

EasyLocal — Coming from EasyLocal 3

A C++23 framework for local search and metaheuristics

Contents

Coming from EasyLocal 3	3
Before you start	3
Porting the problem	4
Composing and running	14
Checking the port	21
Parameters, configuration and tuning	23
What EasyLocal 4 adds	24
A porting checklist	25
Reference: the two frameworks side by side	25
Next steps	29

Coming from EasyLocal 3

This guide walks through porting a TSP program with 2-opt moves from EasyLocal 3 to EasyLocal 4. It follows the program from the [problem model](#) through [composition and execution](#) to [checking the port](#). You can then look up [parameters and tuning](#) or use the [comparison tables](#) to find the counterparts of familiar concepts and runners.

Much of your problem-specific code can stay: the solution representation, move logic and cost formulas. The main benefit is in the code around it. Recipes replace manual object registration, one app supplies the command line and interactive tester, and checks that once lived in tester menus can run automatically in your test suite.

The implementations also reduce the work spent around each move. Algorithms, cost components and policies are composed at compile time, allowing direct, inlined calls in place of virtual dispatch. When every component has a delta, evaluating a proposal needs no candidate solution copy; unused tracing is removed at compile time. These savings matter especially when moves are cheap.

The [published benchmarks](#) give a concrete comparison: on 7 October 2026, EL4 v4.0.0-alpha.2 versus EL3 v3.4.1 achieved a **1.82× geometric mean speed-up in time per evaluation** across the tested problems, algorithms and delta modes. On the 300-city TSP with full deltas, the factors were 5.73× for steepest descent and 2.76× for Simulated Annealing. These are measurements on the same Ubuntu/GCC 16 runner, not a guarantee for every problem: two Assignment descent cases were slightly slower. Compare time per evaluation as well as final cost; different search trajectories can require different numbers of evaluations.

The EasyLocal 3 examples are abridged from the TSP used in the [benchmarks](#), based on [easylocal-legacy v3.4.1](#). The EasyLocal 4 examples come from the [tutorial](#) and are compiled and tested at every build.

Before you start

- EasyLocal 4 needs a C++23 compiler (see the [quick start](#)). It parses the command line itself and no longer depends on Boost.
- Use `#include <easylocal/easylocal.hpp>` instead of `easylocal.hh`, and namespace `easylocal` instead of `EasyLocal::Core`. The name `EasyLocal::Core` now identifies the CMake target provided by `find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core)`.
- The examples use short namespace aliases in place of using `namespace EasyLocal::Core;`

```
cpp title="EasyLocal 4" namespace el = easylocal;           // el::app, el::component, ...
namespace runners = easylocal::runners; // runners::FirstImprovement, ...
```

Thus `el::app` means `easylocal::app`. Keep these aliases in a function or source file to avoid introducing them into files that include your headers (see the [tutorial conventions](#)).

- Start with the SolutionManager, the cost components and one explorer. Without deltas, the framework evaluates each move on a copy of the solution. Once this version works, add deltas one at a time and [check each against full evaluation](#), as you would with EasyLocal 3's MoveTester.
- Before you start, look up your runners and solvers in the [comparison tables](#). Most have a counterpart; some have a different name, and a few are not available yet.

Two C++ features used below may also be unfamiliar:

inline lets a free function defined in a header appear in several translation units without causing a multiple-definition error. Use it for header-defined hooks such as `read_solution` and `describe` (step 4). If you define them in a `.cpp` file, it is unnecessary. The compiler decides whether to inline calls for speed independently of this keyword.

`[[nodiscard]]` asks the compiler to warn when a caller ignores a result. The library uses it for functions such as `check(app, input)`, a parameter set's `apply`, and `random_move(explorer, solution, rng)`. This is why the examples inspect the check report with `if (!report) return 1;`. In your own code, use it where dropping a result would be a bug; the user hooks below do not need it.

Building without CMake

If your EasyLocal 3 program is built by a makefile, it can stay one: [Building without CMake](#) gives the whole workflow, from the include path to the libraries of the optional components. Two lines of the old makefile change: the C++ standard becomes `-std=c++23`, and `-lboost_program_options` goes, since EasyLocal 4 parses the command line itself. Core still needs no library to link and no `-D` definition.

