow::json::rvalue&
    value) const;

    // Optional. Called only when the request contains a target that is not a
    // string; without it a target is a JSON number, for arithmetic costs only.
    Cost decode_cost(const crow::json::rvalue& value) const;
};
```

The framework owns only the request envelope. `input` and `initial_solution` are otherwise opaque JSON values passed unchanged to the codec. A codec may therefore accept a structured JSON object/array, a JSON string containing an existing text file representation, or any other JSON representation appropriate to the problem. Through a text hook, a value that is not a string, or a text the hook cannot read, is rejected with 422. A decoded `initial_solution` must be valid for the `Input` (the `SolutionManager`'s `is_valid`), or the request is rejected with 422. If `initial_solution` is omitted, the run starts as its start says (below).

The generic Blueprint is mounted explicitly:

```
auto api = easylocal::rest::blueprint(
    "/assignment",
    application,
    AssignmentCodec{},
    {
        .workers = 4,
        .queue_capacity = 32,
        .completed_run_capacity = 64,
    });

crow::SimpleApp server;
server.register_blueprint(api.crow_blueprint());
```

`app_blueprint` is deliberately non-copyable/non-movable because its Crow route callbacks refer to its state. Keep it alive for at least as long as the Crow application uses the registered Blueprint. Codec calls are serialized by the adapter, so a codec need not provide its own synchronization.

The Assignment REST example demonstrates both supported input styles: a structured object with `demand/capacity` and a JSON string containing the existing textual assignment-instance representation.

Run creation envelope

For a Blueprint mounted at `/assignment`, run creation is:

```
POST /assignment/runners/fi/runs
Content-Type: application/json
```

with a generic envelope:

```
{
  "input": {
    "demand": [4, 4, 2],
    "capacity": [5, 5]
  }
}
```

or, for a codec that accepts the existing text representation:

```
{
  "input": "3 2 4 3 2 5 5"
}
```

An application codec may additionally support:

```
{
  "input": { "...": "..." },
  "initial_solution": { "...": "..." }
}
```

Without an `initial_solution`, the envelope may say where the run starts: `"start": "random"`, the `SolutionManager`'s `random_solution()`, drawn from the run's seed, or `"start": "initial"`, its `initial_solution()`. Without either, a run starts from a random solution when the problem has them, from the initial one otherwise, as `cli::run` does; run resources report the start (`"solution"` for a given one). Both fields together, or a start the problem cannot build, are rejected with 422.

```
{
  "input": { "...": "..." },
  "start": "initial"
}
```

The envelope may also carry a seed, a non-negative integer. Stochastic runners receive an RNG seeded with it; deterministic runners ignore it. Without an explicit seed a run uses `blueprint_options::seed + <run id>`, so runs differ but stay reproducible. Status responses report the seed used.

```
{
  "input": { "...": "..." },
```

```
"seed": 12345
}
```

A target cost stops the run as soon as its best cost is at least as good as the target, a lower bound for example (stop_at in C++; every runner honours it). A JSON string is a cost in the syntax of the command line and the TextUI: a number, "[hard, soft]", or the problem's own notation when it provides read_cost (Costs as text). Otherwise, for an arithmetic cost the target is a JSON number, an integer for integral costs, within the range of the cost type (422 otherwise); for other costs the codec decodes it with decode_cost, and without one a request with a target is rejected with 422. Run resources repeat the target, encoded by encode_cost.

```
{
  "input": { "...": "..."},
  "target": 426
}
```

A timeout, in seconds (a non-negative JSON number), stops the run once it has passed (timeout in C++), and max_evaluations, a non-negative integer, once it has made that many evaluations (max_evaluations in C++; a runner's own, if smaller, still applies); run resources repeat them. Any other value is rejected with 422. A service with blueprint_options::max_timeout bounds the time of every run: a longer timeout is rejected with 422, and a run without one gets max_timeout (GET / reports it).

```
{
  "input": { "...": "..."},
  "timeout": 2.5,
  "max_evaluations": 100000
}
```

A run may change the app's parameters, for itself only: the same paths as the command line and the TextUI (runners.<name>.*, cost.*, solution_manager.*, neighborhood.*), listed by GET /parameters. Nested objects and dotted keys are equivalent, and may be mixed:

```
{
  "input": { "...": "..."},
  "parameters": {
    "runners": { "sa": { "temperature": { "cooling_rate": 0.9 } } },
    "cost.weights": [1, 20]
  }
}
```

Values are JSON numbers, booleans, strings, or arrays of numbers, booleans and such arrays ([1, 2], [3]), as in a TOML file; a string inside an array is rejected, since list elements are not quoted. A string is read as the parameter's text, so the values that GET /parameters lists can be sent back as they are. They are applied all or none before the run's initial solution is built: an unknown path or an invalid value rejects the request with 422 and the code invalid_parameters, whose message names each path. Run resources repeat the parameters they were submitted with, by path and as text ("parameters": {"runners.sa.temperature.cooling_rate": "0.9"}), so that a run can be repeated together with its seed.

A successful submission returns 202 Accepted, sets Location to the run resource, and uses id consistently:

```
{
  "id": "42",

```

```

"runner": "fi",
"seed": 42,
"status": "queued",
"cancellation_requested": false,
"progress": {
  "evaluations": 0,
  "iterations": 0
}
}

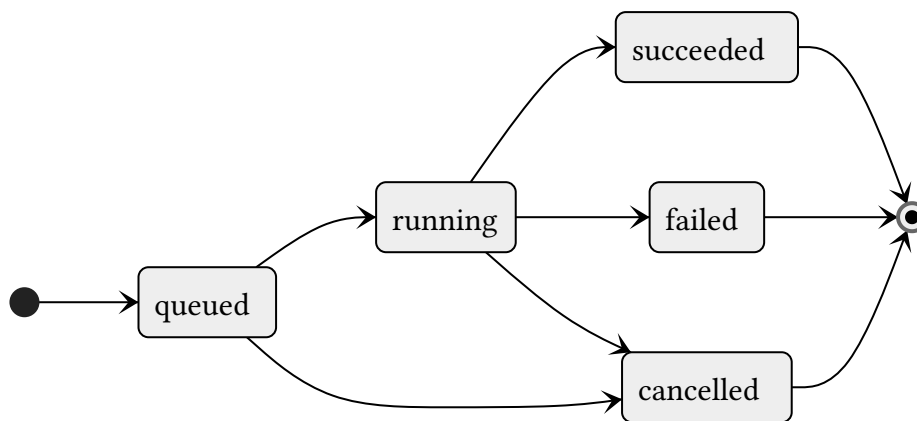
```

Routes and lifecycle

The generic surface under a chosen prefix is:

Method	Route	Meaning
GET	/assignment/	application/executor metadata
GET	/assignment/runners	registered runner names
GET	/assignment/parameters	the app's parameters: path, description, value (as text), kind (boolean, integer, real, limit, text, path, list), domain (as text, such as [1, unlimited]; absent when none is declared), active (whether its condition holds) and condition (as text, with full paths; absent when it has none)
POST	/assignment/runners/<runner>/runs	enqueue a run
GET	/assignment/runs/<id>	inspect status/progress
GET	/assignment/runs/<id>/solution	retrieve terminal solution and cost (and the front, with a Pareto cost)
POST	/assignment/runs/<id>/cancel	request cooperative cancellation
DELETE	/assignment/runs/<id>	forget a terminal run

Runs move through:



Cancellation and deletion deliberately have different semantics. `POST /runs/<id>/cancel` requests cooperative stop and returns 202; the resource remains queryable so the client can observe cancelled and retrieve a partial solution. A queued run is cancelled at once, without a solution, and leaves its place in the queue. `DELETE /runs/<id>` is permitted only after the run is terminal and returns 204 No Content. Deleting an active run returns 409.

Every runner is cancellable: the control is carried by the framework-owned `search_run`, so no per-runner capability is advertised.

Status and result shape

`GET /runs/<id>` has the same generic shape for every problem (an Experimental surface, as [Stability](#) says of the adapters):

```

{
  "id": "42",
  "runner": "fi",
  "seed": 42,
  "status": "running",
  "cancellation_requested": false,
  "progress": {
    "evaluations": 237,
    "iterations": 14,
    "evaluation_limit": 2000,
    "best": 31
  }
}

```

`evaluation_limit` is omitted when the Runner does not report one. `best` is the best cost found so far, encoded as cost is (by the codec's `encode_cost`, else as a number or as text); it is omitted until the run has evaluated its first solution, and is the best of all the runs of a MultiStart or a pipeline (of the stages on the app's cost). Once the run has succeeded, or was cancelled with a partial solution, the status also gives its final progress, its cost and `"solution_url"`, the address of `GET /runs/<id>/solution`, and termination: why the run ended (completed, local_optimum, evaluation_budget_exhausted, cancelled, target_reached, idle_limit_reached, time_limit_reached). A run is cancelled when its termination says so: one whose cancellation came after it ended on its own terms is succeeded, as is the run of a custom runner whose result has no termination (completed). A failed run gives `"error": {"code": "run_failed", "message": ...}` instead.