Porting the problem

1. The Input

An EasyLocal 3 Input was a class that read its own file in the constructor:

EasyLocal 3

```
class Input
{
public:
    explicit Input(const std::string& path)
    {
        std::ifstream is(path);
        is >> city_count;
        distances.resize(city_count * city_count);
        for (auto& d : distances)
            is >> d;
    }
    double Distance(std::size_t from, std::size_t to) const;

    std::size_t city_count = 0;
    std::vector<double> distances;
};
```

In EasyLocal 4, the Input is a plain value. A separate function reads it from a stream (step 4), so you can also construct it in a test or receive it over HTTP:

EasyLocal 4

```
// Input: the instance, immutable during the search.
struct Tsp
{
    // distance[a][b] is the distance between cities a and b (symmetric).
    std::vector<std::vector<double>>> distance;

    std::size_t cities() const
    {
        return distance.size();
    }
};
```

You can keep accessors such as `Distance(from, to)`: the library never calls Input members. Data precomputed from the instance can also stay in the Input, or be computed in the constructor of the service that uses it.

2. The Solution

An EasyLocal 3 State was constructed from the Input and often kept a reference to it:

EasyLocal 3

```
class Tour
{
public:
    explicit Tour(const Input& in) : tour(in.city_count) {}
    std::vector<std::size_t> tour;
};
```

In EasyLocal 4, the Solution holds the same data and supports copying, moving and assignment:

EasyLocal 4

```
// Solution: order[k] is the k-th city visited; after the last city the tour
// returns to order[0].
struct Tour
{
    std::vector<std::size_t> order;
};
```

- **Remove the Input reference.** The SolutionManager, explorers, cost components and deltas already have access to the Input, usually through `input()`. The I/O hooks receive it as a parameter. A reference member prevents implicit assignment; a pointer requires every copy to refer to the same Input and that Input to outlive them all.
- **Let the SolutionManager size the containers** when it builds a solution (step 5). Keep a constructor from the Input only while you still use `operator>>`, which reads into `Solution{input}` (step 4).
- **Keep cached data in the Solution** if it speeds up evaluation. Update redundant matrices or counters in `make_move`. Use the SolutionManager's `hash` and `equal` to exclude them from solution identity when a tabu list or trace compares solutions (see the [SolutionManager reference](#)).

3. The Move

An EasyLocal 3 Move typically had a constructor with default arguments, so runners could create a move before filling it. It also supplied the comparison and stream operators required by the framework:

EasyLocal 3

```
class TwoOpt
{
public:
    TwoOpt(std::size_t first_edge = 0, std::size_t second_edge = 0)
        : first_edge(first_edge), second_edge(second_edge) {}
    std::size_t first_edge, second_edge;
};

bool operator==(const TwoOpt& a, const TwoOpt& b);
bool operator!=(const TwoOpt& a, const TwoOpt& b);
bool operator<(const TwoOpt& a, const TwoOpt& b);
std::ostream& operator<<(std::ostream& os, const TwoOpt& mv);

// in the explorer
mv = TwoOpt(first_edge, second_edge);
```

In EasyLocal 4, a Move can be a simple struct with public fields:

EasyLocal 4

```
// Move: reverse the part of the tour between positions i + 1 and j.
struct TwoOpt
{
    std::size_t i;
    std::size_t j;
};
```

The explorer writes `move.i` and `move.j`; the delta cost component reads them. The explorer's `is_valid` checks whether these positions form a legal 2-opt move. Since the fields can be set freely, the examples use `struct`. A class with public fields works too; the library never calls a member on a move.

Without a user-declared constructor, `TwoOpt` is an aggregate. You can create it as `TwoOpt{i, j}`, `TwoOpt{.i = i, .j = j}`, or `TwoOpt{}` with both fields zero. The last form lets the cursor in step 7 create a move to fill in.

The comparison and stream operators are optional. Add `operator==` for the [neighborhood checks](#); a defaulted `bool operator==(const TwoOpt&) const = default;` keeps the type an aggregate. For display in the tester and reports, provide `describe(move)` (step 4) or `operator<<`. The framework does not call `< or !=`.

A tabu list needs `std::hash` and `==` for the move, unless the explorer provides a separate `tabu_attribute(move)`.

4. Reading, writing and describing

EasyLocal 3 read the Input in its constructor and used `operator>>` and `operator<<` for the State. **The stream operators still work** as fallbacks, so you can keep them during the port and replace them later.

EasyLocal 4 — what an EasyLocal 3 program already has

```
// Read into a default-constructed Tsp, in place of the file constructor.
std::istream& operator>>(std::istream& in, Tsp& tsp);

// Read into Tour{tsp}: on this path the Solution keeps the constructor from
// the Input that step 2 could otherwise drop.
std::istream& operator>>(std::istream& in, Tour& tour);

// Written as it was, by the tester, cli::run and the REST service.
std::ostream& operator<<(std::ostream& out, const Tour& tour);
```

Dedicated hooks can receive the Input as well as the stream. This is useful when reading a solution: the TSP reader needs the number of cities, which the Solution no longer stores.

Define these free functions alongside your types. The library finds them through argument-dependent lookup and prefers them to stream operators:

- `read_input(std::type_identity<Input>, in)` returns an Input. The type tag selects which Input to read.
- `read_solution(const Input&, std::istream&)` returns a Solution.
- `write_solution(const Input&, const Solution&, std::ostream&)` writes it.
- `describe(value)` provides display text for an Input, Solution or Move.

The tutorial's solution hooks replace the two State operators above:

EasyLocal 4

```
inline Tour read_solution(const Tsp& tsp, std::istream& in)
{
```

```

Tour tour{std::vector<std::size_t>(tsp.cities())};
for (auto& city : tour.order)
    if (!(in >> city))
        throw std::runtime_error{"invalid tour"};
return tour;
}

inline void write_solution(const Tsp&, const Tour& tour, std::ostream& out)
{
    for (const auto city : tour.order)
        out << city << ' ';
    out << '\n';
}

```

For reading and writing, the library tries a member, a free function, then a stream operator. It uses the first available form, so a hook can override an operator without removing it:

EasyLocal 3	EasyLocal 4, in the order the library tries them
Input(const std::string& path) reading the file	a static Input::read(in), read_input(std::type_identity<Input>, in), or operator>> into a default-constructed Input
operator>>(is, State&)	a static Solution::read(input, in), read_solution(input, in), or operator>> into Solution{input}
operator<<(os, const State&)	a member solution.write(input, out), write_solution(input, solution, out), or operator<<
PrintViolations, “pretty print output”	describe(input), describe(solution), describe(move), and the describe(solution) of a cost component (step 6)

The file helpers (load_input, load_solution), interactive tester, cli::run and REST service all use these same hooks. Your main can leave file handling to them, and a change to your file format reaches every frontend through the same implementation (chapter 6).

5. The SolutionManager

EasyLocal 3

```

class TspSolutionManager : public SolutionManager<Input, Tour, CostStructure>
{
public:
    explicit TspSolutionManager(const Input& in)
        : SolutionManager<Input, Tour, CostStructure>(in, "TspSolutionManager") {}

    void RandomState(Tour& st) override
    {
        st.tour.resize(in.city_count);
        std::iota(st.tour.begin(), st.tour.end(), std::size_t{0});
        std::shuffle(st.tour.begin(), st.tour.end(), Random::GetGenerator());
    }

    bool CheckConsistency(const Tour& st) const override
    {

```

```

        return st.tour.size() == in.city_count;
    }
};

```

EasyLocal 4

```

class TourManager : public easylocal::solution_manager_base<Tsp, Tour>
{
public:
    // The constructor of the base: from the Input, which input() gives back.
    using solution_manager_base::solution_manager_base;

    // The cities in index order: 0, 1, ..., n - 1.
    Tour initial_solution() const
    {
        Tour tour{std::vector<std::size_t>(input().cities())};
        std::ranges::iota(tour.order, std::size_t{0});
        return tour;
    }

    // The same cities in a random order.
    template<std::uniform_random_bit_generator RNG>
    Tour random_solution(RNG& rng) const
    {
        auto tour = initial_solution();
        std::shuffle(tour.order.begin(), tour.order.end(), rng);
        return tour;
    }

    // A tour is valid when it is a permutation of 0, 1, ..., n - 1: it has
    // n positions and visits every city exactly once.
    bool is_valid(const Tour& tour) const
    {
        return std::ranges::is_permutation(
            tour.order,
            std::views::iota(std::size_t{0}, input().cities()));
    }
};

```

EasyLocal 3	EasyLocal 4
constructor (in, name)	inherited: using solution_manager_base::solution_manager_base;
in	input()
RandomState(Solution&) with Random::	random_solution(RNG&), which returns the solution
GreedyState(Solution&)	initial_solution(), which returns the solution
CheckConsistency	is_valid: the representation is well formed
CostFunctionComponents, AddCostComponent	the recipe (steps 6 and 10)
LowerBoundReached, OptimalStateReached	a target cost for the run: stop_at(cost), or --target
StateDistance	no counterpart
PrintState, DumpState	describe(solution), write_solution (step 4)

The random construction logic can stay the same. Take the generator as a parameter, replacing `Random::GetGenerator()` with `rng` and `Random::Uniform<T>(a, b)` with `std::uniform_int_distribution<T>{a, b}(rng)`.