```
{
  "id": "42",
  "runner": "fi",
  "seed": 42,
  "status": "succeeded",
  "cancellation_requested": false,
  "progress": { "evaluations": 2000, "iterations": 118, "evaluation_limit": 2000,
  "best": 27 },
  "termination": "evaluation_budget_exhausted",
  "cost": { "...": "..." },
  "solution_url": "/assignment/runs/42/solution"
}
```

GET /runs/<id>/solution gives the solution and its cost, whose values remain problem-specific and are produced by the codec, with the request the run was submitted with (seed, and target, timeout, max_evaluations and parameters when it had them):

```
{
  "id": "42",
  "runner": "fi",
  "seed": 42,
  "max_evaluations": 2000,
  "parameters": { "runners.fi.max_evaluations": "5000" },
  "status": "succeeded",
  "cost": { "...": "..." },
  "solution": { "...": "..." }
}
```

Before the run ends it answers 409 result_not_ready; for a run that ended without a solution, cancelled while it was queued, 409 no_solution; for a failed run, 409 run_failed.

With a cost::pareto cost the result also has front, the non-dominated solutions the run reached, ordered by their objectives, each with its cost, encoded as cost and solution are (by the codec or the text hooks):

```
{
  "id": "42",
  "runner": "plahc",
  "status": "succeeded",
  "solution": { "...": "..." },
  "cost": "[0, 9]",
  "front": [
    { "cost": "[0, 9]", "solution": { "...": "..." } },
    { "cost": "[1, 8]", "solution": { "...": "..." } }
  ]
}
```

A result without a front, with any other cost, has no front field.

A cooperatively cancelled run may expose the same result shape with "status": "cancelled"; the solution is then the valid partial solution returned by the controlled Runner.

Error mapping

Every response with an error status (4xx, 5xx) has the same envelope; the status of a failed run, a 200 response, carries one too, under error:

```
{
  "error": {
    "code": "unknown_runner",
    "message": "runner 'missing' is not registered"
  }
}
```

The generic mapping is:

HTTP	Meaning
400	syntactically invalid JSON, or arrays and objects nested deeper than 64 levels (<code>invalid_json</code>)
404	unknown runner or run (<code>unknown_runner</code> , <code>run_not_found</code>)
409	valid operation in the wrong run state/capability (<code>result_not_ready</code> , <code>run_not_terminal</code> , <code>run_not_active</code>), the solution of a run that ended without one (<code>no_solution</code>), or of a run that failed (<code>run_failed</code>)
422	valid JSON but invalid run envelope/domain data (<code>invalid_run_request</code>), or parameters that do not apply (<code>invalid_parameters</code>)
503	bounded solver queue full (<code>queue_full</code>)
500	unexpected adapter/application failure (<code>internal_error</code>)

Codec/domain validation should throw `std::invalid_argument` for client-supplied semantic errors; those are mapped to 422. Other unexpected exceptions while creating a run are treated as internal failures.

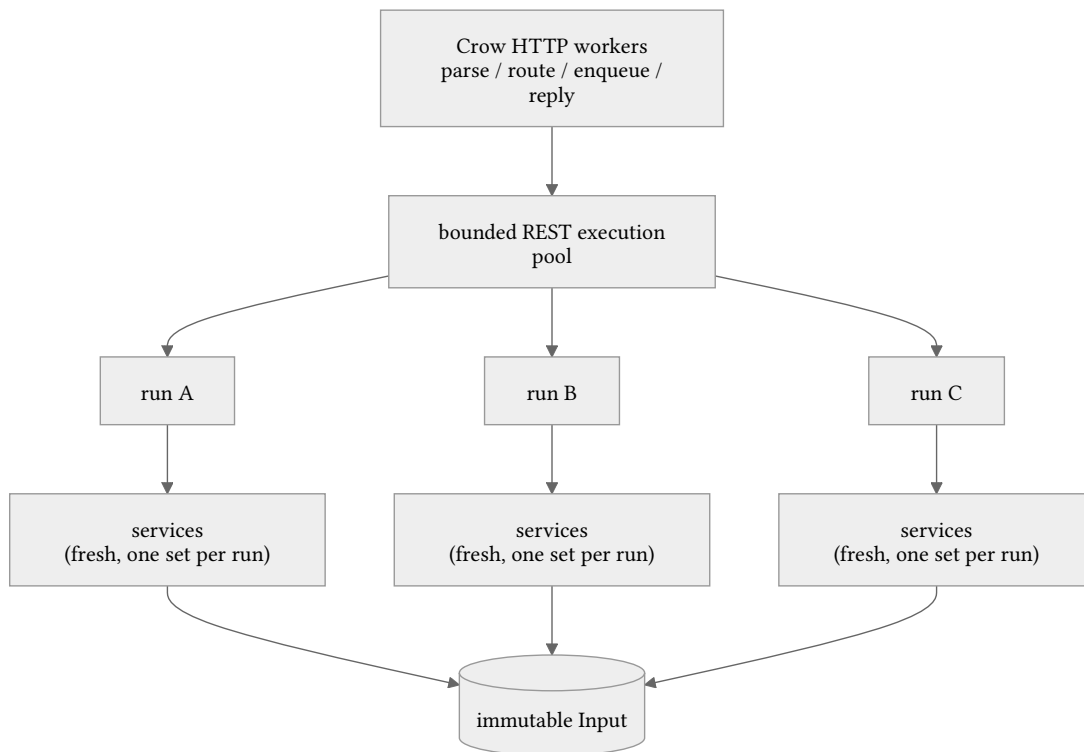
Bounded completed-run retention

The waiting queue and completed-run history are independently bounded. `blueprint_options::completed_run_capacity` defaults to 64 and must be positive. Only terminal records participate in history eviction; queued/running runs are never evicted. Once the bound is exceeded, the oldest terminal record is removed. Clients may free a terminal record earlier with `DELETE /runs/<id>`.

This retention is intentionally an in-memory convenience for result/status retrieval, not a persistent history system. Durable audit/history belongs to the surrounding web/infrastructure layer.

Concurrency model

Crow concurrency and solver concurrency are intentionally separate:



Crow request threads do not execute a CPU-bound local search to completion. They decode the request, materialize the immutable Input and initial Solution, enqueue work, and return the run identifier.

Each accepted job owns a Session on the run's Input, created when the run is submitted: configured with the run's parameters, then bound to the Input once, with the initial solution and the run's seed. The worker calls `session.run("name", with(control))`, which runs on the services of that Session, bound for that run alone, and stores the resulting solution and its cost, encoded once for every status poll. Mutable SolutionManager, neighborhood, algorithm, Runner, RNG, and Solution state therefore belongs to that run only. No mutex is added to those Core objects and no `Clone()` protocol is required.

The default worker count is `max(1, hardware_concurrency() - 1)` and the default waiting-queue capacity is 64; both are adapter policy, configurable per Blueprint, and must be positive: the Blueprint's constructor rejects a zero with `std::invalid_argument` before any worker starts.

Cooperative control and TextUI

Core exposes a small std-only control object:

```

std::stop_source stop;
auto observer = [](const easylocal::run_progress& progress) { /* observe */ };
easylocal::run_control control{stop.get_token(), observer};

```

The observer is non-owning and valid only for the controlled run. It is passed as the trailing run option, `run(solution, ..., easylocal::with(control))`. `control.observe_best_cost<Cost>(best)` adds a second observer, called with each better cost of type `Cost` the run finds (a solver's runs together: a cost better than those of the runs before). `shared_run_progress` and `shared_best_cost<Cost>` keep the last report for another thread: the REST service stores them in each run and reads them for each status, the TextUI for its progress line. Algorithms never see the control directly: `search_run::should_stop()` checks it and the `search_run` primitives report progress, so every runner honours cancellation through

the same contract. A run without an explicit control uses an empty `run_control`, whose checks reduce to null tests.

TextUI and REST therefore share the same architectural contract without sharing a threading subsystem:

```
Core:      fresh services per run: Session::run, app.run("name", ...)
TextUI:    one background run owned by the frontend
REST:      bounded pool of background runs owned by the adapter
```

Security boundary

`EasyLocal::REST` is not an Internet edge/security framework. TLS termination, authentication, authorization, request-rate limiting, reverse-proxy policy, ingress controls, and deployment hardening belong to the surrounding network infrastructure.

The adapter still performs ordinary protocol/application validation and relies on maintained Crow/Asio HTTP parsing. `Queue/history` bounds and `max_timeout` are resource-control semantics, not a substitute for edge security. The examples listen on `127.0.0.1` only, as a service should unless it is meant to be reached from other hosts. Without `max_timeout` a client may ask for a run without a time limit, which holds a worker until it ends or is cancelled. The body size is left to Crow and the proxy in front of it.

No Crow, Asio, HTTP, or JSON type appears in a Core signature. The architecture test also prevents optional adapters from reaching into `easylocal/detail/*`.

HTTP integration test

When REST is enabled on a Unix-like host with `curl`, CTest registers `easylocal.rest-http`. It starts the real Assignment Crow example and exercises the public surface with HTTP requests: discovery, malformed JSON, semantic 422 errors, structured and opaque-text input decoding, `Location/run` IDs, status and nested progress, result retrieval, active-run delete rejection, cooperative cancel, cancelled partial result, terminal deletion, and bounded retention.

The test carries both `integration` and `rest-http` labels. `scripts/build-and-test.sh --with-rest` runs it explicitly even in the normal fast profile; `--integration` additionally enables the remaining integration tests such as the installed-package consumer.

Semantic search tracing

Tracing (`<easylocal/trace.hpp>`) produces typed search events suitable for trajectory analysis; it is not a progress log, and it is separate from [logging](#) (`<easylocal/utils/logging.hpp>`), which carries text messages to an application's logger.

An ordinary `run(...)` call instantiates a `trace::null_tracer`. Its `observes<Event>` value is false, so event construction and emission are removed with `if constexpr`. Applications that need data pass a tracer explicitly as the trailing run option:

```
using cost_type = decltype(search)::cost_type; // as a Runner, LocalSearch or Pipeline
has
easylocal::trace::memory_recorder<cost_type> trace;
auto result = search.run(initial_solution, easylocal::with(trace));
```

Core events are emitted by the framework-owned `easylocal::search_run`, not by the individual algorithms: `start()` emits `run_started`, `evaluate_move()` emits `move_evaluated`, `commit()` emits `move_accepted`, `finish()` emits `run_finished`, with the termination reason (preceded by `local_optimum` when that is the reason), and `random_move()` forwards `neighborhood_selection` events. Every built-in runner emits `incumbent_updated` at each new best cost: those that track a best-so-far solution, such as Simulated Annealing, through `incumbent_updated()`, and those whose current solution is the best (Hill Climbing, First and Best Improvement) through `commit_improvement()`; custom events can be sent with `run.emit(event)`.

`start()`, `commit()` and `evaluate_solution()` also emit `solution_visited`, with the hash of the solution reached (`solution_hash`, see [SolutionManager](#)), its cost and `previous_hash`, the hash of the solution the move was applied to (0 at the start of a run and for a solution no move reached, such as a sample of Pareto Late Acceptance's history): the nodes and the edges of Search Trajectory Networks and Local Optima Networks. It is emitted only when the problem has a solution hash, and the hash is computed only when the tracer observes the event, so the hash costs nothing to a run that does not record it. The 64-bit hash is a `u64` in ELTR and, in JSONL and in the output of `eltr.py`, a string of 16 hexadecimal digits ("00000000feedface"): as a JSON number, JavaScript and jq would round it to a double.

A recorder has a cost type, its `cost_type`: it observes the core events of that cost and those without a cost (`run_context`, `neighborhood_selection`, the tabu events, `temperature_changed`), not the events of a run on another cost, which it would otherwise drop or convert. A run rejects at compile time a recorder whose `cost_type` is not its own; a pipeline gives a stage on another cost (an `until_feasible()` stage on the hard cost) the recorder's events without a cost only. A tracer of your own says with `observes<Event>` which events it receives: `trace::emit` rejects at compile time a tracer whose `observes<Event>` is true but whose `emit()` does not take the event, such as an `emit` for another cost type.

Every recorder observes every core event of its cost; a run that builds no trajectory or local optima network leaves the visited solutions out at compile time with `trace::without`, which wraps any tracer and hides the given event templates from the search:

```
easylocal::trace::binary_recorder<cost_type> recorder{out};
auto trace =
easylocal::trace::without<easylocal::trace::event::solution_visited>(recorder);
runner.run(solution, rng, easylocal::with(trace));
```

The events without a cost, which are types rather than templates, are hidden by their type, `without<event::neighborhood_selection, event::tabu_escape>(recorder)`; a wrapper is itself a tracer, so the two forms combine by wrapping one in the other.

A run that an exception ends, thrown by a hook of the problem (a cost component, a move, the Input) or by a tracer, emits no `run_finished`: the recorder keeps what it received, a binary recorder writes its pending block when it is destroyed, and `eltr.py --format summary` marks the run `"unfinished": true`. The exception reaches the caller of `run()` or `solve()` ([Solvers](#)).

The solvers emit `run_context` before each run they start, with the name and index of the pipeline stage (empty and 0 outside a pipeline) and the attempt, or the start of a MultiStart, from 0: in a trace of a whole solve, it tells the runs apart. It carries no cost, so a tracer receives it from the stages that run on another cost, such as a pipeline's `until_feasible()` stage on the hard cost, whose other events a recorder of the full cost does not observe. A run started outside a solver, with `runner.run(...)` or a registered runner of an app, has no `run_context`.

Tabu search adds `aspiration_applied`, when the move just applied was tabu and admitted by the aspiration criterion, `tabu_escape`, after the random moves of a reactive list's escape, with the number applied (the run may stop before the escape ends), and `tabu_tenure_changed`, with the previous and the new tenure of a list with one tenure for all its moves, at the start of the run (previous tenure 0) and at each change; `RandomFoo` draws its first tenure at the first move, and traces it then.

Simulated Annealing adds `temperature_changed`, with the previous and the new temperature, at the start of the run (previous temperature 0, after a calibration) and whenever the schedule's temperature changes: a cooling or a reheat. It reads the temperature only for a tracer that observes the event.

Composite-neighborhood provenance

Moves produced by `neighborhood_union` already carry the originating child in their variant type. Tracing extracts that information recursively, so nested unions produce routes such as `[0, 2, 1]` without adding runtime tags to normal moves.

Random union sampling can also emit `neighborhood_selection` events containing:

- the selected child and hierarchical route;
- configured child bias;
- total active bias at that decision;
- conditional branch-selection probability;
- attempt number and whether the selected child produced a move.

This deliberately describes branch selection, not the probability of an individual move: a child neighborhood is not required to sample its own moves uniformly.

The data is intended to support trajectory statistics, Search Trajectory Networks, Local Optima Networks, and later experiments with adaptive neighborhood-sampling policies. Adaptive/RL policies are not part of the current tracing machinery.

Persistence

`trace::memory_recorder<Cost>` owns copies of recorded events. Its neighborhood routes are copied only when tracing is active. `recorder.replay(tracer)` emits the recorded events again, in order, to another tracer: to a streaming recorder below, it writes the completed trace in that recorder's format. `trace::write_jsonl(out, recorder)` is the replay to a `jsonl_recorder`: one JSON object per line, as during the run. File naming, directory layout and stream ownership are left to the application.

For long runs, `EasyLocal` provides constant-memory streaming recorders. `trace::jsonl_recorder<Cost>` writes a human-readable JSONL representation. Its first line, written at construction, is a header, the same line `eltr.py` starts its output with, without the ELTR cost layout: the format version and the metadata of `jsonl_options`, whatever tells the run apart, as text.

```

easylocal::trace::jsonl_recorder<cost_type> trace{
    out,
    {.metadata = {{{"instance", "ta001"}, {"seed", "42"}}}};
    // {"event":"trace","version":1,"metadata":{"instance":"ta001","seed":"42"}}

```

`write_jsonl(out, recorder, options)` takes the same options.

No event carries a time: a run that does not ask for it reads no clock. For anytime and time-to-target analyses, a streaming recorder writes timestamps when its options set `timestamps`: each core event then ends with `elapsed_ns`, the nanoseconds of the steady clock since the recorder was constructed, a field of the JSONL line or, in ELTR, a u64 field the header adds to every core event schema. The time of an event within its run is its `elapsed_ns` minus that of the run's `run_started`. Application events carry no timestamp, and neither does the memory recorder, whose replay happens after the run.

```

easylocal::trace::binary_recorder<cost_type> trace{out, {.timestamps = true}};
easylocal::trace::jsonl_recorder<cost_type> lines{json_out, {.timestamps = true}};

```

The canonical experimental format is the versioned ELTR binary stream. It describes itself: a header gives the metadata of the run, the layout of the costs and the fields of every event, so a reader needs nothing but the file. Its records are tagged and length-prefixed, so readers can skip event types they do not understand.

`trace::binary_recorder<Cost>` is the synchronous recorder. It encodes events directly into a contiguous block (256 KiB by default) and writes a block at a time instead of calling `std::ostream` for each field. `trace::async_binary_recorder<Cost>` uses the same encoder and byte format, but hands completed blocks to one background writer thread. The producer owns one block while a bounded pool of preallocated blocks connects it to the writer; if the writer falls behind, the producer blocks rather than losing events. Event order is therefore deterministic and tracing is lossless.

Both recorders write and flush the header when they are constructed, so the file of a run that crashes is still a trace: it decodes up to the last block written, and loses only the events still buffered, at most a block (`block_size`) for the synchronous recorder, the current block and the queued ones for the asynchronous one. `eltr.py` reports a file with no bytes at all as an empty trace.

```

std::ofstream out{"run-0042.eltrace", std::ios::binary};
easylocal::trace::async_binary_recorder<cost_type> trace{out};
auto result = search.run(initial_solution, easylocal::with(trace));
trace.flush(); // drain the writer and expose I/O failures
if (!trace.good()) { /* handle output failure */ }

```

An output error of the asynchronous recorder stops the recording, not the search: from then on `emit` drops the events, `good()` is false and `flush()` throws `std::ios_base::failure`, also when the final flush of the stream is what fails.

Block size and the number of queued blocks are explicit policy knobs, not global state, and the options carry the metadata written in the header: whatever tells the run apart, as text.