6. The cost components

EasyLocal 3

```
class TourLength : public CostComponent<Input, Tour, double>
{
public:
    explicit TourLength(const Input& in)
        : CostComponent<Input, Tour, double>(in, 1.0, false, "TourLength") {}

    double ComputeCost(const Tour& st) const override
    {
        double total = 0.0;
        for (std::size_t position = 0; position < st.tour.size(); ++position)
            total += in.Distance(st.tour[position],
                                st.tour[(position + 1) % st.tour.size()]);
        return total;
    }

    void PrintViolations(const Tour&, std::ostream&) const override {}
};
```

EasyLocal 4

```
class TourLength : public easylocal::input_base<Tsp>
{
public:
    using input_base::input_base; // constructed from the Input, read by input()

    double evaluate(const Tour& tour) const
    {
        const auto n = tour.order.size();
        double length = 0.0;
        for (std::size_t k = 0; k < n; ++k)
        {
            const auto from = tour.order[k];
            const auto to =
                tour.order[(k + 1) % n]; // the last city goes back to the first
            length += input().distance[from][to];
        }
        return length;
    }
};
```

`ComputeCost` becomes `evaluate`. The optional `input_base<Tsp>` provides a constructor from the `Input` and the `input()` accessor, with no virtual functions. A component that needs no `Input` can stand on its own.

Weights and hard/soft roles now belong to the recipe's cost expression. Suppose your program has two hard components, `H1` and `H2`, and two soft components, `S1` and `S2`, with weights 1 and 5. Each is a class with an `evaluate` member, like `TourLength`. Combine them as follows:

EasyLocal 4

```

el::solution_manager<Manager>()
| el::cost::hard_soft(
    el::cost::sum(el::component<H1>(), el::component<H2>()),
    el::cost::sum(el::component<S1>(), el::component<S2>() * 5))

```

`cost::hard_soft` replaces `HARD_WEIGHT` with a pair compared **lexicographically**. The hard cost decides first; the soft cost breaks ties. No soft improvement can compensate for a hard violation. If weights within the hard cost helped guide your search, keep them in the hard sum.

This changes Simulated Annealing's behaviour. Its acceptance probability, $\exp(-\text{delta} / \text{temperature})$, needs a scalar delta. For `cost::hard_soft`, that delta is the soft-cost change when the hard cost is unchanged, minus infinity when it improves, and plus infinity when it worsens. The search therefore always accepts a hard improvement and never accepts a hard degradation. Once it reaches feasibility, it stays feasible. EasyLocal 3's weighted sum could instead accept a violation and let the search cross an infeasible region.

To preserve that behaviour, use a single weighted sum, such as `el::cost::sum(el::component<H1>() * 1000, ..., el::component<S1>())`, and check the final solution for feasibility. You can also define your own rule with `cost::apply` at the root of the expression ([Cost](#)).

For violation reports, replace `PrintViolations` with the optional `std::string describe(const Solution&) const`. It returns text for the tester, reports or HTTP responses to display. An optional `name()` supplies the component name previously passed to the constructor. Without these members, reports show the component's position and value ([chapter 12](#)).

The cost type follows from the return type of `evaluate`: an int component stays int. Use `cost::in_order` for more than two priority levels, or `cost::objectives` for several separate objectives ([chapter 3](#)).

7. The NeighborhoodExplorer

EasyLocal 3

```

class TwoOptNeighborhoodExplorer
: public NeighborhoodExplorer<Input, Tour, TwoOpt, CostStructure>
{
public:
    TwoOptNeighborhoodExplorer(const Input& in, TspSolutionManager& sm)
        : NeighborhoodExplorer<Input, Tour, TwoOpt, CostStructure>(in, sm, "2-opt") {}

    bool FeasibleMove(const Tour& st, const TwoOpt& mv) const override;

    void RandomMove(const Tour& st, TwoOpt& mv) const override
    {
        const auto n = st.tour.size();
        if (n < 4)
            throw EmptyNeighborhood();
        while (true)
        {
            auto first_edge = Random::Uniform<std::size_t>(0, n - 1);
            auto second_edge = Random::Uniform<std::size_t>(0, n - 1);
            // ... ordered, and drawn again while not a valid move
        }
    }

    void FirstMove(const Tour& st, TwoOpt& mv) const override; // throws EmptyNeighborhood
    bool NextMove(const Tour& st, TwoOpt& mv) const override;

```

```

    void MakeMove(Tour& st, const TwoOpt& mv) const override;
};

```

An explorer can enumerate moves with a cursor or a generator. The **cursor** is the closest match to EasyLocal 3's FirstMove/NextMove pair.

The caller holds a Move and passes it by reference. `first_move` fills in the first move; `next_move` advances to the next. Both return `false` when no move is available. This is why the cursor needs a default-constructible Move (step 3): the caller must create one before enumeration starts.

EasyLocal 4

```

class TwoOptExplorer : public easylocal::neighborhood_explorer_base<TourManager,
TwoOpt>
{
public:
    using neighborhood_explorer_base::neighborhood_explorer_base;

    static std::string_view name()
    {
        return "2-opt";
    }

    bool first_move(const Tour& tour, TwoOpt& move) const
    {
        move = TwoOpt{0, 1};
        return next_move(tour, move);
    }

    bool next_move(const Tour& tour, TwoOpt& move) const
    {
        const auto n = tour.order.size();
        do
        {
            if (++move.j == n)
            {
                ++move.i;
                move.j = move.i + 2;
            }
            if (move.j >= n)
                return false;
        }
        while (move.i == 0 && move.j + 1 == n);
        return true;
    }

    template<std::uniform_random_bit_generator RNG>
    std::optional<TwoOpt> random_move(const Tour& tour, RNG& rng) const
    {
        const auto n = tour.order.size();
        if (n < 4)
            return std::nullopt;
        std::uniform_int_distribution<std::size_t> pick{0, n - 1};
        while (true)
        {
            auto i = pick(rng);
            auto j = pick(rng);
            if (j < i)
                std::swap(i, j);

```

```

        if (i + 2 <= j && !(i == 0 && j + 1 == n))
            return TwoOpt{i, j};
    }

    bool is_valid(const Tour& tour, const TwoOpt& move) const
    {
        const auto n = tour.order.size();
        return move.i + 2 <= move.j && move.j < n && !(move.i == 0 && move.j + 1 == n);
    }

    void make_move(Tour& tour, const TwoOpt& move) const
    {
        std::ranges::reverse(std::span{tour.order}.subspan(move.i + 1, move.j -
move.i));
    }
};

```

EasyLocal 3	EasyLocal 4
base NeighborhoodExplorer<Input, Solution, Move, CostStructure>, constructor (in, sm, name)	neighborhood_explorer_base<SolutionManager, Move>, inherited constructor
the name in the constructor	a static name(), which the tester puts on its Move page
FirstMove (throws EmptyNeighborhood), NextMove	first_move, next_move, both returning false when there is no move; or a generator moves(solution) (chapter 4)
RandomMove(st, mv) (throws EmptyNeighborhood)	random_move(solution, rng), returning std::nullopt when there is no move
FeasibleMove	is_valid
MakeMove	make_move
AddDeltaCostComponent	the recipe: neighborhood<E>() \ delta<C, D>()
the Inverse function given to TabuSearch	inverse(solution, move, tabu_move) in the explorer

The move logic barely changes. `first_move` writes a move and returns true on success, or false for an empty neighborhood. `random_move` returns the move directly, or `std::nullopt` when there is none.

The example's `name()` returns a `std::string_view`: a read-only view of text, represented by a pointer and a length. It avoids copying the string. A view must not outlive its text, so a literal such as "2-opt" is a natural fit. You can also return a `std::string`; the library accepts either.

The alternative to a cursor is a **moves(solution) generator**. This coroutine produces one move with `co_yield`, then resumes when the caller asks for the next. For a swap neighborhood, the entire enumeration is two nested loops:

EasyLocal 4

```

// Every pair of positions i < j, one move at a time.
easylocal::generator<SwapCities> moves(const Tour& tour) const
{

```

```

const auto n = tour.order.size();
for (std::size_t i = 0; i < n; ++i)
    for (std::size_t j = i + 1; j < n; ++j)
        co_yield SwapCities{i, j};
}

```

The generator keeps the loop state for you. Both forms produce moves only as needed: First Improvement stops enumeration as soon as it accepts a move. Algorithms use either form in the same way.