```

easylocal::trace::binary_recorder_options options{
    .block_size = 256 * 1024,
    .async_queue_blocks = 4,
    .metadata = {{{"instance", "ta001"}, {"runner", "ts"}, {"seed", "42"}}},
};
easylocal::trace::async_binary_recorder<cost_type> trace{out, options};

```

The asynchronous recorder allocates its queued blocks, and the one the search fills, at construction: `async_queue_blocks` is finite, and `easylocal::unlimited` throws `std::invalid_argument`.

The async recorder is deliberately single-producer: it is designed to observe one search trajectory. Synchronization happens at block boundaries, not for every event. The writer thread may reduce search-thread stalls when persistence is slower than encoding, but it does not make I/O free; benchmarks therefore report producer time separately from total drain time.

The memory recorder is intended for tests, short traces and in-process analysis. JSONL is convenient for inspection and interchange, while ELTR is preferred when a run may emit millions of events and trace volume matters. Core does not choose a trace path and does not depend on a JSON, binary-serialization or logging library.

From the command line

`cli::run` records the run with `--trace <file>`: JSON Lines for a `.jsonl` name, ELTR otherwise, with timestamps, and with the program and every parameter (the runner that runs, the seed, the instance, the app's parameters) as metadata. The recorder has the app's cost type: a pipeline stage on another cost, such as an `until_feasible()` stage on the hard cost, sends only its `run_context`. A file that cannot be written is an error, after the run when the writes fail. The TextUI and the REST service do not record traces; a program records one from a Session with `session.run(name, with(recorder))`.

```
easylocal_tutorial_cli --instance five.tsp --runner sa --trace sa.eltrace
scripts/eltr.py sa.eltrace --format summary
```

ELTR layout

All integers are little-endian; a string is a u32 size and UTF-8 bytes. A stream starts with ELTR, the format version as a u32 (1) and the header: its size (u32), then

- the metadata: a u32 count, then a key and a value string each;
- the cost layout: a field list;
- the core event schemas: a u32 count, then each event's tag (u8), name (string) and field list.

A field list is a u32 count, then each field's name (string) and type (u8): 1 u8, 2 i8, 3 u16, 4 i16, 5 u32, 6 i32, 7 u64, 8 i64, 9 f32, 10 f64, 11 bool, 12 string, 13 bytes (a u32 size and the bytes), 14 route (a u32 count and a u32 per level), 15 cost (the fields of the cost layout). A scalar cost is one field with an empty name; the fields of a structured cost are named by their path, `hard.0` for the first level of the hard part.

The records follow, each a tag (u8), its payload size (u32) and the payload, the fields of its tag's schema in order. Tag 0 is a schema record (the described tag, name and field list), written before the first record of an application event that describes itself. The core events are:

Tag	Event	Fields
1	run_started	cost
2	move_evaluated	evaluations, iterations, current_cost, candidate_cost, neighborhood
3	move_accepted	evaluations, iterations, previous_cost, cost, neighbor- hood
4	incumbent_updated	evaluations, iterations, previous_cost, cost
5	local_optimum	evaluations, iterations, cost
6	neighborhood_selection	attempt, child, bias, active_bias_total, conditional_probability, produced_move, neighbor- hood
7	run_finished	evaluations, iterations, cost, termination (its name, as to_string writes it)
8	solution_visited	evaluations, iterations, hash, cost, previous_hash
9	aspiration_applied	evaluations, iterations, cost
10	tabu_escape	evaluations, iterations, moves
11	tabu_tenure_changed	evaluations, iterations, previous_tenure, tenure
12	run_context	stage (string), stage_index, at- tempt
13	temperature_changed	evaluations, iterations, previous_temperature (f64), temperature (f64)
128–255	application events	as their schema says, if they have one

With timestamps, every core event ends with `elapsed_ns` (u64). The counters, the stage index and the attempt are u64, the biases and the probability f64, `produced_move` a bool, the neighborhoods routes; the header has the authoritative list.

Decoding ELTR

`scripts/eltr.py` decodes a trace (standard-library Python, python3 or `uv run`) from its own description, with no options. By default it writes JSON Lines: a first trace line with the format version, the metadata and the cost layout, then the records with the field names of `jsonl_recorder`, so a decoded binary trace feeds the same tools as a JSONL one:

```

scripts/eltr.py run-0042.eltrace > run-0042.jsonl
scripts/eltr.py run-0042.eltrace --events incumbent_updated,run_finished
scripts/eltr.py run-0042.eltrace --format summary # metadata, event counts, runs
scripts/eltr.py run-0042.eltrace --format stn # search trajectory network
scripts/eltr.py run-0042.eltrace --format schema # what the trace records

```

A cost decodes to a number, or to an object nested as its fields are named, with lists for numbered levels: {"hard": [0, 2], "soft": 13.5}; NaN and the infinities become null, as in the JSONL recorder, so the output is strict JSON. An application event without a schema is kept as its tag and the hexadecimal payload, and a run interrupted mid-record is read up to its last whole record with --allow-truncated. The summary lists the runs in order, each with its initial and final cost, its effort and its termination, and with its stage, stage index and attempt when a run_context came before it. As a module, eltr.Trace(stream) reads the header (metadata, cost_fields, schemas) and iterates over the records.

The stn format builds the network from the solution_visited events, which are recorded when the problem has a [solution hash](#): one node per distinct hash, with its cost, its number of visits and the runs that visit it, and one edge per move, from its previous_hash to its hash, with its count and runs; a visit no move reached has no edge, so an algorithm that keeps several solutions gets the right edges. A run is its index in the trace, from 0, as the summary lists it (with its stage and attempt), and starts and ends give the first and the last solution each run visits: the network of a whole solve, or of several runs in one file, keeps them apart.

Extending ELTR without touching EasyLocal

The binary encoder is intentionally small enough to customize in application code. The default cost writer, default_binary_cost_writer<Cost>, encodes an arithmetic cost, a cost::lexicographic, a cost::pareto and a cost::hierarchical, nested as they are. Another cost needs a writer, which receives a binary_record_writer, whose primitive operations (u8, i8, u16, i16, u32, i32, u64, i64, f32, f64, boolean, bytes, string and route) always use the ELTR representation, and describes what it writes with fields(), for the header. The writer does not calculate payload sizes and does not know about block or thread management:

```

struct cost_binary
{
    void operator()(
        easylocal::trace::binary_record_writer& out,
        const Cost& cost) const
    {
        out.i64(cost.hard_value());
        out.i64(cost.soft_value());
    }

    static std::vector<easylocal::trace::binary_field> fields()
    {
        using enum easylocal::trace::binary_type;
        return {"hard", i64}, {"soft", i64};
    }
};

easylocal::trace::binary_recorder<Cost, cost_binary> trace{out, cost_binary{}};

```

Applications can also add their own semantic event types through ADL. Tags 1..127 are reserved by EasyLocal; user_binary_event_tag<N>() maps application indices 0..127 onto tags 128..255:

```

namespace my_problem
{
    struct weight_changed
    {
        std::uint64_t iteration;
        double weight;
    };

    constexpr std::uint8_t binary_event_tag(const weight_changed&)
    {
        return easylocal::trace::user_binary_event_tag<0>();
    }

    void encode_binary_event(
        easylocal::trace::binary_record_writer& out,
        const weight_changed& value)
    {
        out.u64(value.iteration);
        out.f64(value.weight);
    }

    // Optional: the schema, written before the first record of the tag.
    easylocal::trace::binary_event_schema describe_binary_event(
        std::type_identity<weight_changed>)
    {
        using enum easylocal::trace::binary_type;
        return {"weight_changed", {{"iteration", u64}, {"weight", f64}}};
    }
} // namespace my_problem

```

No EasyLocal specialization is required. A binary recorder automatically advertises `observes<my_problem::weight_changed>` and the ordinary `trace::emit(...)` path can emit it. Application events are ELTR records only: the memory and JSONL recorders, which have no encoding for them, do not observe them. This keeps problem-specific instrumentation close to the problem code while the recorder owns buffering and persistence. Tag ownership is application-level metadata; independent extensions should coordinate their user-tag assignments when they share a trace schema.

A cost writer or an `encode_binary_event` that throws leaves no part of its record in the trace: the binary recorders drop the whole record (and the schema written before it, which the next record of the tag writes again), and the JSONL recorder formats a line before writing it at once. The exception reaches the caller of `emit`.

The default JSON cost writer, `default_json_cost_writer`, writes the costs the default ELTR writer encodes, in the shape `eltr.py` decodes them to: a number as the shortest text that reads back to the same value, NaN and the infinities, which JSON has no numbers for, as null, a `cost::lexicographic` or a `cost::pareto` as the array of its levels and a `cost::hierarchical` as `{"hard": ..., "soft": ...}`, nested as the types are: `{"hard": [0, 2], "soft": 13.5}`. Another cost needs a small compile-time writer, which owns the JSON representation and adds no virtual dispatch:

```

struct cost_json
{
    void operator()(std::ostream& out, const Cost& cost) const
    {
        out << "{ \"hard\": " << cost.hard_value()
            << ", \"soft\": " << cost.soft_value() << '}'';
    }
};

```

```
easylocal::trace::jsonl_recorder<Cost, cost_json> trace{out, cost_json{}};
```

The same writer can be passed as the third argument to `write_jsonl(...)` when a `memory_recorder` is serialized after the run.

Overhead benchmark

The tracing overhead is measured in [easylocal-benchmarks](#) (infrastructure/) at every release, and shown on the [Benchmarks](#) page. `easylocal_trace_benchmark` compares the same First Improvement workload under:

- ordinary baseline execution;
- explicit `null_tracer`;
- counter-only tracing;
- the full in-memory recorder;
- block-buffered ELTR binary recording to a discard stream;
- asynchronous ELTR recording, with producer and total drain time reported separately;
- synchronous and asynchronous ELTR recording to temporary files.

JSONL is intentionally not part of the timing comparison. Its formatting cost is a property of the textual serialization policy and obscures the lower-level trace boundary that this regression probe is intended to track. JSONL functionality is still covered by deterministic correctness tests.

The benchmark checksum consumes the final cost, termination reason, evaluation count, and solution contents. This prevents optimizer dead-code elimination from making tracer specializations incompatible. Temporary-file modes flush the C++ stream but deliberately do not call `fsync`; they measure cost visible to the search process, including OS page-cache interaction, rather than a durability guarantee.

A second target, `easylocal_trace_cost_encoding_benchmark`, isolates ELTR cost encoding using equivalent `move_evaluated` events for three representative cost shapes: scalar `int64`, a two-component lexicographic cost, and a hierarchical cost containing that hard lexicographic branch plus one soft component. It reports nanoseconds and serialized bytes per event.

These numbers are regression diagnostics, not cross-machine performance promises.

Logging

Experimental

The logging API is [Experimental](#): the library itself emits no records yet, and the API may change when it does.

EasyLocal Core provides a deliberately small, dependency-free logging boundary in `<easylocal/utils/logging.hpp>`: records with a level and an origin, sent to one process-wide sink that an application can route into its own logging system, without making Core depend on a logging library.

A log record contains:

- level (trace, debug, info, warning, error);
- origin (framework or application);
- a category string;
- the message;
- `std::source_location` captured at the call site.

The active sink is a plain `noexcept` function pointer. There is no `std::function`, virtual dispatch, dynamic allocation, or formatting machinery in the dispatch path. The default sink writes warnings and errors to `stderr`, one write per record; installing `nullptr` disables dispatch completely.

```
#include <easylocal/utils/logging.hpp>

void my_sink(const easylocal::logging::record& entry) noexcept
{
    // Bridge to the application's logger here. The string views in entry are
    // valid only for this call.
}

int main()
{
    const auto previous = easylocal::logging::set_sink(&my_sink);

    easylocal::logging::emit(
        easylocal::logging::level::info,
        "application.model",
        "instance loaded");

    easylocal::logging::set_sink(previous);
}
```

The sink is process-wide and is stored atomically so it can be replaced safely. A sink is invoked synchronously and must not throw; an adapter for a logging library that may throw should catch exceptions inside the bridge.

The library has no log sites today: the framework origin is reserved for them. When it has some, they will stay on cold paths (configuration, diagnostics), never in the search loops, so an application that never calls `emit` pays nothing in a search.

External dependency policy

EasyLocal keeps its algorithmic and configuration core independent from third-party libraries. External packages may be used only by optional adapter targets whose dependencies are visible to consumers through those targets.

Public target model

EasyLocal::Core is the canonical public target. It is header-only, requires C++23, and has no third-party link dependencies.

Optional integrations use separate package components and targets. The component model is intentionally one-way: adapters depend on Core, never the reverse. Current and reserved names are:

Package component	Target	Purpose
Core	EasyLocal::Core	dependency-free framework and std-only frontends
ConfigTOML	EasyLocal::ConfigTOML	TOML configuration adapter
ConfigYAML	EasyLocal::ConfigYAML	YAML configuration adapter
Logging	EasyLocal::Logging	external logging integration
TUI	EasyLocal::TUI	FTXUI-based textual tester frontend
REST	EasyLocal::REST	Crow-based HTTP/JSON application adapter

ConfigTOML is implemented with toml++ 3.4.x, TUI with FTXUI 7.x, and REST with Crow 1.3.x plus standalone Asio. The component names ConfigYAML and Logging are reserved.

Dependency resolution

An optional adapter follows this order when it is enabled:

1. try to use an already installed package through `find_package`;
2. if the package is unavailable and `EASYLOCAL_FETCH_DEPENDENCIES=ON`, the adapter may use an explicit `FetchContent` fallback;
3. otherwise configuration fails with a diagnostic naming the missing package and the EasyLocal feature that requested it.

`EASYLOCAL_FETCH_DEPENDENCIES` defaults to `OFF`. Configuring EasyLocal must never initiate network access unless the caller explicitly enables that option. When an optional dependency is fetched, the installed EasyLocal component must remain self-contained and relocatable. The adapter may install a private copy of header-only dependency artifacts under the EasyLocal install tree instead of assuming that a `FetchContent` subproject provides install rules. When the dependency was found in the system instead, the installed EasyLocal component retains a normal `find_dependency` requirement on that external package. Vendoring third-party source into the EasyLocal repository is not the default strategy and requires a separate design decision.

Feature switches use `EASYLOCAL_ENABLE_<FEATURE>` names and are `OFF` by default when the feature introduces a third-party dependency. ConfigTOML first tries `find_package(tomlplusplus 3.4 CONFIG)` and may fetch pinned `v3.4.0`; TextUI first tries `find_package(ftxui 7 CONFIG)` and may fetch pinned `v7.0.3`. REST first tries `find_package(Crow 1.3 CONFIG)` and, when explicitly allowed to

fetch dependencies, uses pinned Crow v1.3.3 plus standalone Asio 1.38.2. All fallbacks require `EASYLOCAL_FETCH_DEPENDENCIES=ON`.

CMake options

Option	Default	Effect
<code>EASYLOCAL_BUILD_EXAMPLES</code>	ON at the top level	builds the example programs
<code>EASYLOCAL_BUILD_TESTS</code>	ON at the top level	builds the test suite
<code>EASYLOCAL_STRICT_WARNINGS</code>	ON at the top level	compiles the tests and examples with <code>-Wall -Wextra -Wpedantic (/W4 /permissive- with MSVC)</code>
<code>EASYLOCAL_WARNINGS_AS_ERRORS</code>	OFF	makes those warnings errors (the presets set it)
<code>EASYLOCAL_ENABLE_COVERAGE</code>	OFF	instruments the tests and examples for coverage (GCC and Clang)
<code>EASYLOCAL_ENABLE_CONFIG_TOML</code> , <code>EASYLOCAL_ENABLE_TUI</code> , <code>EASYLOCAL_ENABLE_REST</code>	OFF	build the optional components
<code>EASYLOCAL_FETCH_DEPENDENCIES</code>	OFF	lets a component fetch its pinned dependency when it is not installed

The warning and coverage options apply only to EasyLocal's own tests and examples, never to a project that consumes the headers.

Installation and consumers

The installed package supports component-aware discovery. The dependency-free core can be requested explicitly:

```
find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core)
target_link_libraries(my_solver PRIVATE EasyLocal::Core)
```

Without components, `find_package(EasyLocal CONFIG REQUIRED)` gives the Core as well.

Only adapters built and installed by the producer may report their package component as available. Requesting an unavailable component must fail during `find_package`; it must not silently fall back to Core or download dependencies in the consumer project. Optional targets are loaded lazily: requesting only `COMPONENTS Core` does not create `EasyLocal::ConfigTOML`, `EasyLocal::TUI`, or `EasyLocal::REST`, even when the installation contains those components. Each optional component is exported through its own target file.

Architectural boundary

Third-party types must not leak into Core Runner, neighborhood, search, app, check, Session, parameter, or configuration-tree contracts. An adapter translates between its external library and stable public EasyLocal APIs. In particular:

- optional adapters may include public `easylocal/...` headers, but must not reach into `easylocal/detail/...`;
- TOML/YAML adapters produce the same textual overrides used by the std-only configuration frontends;
- TextUI consumes `app/Session/check` facilities rather than becoming part of Runner or search algorithms;
- REST owns all Crow/HTTP/JSON/server concerns outside Core and translates requests into public `app` and `Session` operations; no Crow, HTTP, or JSON type may enter a Core signature;
- a logging integration must not make Core depend on the selected logging or formatting library.

The boundary is checked by a deterministic architecture test: Core headers may not include TextUI, ConfigTOML, REST, FTXUI, toml++, or Crow, and optional adapter headers may not include `easylocal/detail/*`. This keeps `#include <easylocal/easylocal.hpp>` and `EasyLocal::Core` usable in a pure standard-library consumer regardless of which optional integrations exist.

Bound apps and bound runners borrow an `Input` by `const&`. `Input` is the application-boundary term (`input_type`, `input()`). Binding a temporary `Input` is rejected at compile time so an adapter cannot accidentally create a bound app that outlives its `Input`.

ConfigTOML

`EasyLocal::ConfigTOML` is the first concrete optional adapter. It links `EasyLocal::Core` and `tomlplusplus::tomlplusplus`, but the reverse dependency does not exist. The adapter header is installed only when the feature is built:

```
#include <easylocal/adapters/toml.hpp>

auto source = easylocal::config::load_toml_file("solver.toml");
```

TOML tables are flattened to the same dotted paths as the command line. Native strings, integers, floating-point values, booleans, and numeric/bool arrays are converted to owned textual overrides. TOML date/time values, arrays of tables, and string arrays are rejected with adapter diagnostics rather than silently changing semantics. Typed lookup, parsing, cross-field validation, and atomic commit remain the responsibility of `parameter_set::apply`.

CI verifies `ConfigTOML` through both supported dependency providers: a system-installed `toml++` with fetching disabled, and a forced `FetchContent` path with `CMAKE_DISABLE_FIND_PACKAGE_tomlplusplus=TRUE`. The system-provider install keeps a normal `find_dependency(tomlplusplus)` requirement. The `FetchContent` provider installs a private copy of `toml++` headers under `include/easylocal/third_party/tomlplusplus`, and the installed adapter exposes that include root without requiring a separately installed `toml++` package. The installed-package consumer is exercised in both modes so optional-component packaging is tested as well as source-tree compilation.

TextUI

`EasyLocal::TUI` is an optional FTXUI 7.x adapter. It is installed only when `EASYLOCAL_ENABLE_TUI=ON` and is discovered explicitly with:

```
find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core TUI)
target_link_libraries(my_tester PRIVATE EasyLocal::TUI)
```

The installed TUI target is exported separately from Core. When FTXUI comes from the system, the component retains a normal `find_dependency(ftxui 7)` requirement. When it comes from `FetchContent`,

FTXUI's installed package is used from the EasyLocal installation prefix. In either case, a Core-only consumer loads neither FTXUI nor `EasyLocal::TUI`.

REST

`EasyLocal::REST` is an optional Crow 1.3.x adapter. It is installed only when `EASYLOCAL_ENABLE_REST=ON` and is discovered explicitly with:

```
find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core REST)
target_link_libraries(my_server PRIVATE EasyLocal::REST)
```

The adapter produces a generic Crow Blueprint for a configured EasyLocal app; it does not own the Crow server. Crow request threads parse/route/enqueue work, while CPU-bound search runs execute on a separate bounded adapter-owned pool. Every run materializes fresh mutable EasyLocal runtime state and may share only the immutable Input. This is the same isolation rule used by asynchronous TextUI runs and avoids making Core search objects internally synchronized.

When Crow is provided by the system, the installed component keeps the normal Crow package dependency. With the explicit `FetchContent` path, the pinned Crow package and standalone Asio headers are installed with the EasyLocal REST component so the result remains relocatable. A Core-only consumer loads neither Crow/Asio nor `EasyLocal::REST`.

TLS, authentication/authorization, rate limiting, reverse-proxy policy, and Internet-edge hardening are deliberately outside the adapter and belong to the deployment infrastructure. See [rest.md](#) for the routes, codec contract, lifetime rules, and concurrency model.

API stability

EasyLocal follows [Semantic Versioning](#). This page states what the version number promises in the 4.x series.

Pre-releases

The versions 4.0.0-alpha.N and 4.0.0-beta.N promise nothing: any part of the library, header paths included, may change between one pre-release and the next, and every incompatible change is noted in the [changelog](#). The levels below apply from 4.0.0.

The installed CMake package reports the numeric version in `EasyLocal_VERSION` (`find_package(EasyLocal 4.0)` accepts a pre-release of 4.0.0) and the pre-release in `EasyLocal_VERSION_PRERELEASE` (alpha.1, empty for a release).

Levels

Level	What	Promise
Stable	<p>The headers under <code>include/easylocal/</code> outside detail, except what the Experimental level lists: the problem model (helpers/: concepts, bases, neighborhood union, recipes), <code>cost/</code>, <code>Runner</code>, <code>BoundRunner</code>, <code>make_runner / make_solver</code>, the composition, both its <code>with_*</code> members and the pipe (<code>\ </code>, <code>&</code>) that is their shortcut, the built-in runners and their <code>parameters_type</code>, <code>search_result</code>, <code>termination_reason</code>, <code>search_result_for</code>, <code>run_control</code> and the run options (<code>with</code>, <code>stop_at</code>, <code>timeout</code>, <code>max_evaluations</code>), the tracer protocol (<code>trace::null_tracer</code>, <code>observes</code>, <code>trace::emit</code>, a tracer passed with <code>with</code>), the solvers, their parameters and their builders (<code>seed</code>, <code>initialization</code>), <code>app</code>, <code>App</code> and <code>BoundApp</code> (their members keyed by name), <code>check</code>, <code>Session</code>, <code>cli::run</code> and its switches, the I/O hooks of <code>app/io.hpp</code> (<code>load_input</code>, <code>load_solution</code>, <code>save_solution</code>, <code>describe</code>), <code>config/</code> (<code>parameters</code>, command line, configuration files), <code>testing/</code>. The CMake targets <code>EasyLocal::*</code> and the <code>EASYLOCAL_*</code> options.</p>	No incompatible change before 5.0.
Stable extension API	<p><code>search_run</code>, for writing runners: the members documented in chapter 8 and in the runners reference.</p>	Members may be added in minor releases, none removed or changed.
Experimental	<p>The optional adapters, <code>adapters/</code> (<code>TOML</code>, <code>TextUI</code>, <code>REST</code>): their headers, the <code>TextUI</code> options and layout, the <code>REST</code> routes and <code>JSON</code> envelopes. Tracing beyond the tracer protocol, <code>trace/</code>: the events and their fields, the recorders, the <code>JSONL</code> and <code>ELTR</code> formats, <code>scripts/eltr.py</code> and its output. The export headers, <code>EASYLOCAL_DETAIL_*</code> to <code>trace (app2tuning.hpp)</code> macros, anything not documented.</p>	May change in a minor release, always noted in the changelog .
Internal	<p><code>detail/namespaces.*detail/</code> and <code>headers</code>, <code>EASYLOCAL_DETAIL_*</code> to <code>trace (app2tuning.hpp)</code> macros, anything not documented.</p> <p><code>TuningParameters</code>, <code>tuning_range</code>, and so <code>cli::options::tuning</code>) and the</p>	None.

A Stable declaration that uses an Experimental one is as stable as the Experimental one: `cli::options` and `cli::CommandLineParameters` are Stable, but the field `cli::options::tuning`, a list of `tuning_range`, is Experimental, like the export to `irace`. The levels are reviewed once more before 4.0.0, which may still move parts from one to another.

Adding members to stable types, overloads, concepts' optional capabilities, enumerators to `termination_reason` and new runners, solvers or cost models are compatible changes. Code that switches exhaustively over `termination_reason` should keep a default branch.

Reproducibility

A seed reproduces a run or a solve on the same platform and standard library, under these conditions:

- **No time limit.** A run bounded by evaluations, iterations or a target is reproduced; one ended by a time limit (`timeout`, `--timeout`, the TextUI's *seconds*, REST's `timeout`) stops after however many evaluations the machine makes in that time.
- **The same commands, in the same order.** A session (`Session`, `cli::run`, the TextUI, each REST run) seeds its RNG once. A random solution draws from it, and each run gets a generator of its own, seeded with one draw of it: the same seed and the same sequence of commands (random solutions, runs, random moves) give the same runs in every frontend, and the second run of a session differs from the first.
- **A solver's stream.** A solver (`LocalSearch`, `MultiStart`, a pipeline) seeds its RNG once, at construction or with `.seed(...)`; each `solve()` continues the stream, so its seed reproduces the sequence of solves, not each solve on its own.
- **REST seeds.** A REST run is reproduced by its seed: the request's seed, or `blueprint_options::seed` plus the run id, which depends on the runs the service started before it.
- **The same standard library.** Results are not reproducible across standard libraries (`libstdc++`, `libc++`, Microsoft's): `std` distributions such as `std::uniform_int_distribution`, `std::shuffle` and `std::hash` (which the helpers of `utils/hash.hpp` combine) are implementation-defined.

Supported toolchains

The compilers and platforms of the CI matrix are supported (see the [changelog](#) and the README). AppleClang needs Xcode 26 or later and a macOS deployment target of 26.0 or later, where its `libc++` has the floating-point `std::from_chars`; the configuration of the project, and `find_package` of the installed one, stop with a message otherwise. Dropping one happens in a minor release and is noted in the changelog.

Roadmap

Planned evolutions of EasyLocal: directions the design has already been prepared for, not yet scheduled. Nothing here is a promise of the [API stability](#) policy; an item becomes part of the API only when it is released and listed in the changelog.

Asynchronous runs on a Session

Why. A Session runs a registered runner on the calling thread: `session.run("sa", with(control))` returns when the runner has finished. That suits a program that only waits for the result, and the REST service, where every run has a Session of its own on a worker thread. It does not suit an interactive frontend, which keeps reading the session (the current solution, the selected move, the pages it shows) while a runner works, and must stay responsive. The TextUI does this today on its own: it copies the app, the Input and the current solution, runs the runner on a worker thread with `app.run("name", ...)`, and puts the result back when it arrives.

What. The Session would offer the same in two steps, for every frontend:

- `session.start("sa")` prepares an independent run, with copies of what it needs, a `run_control` for cancellation and progress, and its own RNG drawn from the session's, to be executed on another thread;
- `session.accept(result)` makes the run's result the current solution, unless the session has moved on since the run started.

When. When a second interactive frontend needs it, for example a GUI, a web frontend, or REST resources for interactive sessions (selecting and applying moves over HTTP). The TextUI would then move to the same mechanism, and the design would start from the needs of two real users.

The Pareto front in the frontends

Why. A run with a `cost::pareto` cost returns the non-dominated solutions it reached: `app.run("name", ...)` carries them in `named_run_result.front`, the Session keeps them (`last_run_front()`), `cli::run` prints them and REST returns them with the solution. The TextUI still shows one solution, the current one, so a multi-objective app shows a single trade-off there.

What. A TextUI page that lists the front of the last run, each point with its cost, and makes one of its solutions the current one.

When. Next, as a change of its own: the other frontends already show the front.

A failed run in the trace and in the result

Why. An exception thrown in the middle of a run, by a hook of the problem or by the tracer, ends the run without a trace record and leaves `solve()` with nothing: MultiStart loses its best start so far, a pipeline its earlier stages and their reports. A trace shows only that the run did not finish (`eltr.py` marks it unfinished), not why, and a long solve that fails at its last start returns nothing of the work done.

What.

- a `run_aborted` core event, emitted by `search_run` when an exception leaves the run, with the counters and the cost reached, before the exception goes on;
- a `solve_failed` exception thrown by the solvers, which carries the best result so far and the stage reports, with the original exception nested.

When. Not scheduled: when a long-running use, such as a REST service or a tuning campaign, needs the partial results.

Python and Julia bindings

Why. The problem components are C++: the SolutionManager, the neighborhood explorers, the cost components and their deltas, compiled for speed. Experiments around them, such as choosing runners and parameters, running many instances and seeds, and analysing the results, are often more convenient in a notebook or a script.