Generators are usually easier to write; cursors let you keep the structure of an EasyLocal 3 explorer and need no coroutine. You can keep a cursor throughout the port and switch later if useful. If an explorer provides both, the library uses the cursor. [Chapter 4](#) compares the two.

8. The delta cost components

EasyLocal 3

```

class TwoOptTourLengthDelta : public DeltaCostComponent<Input, Tour, TwoOpt, double>
{
public:
    TwoOptTourLengthDelta(const Input& in, TourLength& cc)
        : DeltaCostComponent<Input, Tour, TwoOpt, double>(in, cc,
"TwoOptTourLengthDelta") {}

    double ComputeDeltaCost(const Tour& st, const TwoOpt& mv) const override
    {
        const auto n = st.tour.size();
        const auto first = st.tour[mv.first_edge];
        const auto first_next = st.tour[(mv.first_edge + 1) % n];
        const auto second = st.tour[mv.second_edge];
        const auto second_next = st.tour[(mv.second_edge + 1) % n];
        return in.Distance(first, second) + in.Distance(first_next, second_next)
            - in.Distance(first, first_next) - in.Distance(second, second_next);
    }
};

```

EasyLocal 4

```

class TwoOptLengthDelta : public easylocal::input_base<Tsp>
{
public:
    using input_base::input_base;

    // The tour goes a -> b ... c -> d; after the move it goes a -> c ... b -> d,
    // the segment b ... c reversed, at the same cost with symmetric distances.
    double delta_evaluate(const Tour& tour, const TwoOpt& move) const
    {
        const auto n = tour.order.size();
        const auto a = tour.order[move.i];
        const auto b = tour.order[move.i + 1];
        const auto c = tour.order[move.j];
        const auto d = tour.order[(move.j + 1) % n];
        const auto& distance = input().distance;
        return distance[a][c] + distance[b][d] - distance[a][b] - distance[c][d];
    }
};

```

- Rename `ComputeDeltaCost` to `delta_evaluate`. It still returns the change in the component's value, without its weight.

- Link the delta to its component in the recipe with `delta<TourLength, TwoOptLengthDelta>()`. A delta can also be a member of the component itself (a *co-located* delta, [chapter 5](#)).
- Leave out deltas you have not ported yet. The framework evaluates those components on a copy of the solution with the move applied, as EasyLocal 3 did for components added to the explorer with `AddCostComponent`.
- [Check each delta](#) before relying on it: a wrong sign or index is easy to miss during a port.

9. Several neighborhoods

A `MultimodalNeighborhoodExplorer` (set union with biases) becomes a `neighborhood_union` of the explorers' recipes, each with its own deltas:

EasyLocal 4

```
auto both =
  el::neighborhood_union(
    el::neighborhood<TwoOptExplorer>()
      | el::delta<TourLength, TwoOptLengthDelta>(),
    el::neighborhood<SwapExplorer>()
      | el::random_biases(3.0, 1.0);
```

`random_biases(3.0, 1.0)` makes random selection choose the 2-opt neighborhood three times out of four. As in the old multimodal explorer, each child's probability is proportional to its bias.

The biases are optional and default to 1. This gives equal probability to each *child*, regardless of how many moves it contains. Adjust them to account for neighborhood size or usefulness. A bias of 0 excludes a child from random selection; enumeration still includes all children. In an app, change or tune them through `--neighborhood.random_biases` without recompiling.

The union represents moves as a variant of its children's move types, replacing EasyLocal 3's manual `ActiveMove` bookkeeping. A component uses delta evaluation only when every child provides a delta for it ([chapter 7](#)).

Composing and running

10. From the objects of main to a recipe

Composition is the largest change in the port. In EasyLocal 3, `main` read the instance, constructed the components and registered them with each other. All these objects had to stay alive during the search. A second instance or a different cost required another set:

EasyLocal 3

```
Input in(instance);
TspSolutionManager sm(in);
TourLength length(in);
TwoOptNeighborhoodExplorer nhe(in, sm);
TwoOptTourLengthDelta delta(in, length);
sm.AddCostComponent(length);
nhe.AddDeltaCostComponent(delta);
```

EasyLocal 4 separates the **description** of a search from the **objects** that run it. You write the description once, without an `Input`. The framework then builds the objects for a particular `Input` when you bind the search.

You still write the same four kinds of service: the `SolutionManager`, cost components, explorer and deltas from steps 5 to 8. Each takes what it needs to work, such as the `Input` or the `SolutionManager`. Two **recipes** describe how they fit together.

The first recipe describes the *cost layer*: which SolutionManager to build and how to evaluate a solution.

EasyLocal 4

```
auto sm = el::solution_manager<TourManager>() | el::component<TourLength>();
```

`solution_manager<TourManager>()` selects the class, and `| component<TourLength>()` adds its cost. For several components, use an expression such as `cost::hard_soft(...)` or `cost::sum(...)`, including any weights (step 6). This replaces `AddCostComponent` and states how the components combine.

The factory names follow a common pattern: pass your class as a template argument to `solution_manager<M>()`, `component<C>()`, `neighborhood<E>()`, `delta<C, D>()` or `runner<A>("name")`. These create recipes or registrations. Factories named `make_` create the runner or solver you will use: `make_runner<Algorithm>(parameters)` below and `make_solver(runner)` in [chapter 9](#).

Pass any extra constructor arguments to the factory, for example `component<Excess>(ExcessParameters{.bound = 8.0})`. The recipe keeps them until it constructs the component.

The second recipe describes the *delta cost layer*: the neighborhood and the deltas available for evaluating its moves.

EasyLocal 4

```
auto nhe =
    el::neighborhood<TwoOptExplorer>() | el::delta<TourLength, TwoOptLengthDelta>();
```

`delta<TourLength, TwoOptLengthDelta>()` replaces `AddDeltaCostComponent`. The first type identifies the cost component; the second supplies its delta. Omit components that have no delta for this neighborhood: the framework evaluates them on a copy of the solution with the move applied.

Both recipes are ordinary values. You can copy them, return them from functions or share them through a header, as the TSP example does in `examples/tsp/apps.hpp`. So far, they hold only types and parameters. No `TourManager` or `TourLength` has been constructed, and no `Input` is needed.

A runner is an algorithm plus the two recipes:

EasyLocal 4

```
auto fi =
    el::make_runner<runners::FirstImprovement>(runners::FirstImprovementParameters{})
    | sm | nhe;

auto search = fi.bind(tsp);
const auto result = search.run(search.initial_solution());
```

- `make_runner<runners::FirstImprovement>(parameters)` selects the algorithm and stores its parameters.
- `| sm | nhe` attaches the `SolutionManager` and cost, then the neighborhood and deltas. The framework deduces the types you previously supplied as `<Input, Solution, Move, CostStructure>`. The order matters: attach `sm` before `nhe`. At this point, `fi` is still independent of any instance.
- `fi.bind(tsp)` constructs the services and algorithm. The returned *bound runner* owns them all, but borrows `tsp`. The `Input` must outlive the bound runner; `bind` rejects a temporary `Input`.
- `search.run(solution)` returns the final solution, cost, iteration and evaluation counts, and termination reason. Get a starting point with `search.initial_solution()` or `search.random_solution(rng)`, the counterparts of `GreedyState` and `RandomState`.

You can bind `fi` again to another instance or use it on another thread. Each bind creates its own services. Searches on the same instance share only the immutable Input, so their mutable state stays separate. Step 13 uses this composition to run searches with different costs. You can reuse the problem definition across experiments without rebuilding the object wiring by hand.

Missing pieces are reported at compile time, where you compose the search. For example, `solution_manager<TourManager>()` without a cost produces:

```
a SolutionManager recipe needs a cost: the cost is always computed by cost
components, add `| component<C>()` or a cost expression such as
`| cost::sum(component<A>(), component<B>())`
```

In EasyLocal 3, forgetting to add a cost component could instead produce a program that compiled and ran with a cost of zero.

An app puts names on runners. Combine the two recipes with a named registration for each algorithm your program offers. This app has two; an app with a single runner works the same way:

EasyLocal 4

```
auto application = el::app("tsp") | sm | nhe
    | el::runner<runners::FirstImprovement>("fi")
    | el::runner<runners::SimulatedAnnealing<Classic>>("sa");
```

The command-line program (step 11), interactive tester (step 12) and REST service all take this app. Each uses a Session to bind it to an Input, keep a current solution and run algorithms by name: `--runner sa` calls `session.run("sa")`. This replaces the wiring around EasyLocal 3's Solver, SetRunner and `--main::method` ([chapter 12](#)).