What. An optional adapter, like the TextUI and the REST service, that exposes a compiled app to Python or Julia: list its runners, set their parameters through their parameter set, create a Session on an Input, run runners by name, and read solutions, costs and traces. The C++ problem is written once; the scripting side never reimplements it. It would build on the same public surface the other adapters use (app, Session, app.run, the parameter sets), with a binding library for Python, such as nanobind, and a C interface for Julia.

Each frontend is a template on the app's type today: the TextUI and REST are compiled with the problem. A binding needs a session whose type does not depend on the app's, a runtime-erased Session (runner names, parameter paths, solutions and costs as text or JSON, runs by name), which the TextUI and REST could then share instead of each instantiating its own; after 4.0.

When. Not scheduled. Like the other adapters, it would live outside `EasyLocal::Core` and could become a separate project once the API is stable.

Shifting penalty

Why. When the hard constraints are folded into the cost with a fixed weight, the right weight depends on the instance and on the phase of the search: too low and the search stays infeasible, too high and it cannot cross infeasible regions to reach better feasible ones. EasyLocal 3 had a `ShiftingPenaltyRunner` that wrapped any move runner and adapted the weights of the constraints while the search ran.

What. Weights that change during a run, raised after a number of iterations in which a constraint stays violated and lowered after a number of iterations in which it is satisfied, within a range and with a random perturbation. In EasyLocal 4 the weights live in the cost expressions (`weighted`, `hard_soft`), so the design starts there: dynamic weights in the expression tree, which a runner, or a wrapper around any runner, adjusts, with the deltas staying consistent with the current weights.

When. Not scheduled. It depends on how the cost expressions settle.

A name for every service

Why. A cost component may have a `name()`, which the reports and the interactive tester show in place of its position, and a neighborhood explorer a static `name()`, which the tester puts on its Move page. A SolutionManager, a delta cost component and a runner have none, so a frontend that lists them names them by their position or by their registration. The two `name()` that exist are also read for two different purposes: the label of a component, and the path a configurable class is configured under (`cost.<name>.*`), where it is required rather than optional.

What. One rule for the whole model: which services may be named, whether a name is a label, a path, or both, and what names a service that has none. A mixin, `named<"2-opt">` or a `name_base` beside `input_base` and `parameters_base`, would spare the boilerplate and make the capability visible in the class's declaration; it has to stay optional, since a class that nobody lists needs no name, and it must not collide with the static `name()` the configurable components already require.

When. Not scheduled. It is worth settling before the frontends grow: the launcher, a GUI and the REST resources all list services.

A tester on an app without a neighborhood

Why. An app is built one component at a time, and the tools open on one without runners: the tester shows the Input, the solutions and the moves, and has no runner to run on its Run page. A neighborhood is still required, so the first step of a model, a SolutionManager with its cost components, cannot be opened in the tester, which is where it would help most: the cost of a solution read from a file, the value and the description of each component, the initial and the random solutions.

What. A stub service for an app whose neighborhood recipe is still unconfigured: BoundApp builds a neighborhood explorer that neither enumerates nor draws moves, so every capability of the Session about moves is already false and its move commands drop out of the overload set. The Session says so with a flag of its own, and the tester leaves out its Move page rather than showing an empty one; App::configuration() has no neighborhood.* block, and check() reports the neighborhood as missing instead of failing on it.

When. Not scheduled.

A launcher across SolutionManagers

Why. The TextUI's launcher opens several apps on the same problem and passes the Input and the current solution from one to the next, so the apps must have the same SolutionManager. Some problems are explored with different ones: two representations of a solution (a permutation and an assignment, for example), or a relaxed model next to the full one. Moving a solution between them today means saving it from one app and loading it in the other, when the two formats agree.

What. A launcher whose apps may have different SolutionManagers, with conversions between their solutions declared by the problem, for example a hook convert(const Input&, const SolutionA&) -> SolutionB found by ADL. The launcher converts the current solution when another app opens, and checks at compile time that every app it may switch to can be reached by a conversion; apps with the same SolutionManager keep passing it unchanged.

When. Not scheduled.

Adaptive neighborhood selection

Why. In a multi-neighborhood search the share of moves drawn from each neighborhood is fixed by the user, while the neighborhoods that pay off change with the instance and along the run. EasyLocal 3 had a SimulatedAnnealingWithLearning runner that learned these shares during the search.

What. A selection policy for the union of neighborhoods that updates the probability of each one from what its moves achieve (improving, sideways and accepted moves, the improvement obtained, the time spent evaluating), with a learning rate and a lower bound on each probability. It belongs to the multi-neighborhood composition rather than to Simulated Annealing, so that every runner that draws random moves could use it. A simpler fixed policy fits the same place: an optional size of each child's neighborhood for a solution would let the union draw a child in proportion to it, and so draw uniformly among all the moves when the children do.

When. Not scheduled.

The fixed cost of a neighborhood union

Why. A union dispatches on the variant that is its move type several times for each move it evaluates: the iterator of the enumeration holds a variant of the children's iterators and visits it to advance and to build the tagged move, and is_valid(), make_move() and each delta visit the move again. The bindings of the union are built on every call of delta_bindings(), which the evaluation makes once per component of every move, where a neighborhood with a delta layer returns the tuple it holds. None of this showed while a union without a delta everywhere evaluated its components in full; now that the deltas are used

per child, it is what is left. The benchmark measures it on the TSP (results `main-d2ceb00`): with every delta attached, a steepest descent costs 5.1 nanoseconds per evaluation on the 2-opt neighborhood alone and 20.5 on the union of 2-opt and swap, a first improvement 4.3 against 17.5. The deltas there are $O(1)$, so what the union adds is the whole difference; with a delta that scans the solution it would disappear in the noise.

`delta_bindings()` is built once, when the union is constructed, and returned by reference, since 4.0.0-alpha.3, and the infrastructure benchmarks now measure the union against the same moves with its machinery written by hand: `tsp-union-medium` enumerates them child after child, `tsp-union` also evaluates them, with one delta that visits the move in place of the union's binding per child. The union's iterator costs about 0.7 nanoseconds per move to enumerate, and the two are within 5% of each other under a runner. Neither the iterator nor the bindings, then, are what a union costs: what is left against a single neighborhood is the variant move type itself, which every `is_valid()`, `make_move()` and `delta` visits, and the second child's delta.

What. Evaluating the moves of a child with that child's own move type: the union's enumeration would hand a runner the child it is in, and the variant would be needed only to carry the move the runner keeps, so the visits of the evaluation disappear. It changes how a runner walks a neighborhood, which is the same question as [where a first improvement starts its scan](#).

When. Not scheduled, and the measurements bound what it can give: the union's own machinery is already free, so what it would remove is the cost of the variant alone.

Where a first improvement starts its scan

Why. `FirstImprovement` applies the first improving move of the enumeration and starts the next scan at the first move again, so the moves the previous iterations rejected are scanned once more before a new one is reached. `EasyLocal 3's FirstDescent` instead starts each scan at a move drawn at random and wraps around, which is why the two walk different trajectories. On a large neighborhood that is many more evaluations for the same number of iterations: on a 300-city TSP, with the 2-opt neighborhood of about 44500 moves, `EasyLocal 3` evaluates some 1100 moves per iteration and `EasyLocal 4` some 13500. It also decides *which* moves are evaluated: a union enumerates its children one after the other, so a scan that always restarts evaluates almost only moves of the first child, and a delta attached to that child alone looks nearly free. The benchmark measures the two walks apart, `fd` and `rfd`, rather than comparing them.

What. A scanning strategy for the algorithms that apply the first improving move: from the first move (what it does today), from a move drawn at random, or resuming where the previous scan stopped. A scan that does not begin at the first move needs a neighborhood it can enter at a given move, which the cursor protocol (`first_move`, `next_move`) gives and a `moves()` generator does not: the strategy is available where the explorer has a cursor, and a union needs a cursor of its own, which it has not. The same question is behind the candidate strategies of Tabu Search below.

When. Not scheduled. It is worth settling before the benchmarks are read as a comparison of the two frameworks' first descent.

Candidate strategies of Tabu Search

Why. The four tabu searches (`TabuSearch`, `FirstImprovementTabuSearch`, `AspirationPlusTabuSearch`, `EliteCandidateTabuSearch`) are one algorithm with four ways of scanning the neighborhood; they already share one parameter block, `TabuSearchParameters<List, Candidates>`, but each strategy is still a runner of its own.

What. One `TabuSearch<List, Aspiration, Candidates>` with a candidate strategy policy, `candidates:: {Full, FirstImprovement, AspirationPlus, EliteList}`, whose parameters are the group candidates they already have, and aliases for the names of `EasyLocal 3`.

When. Not scheduled.

Cooperative runners

Why. A pipeline runs its stages one after the other, each from the solution of the previous one. Some searches combine runners differently: several runners at the same time, on different neighborhoods or with different parameters, that exchange their best solutions (a portfolio, an island model), or that alternate on the same solution under a common control, without a fixed order.

What. A solver of cooperative runners: the runners, a policy that says when they run (in parallel threads, or interleaved) and what they share (the best solution found so far, at given intervals or when one improves it), a common stopping criterion and budget, and the effort and best result of each runner in the result. Its determinism with a seed when the runners run in parallel is part of the design.

When. Not scheduled.

Parallel search

Why. A run uses one thread. EasyLocal 3 explored a neighborhood in parallel (ParallelNeighborhoodExplorer, over TBB) and kicked in parallel, and the machines a search runs on have many cores, idle for the whole run. The only parallelism today is outside a run: the REST service runs independent requests on a pool of workers.

What. Three levels, each useful on its own:

- **Inside a neighborhood.** A scan that evaluates every move (Best Improvement, Tabu Search) splits the moves over threads and reduces the results deterministically, so the move chosen is the one the sequential scan would choose. The delta cost components are const members over an immutable Input, so they can already be called concurrently unless a component caches something; the contract would have to say so.
- **Across runs.** Runners that search the same Input at once, sharing their best solutions or nothing at all (a portfolio): the solver level of [Cooperative runners](#).
- **Across solves.** Instances and seeds run in parallel, which belongs to the [experiment configurations](#) rather than to the library.

The design questions are shared: who owns the threads, how a budget, a progress observer and a cancellation cross them, what a trace of a parallel run records, and what a seed guarantees when the work is split.

When. Not scheduled. The first level is the one an EasyLocal 3 program that used the parallel explorer needs.

Kicks, Iterated Local Search and Variable Neighborhood Descent

Why. EasyLocal 3 had kickers (a perturbation of several moves applied at once to leave a local optimum), the TokenRingSearch and VariableNeighborhoodDescent solvers. A pipeline runs its stages once, in a fixed order: it cannot go back to an earlier stage when a later one improves, nor perturb its solution between rounds, which Iterated Local Search and VND need.

What. A kick, a sequence of random (or enumerated) moves of a neighborhood applied together, as a stage of its own; pipeline rounds (run the stages again, from the last solution, while a round improves the best cost or until a budget), and a restart rule (back to the first stage on an improvement), which together give Iterated Local Search (descent, kick, acceptance) and VND (descents on neighborhoods in turn, back to the first after an improvement). The rounds share the solve's budget and progress, as the stages do.

When. Not scheduled.

A modelling layer

Why. Writing a delta cost component by hand is where the bugs of a search are: the checks of the library exist to catch them, and an incremental evaluation has to be rewritten whenever the neighborhood changes. EasyLocal 3 carried a tentative modelling layer (AutoState, with Tommaso Urli) that derived the cost and its changes from a declarative model; it never became part of the framework people used, but the need it addressed did not go away. Constraint-Based Local Search answers it with a declarative model whose network of invariants derives the cost and its change under a move: [Oscar.cbls](#) as a library, [Hexaly](#) (formerly LocalSolver) as a black-box solver that also owns the search.

What. Not in this library. The core stays a framework where the search is written by its user, and the modelling layer grows outside it, in `easylocal-model` (namespace `easylocal::model`), which consumes the released headers and nothing else: whether that is enough is also the measure of how extensible this API is from the outside. The design it starts from:

- the solution stays the decision state alone, copied and hashed as it is today, and the network of invariants lives in an evaluator owned by the SolutionManager, which follows the current solution (`sync`), answers `probe(move)` without committing and `commit(move)` when the explorer applies one. A solution that carried its network would pay for it at every copy, which is what sank AutoState;
- the graph is built at run time from the instance data, and is made of typed kernels (sum, weighted sum, element, count, min and max, capacity, all-different violations, the successor of a sequence): the dispatch is per node, over a whole vector, not per element;
- a move is differentiated on a trail by default (apply the variable changes, propagate incrementally, read the objective, roll back), with a closed-form fast path for the kernels that have one, which are the linear ones on the hot path;
- a constraint exposes its violation and the violation imputed to each variable, which is what guided neighborhoods (the most violated variable, then the value that reduces it most) are built on, and what makes an explorer that nobody wrote for the problem possible;
- what a model produces are the pieces of this library: each constraint and each objective is a leaf of a cost expression, with a SolutionManager, the neighborhoods and the deltas, so hard and soft costs, hierarchical costs and Pareto costs keep working. A declared component and a hand-written one then compose in the same cost expression: the hot component keeps its closed delta, the rest is declared. That mixing is what justifies doing this here rather than using Hexaly.

A catalog of standard representations (a permutation, an assignment, a selection, a sequence per resource) with their moves, their explorers and the structures they keep consistent complements the model: the moves say what changes, the model says what it costs.

The price is a factor of 3 to 10 per move against a hand-written delta. The layer is there to write a problem in an afternoon instead of a week, and to serve as a reference the hand-written components are checked against (the delta checks of testing compare a delta with a full evaluation already); it is not there to win benchmarks.

When. Not scheduled. A vertical spike on the assignment problem comes first, with a few kernels and no fast path, to answer two questions: whether a mutable evaluator behind the `const delta_evaluate` works without touching the core, and what it costs, measured against `examples/assignment`.

Parameter tuning

Why. The parameters of a search are usually tuned with an automatic configurator: [irace](#) (iterated racing), [SMAC](#) (Bayesian optimization with random forests) or [Optuna](#) (Bayesian optimization in Python, define-by-run). `irace` is supported: every parameter declares its domain (`check(app)` fails on one without), conditions and requirements between parameters (`only_if`, `config::require`) are exported as `irace` conditions and `[forbidden]` expressions, and `cli::run --tuning.irace` writes the scenario. What is left:

What. SMAC and Optuna exporters from the same domains and conditions: the configuration space (ConfigSpace) and a target function for SMAC; the search space as a Python function for Optuna, which with the Python bindings (above) could also run the app in-process.

When. Not scheduled.

Experiment configurations

Why. Tuning and benchmarking produce configurations (the tuned values, the fixed ones, the instances and seeds) that must be kept with the version of the code that produced them, and found again for a paper or a later comparison. The tool that versions them is outside EasyLocal, but EasyLocal programs must make it easy, with little boilerplate.

What. What a program could give to such a workflow:

- its effective configuration, every parameter with its value, in the `path = value` format that `--config` reads back;
- its identity in its output: the EasyLocal version and the commit of the program's code, recorded at build time;
- a machine-readable result of a run: instance, seed, configuration (or its hash), cost, effort and time;
- irace's elite configurations as configuration files.

When. Not scheduled: to be designed with the workflow it serves.

Trace micro-benchmarks

Why. The tracing overhead benchmark of [easylocal-benchmarks](#) measures a First Improvement workload, whose events are the core ones; it also records the visited solutions, with a hash per committed move, and leaves them out at compile time (`trace::without`). Tabu search's aspirations, escapes and tenure changes, and the cost of each event type on its own, are not measured.

What. Micro-benchmarks of the encoding of each event type, the solution hash included, and a Tabu Search workload with each tabu list, under the same recorders as the existing benchmark (null, in-memory, buffered and asynchronous ELTR), reporting the cost per event and per iteration.

When. Not scheduled.

Delta ablation of a neighborhood explorer

Why. A problem's delta cost components are where most of its speed comes from, but a delta is not always worth it: the EasyLocal 3 versus EasyLocal 4 benchmarks measure each problem with every delta, with some and with none, and on the assignment problem the run without deltas (which applies each move to a copy and recomputes the cost) beats the one with all of them, whose deltas cost $O(\text{jobs})$ each. Which deltas pay off depends on the problem, the instance and the neighborhood, and today it is found by hand.

What. A benchmarking tool for a problem's own neighborhood explorers: it evaluates the moves of sampled solutions with every subset of the delta cost components, from all of them to none, one component at a time (so each delta is measured against the full evaluation it replaces), checks that every subset gives the same costs, and reports the time per move of each configuration and of each delta against none. It would run on the problem's instances, next to the contract checks of `<easylocal/testing.hpp>`, and tell which deltas to keep.

When. Not scheduled.

Parameters described by reflection (C++26)

Why. A parameter block describes itself by hand: `parameter_schema()` lists each field with its name, a pointer to the member and a description, next to the field itself. Everything else is generic (the command line, configuration files and TOML, the interactive tester's parameter windows), but the list repeats the struct, and a field added without its schema line is silently not configurable.

What. With C++26 static reflection, the schema would be derived from the struct: one parameter per data member, named after it, with the description given as an annotation on the member. A hand-written `parameter_schema()` would remain possible, and would take precedence, for names or groupings that differ from the members. The schema is then the only part of a runner's configuration still written by hand: a runner already holds its algorithm's parameters and builds the algorithm from them.

When. When the compilers EasyLocal supports provide reflection. In October 2026, GCC 16 implements it behind `-freflection`; GCC 15, Clang 22 and 23 (neither the `^^` operator nor `libc++'s <meta>`) and Apple Clang do not. Until all supported compilers have it, reflection could only be an optional path next to the hand-written schema.