The pipes are a shortcut for named members. These can make composition easier to read while you learn the vocabulary:

EasyLocal 4

```
auto same_runner =
    el::make_runner<runners::FirstImprovement>(runners::FirstImprovementParameters{})
        .with_solution_manager(
            el::solution_manager<TourManager>().with_cost(
                el::component<TourLength>())
        )
        .with_neighborhood(
            el::neighborhood<TwoOptExplorer>()
                .with_delta<TourLength, TwoOptLengthDelta>());
```

`.with_solution_manager(...)`, `.with_cost(...)`, `.with_neighborhood(...)` and `.with_delta<C, D>()` build exactly the same runner as the pipes. Both forms are public API, and you can mix them. Named members make each part explicit; pipes keep repeated compositions short.

11. The main program

An EasyLocal 3 main handled parameter registration, two rounds of command-line parsing, component construction, tester setup, runner selection and output. Much of that code is now handled by the framework.

For a command-line program, define the app and call `cli::run`, as in `examples/tutorial/cli_main.cpp`:

EasyLocal 4

```

auto application = el::app("tsp")
| (el::solution_manager<TourManager>() | el::component<TourLength>())
| (el::neighborhood<TwoOptExplorer>()
  | el::delta<TourLength, TwoOptLengthDelta>())
| el::runner<runners::FirstImprovement>("fi")
| el::runner<runners::SimulatedAnnealing<Classic>>(
  "sa",
  {.temperature = {
    .initial_temperature = 10.0,
    .final_temperature = 0.1,
    .samples_per_temperature = 50}});

return el::cli::run(application, argc, argv);

```

EasyLocal 3	cli::run
ParameterBox main_parameters, Parameter<T>, CommandLineParameters::Parse	--instance, --seed, --runner, --start, --solution, --output, and --config <file>
--main::method and SetRunner	--runner fi: a runner registered in the app, by name
the runners' parameters, --SA::cooling_rate	--runners.sa.temperature.cooling_rate; --help lists them all, with their domains and current values
Random::SetSeed(seed)	--seed, the seed of the run's random generator
--main::init_state	--solution <file>, or --start initial / random
LowerBoundReached, a runner's max_evaluations and timeout	--target, --max_evaluations, --timeout, which bound the run whichever runner it uses
PrintViolations on the final state	--report: the value of each cost component, with its description
the search progress printed by hand from the runner	--trace run.jsonl, the events of the run (what EasyLocal 4 adds)
a tuning scenario written by hand	--tuning.irace <dir> (chapter 13)
solver.Solve(), printing result.cost and result.output	the run, then cost, time, the iterations, evaluations and termination, and the solution, or --output <file>
IsSet() checks	the parameters' own validation, with an exit status of 2

The program runs as before, with plain switches in place of --main::

EasyLocal 4 — output

```

$ easylocal_tutorial_cli --instance five.tsp --runner fi --seed 1
cost 26
time 1.9125e-05
iterations 2
evaluations 9
termination local optimum
0 1 3 4 2

```

Pass your program's own parameters as a parameter set: `el::cli::run(application, argc, argv, {.program_parameters = own})`. They are parsed alongside the framework's parameters ([chapter 12](#)).

For a solver, wrap a runner with `make_solver` ([chapter 9](#)). `solvers::LocalSearch` replaces `SimpleLocalSearch`, and `solvers::MultiStart` replaces `MultiStartSearch`. Step 13 shows a solve in stages.

12. The tester

EasyLocal 3 used a `Tester` for the `SolutionManager` and a `MoveTester` for each explorer, with text menus on standard input:

EasyLocal 3

```
Tester<Input, Tour, CostStructure> tester(in, sm);
MoveTester<Input, Tour, TwoOpt, CostStructure> two_opt_test(in, sm, nhe, "2-opt",
tester);
tester.RunMainMenu();
```

EasyLocal 4 provides the `TextUI` through the optional `TUI` component. Request it with `find_package(EasyLocal CONFIG REQUIRED COMPONENTS Core TUI)` and link `EasyLocal::TUI`.

The `TextUI` takes the app from step 10, which already describes the neighborhood and runners. Its options set the title, seed and starting input:

EasyLocal 4

```
auto application = el::app("tsp")
| (el::solution_manager<TourManager>() | el::component<TourLength>())
| (el::neighborhood<TwoOptExplorer>()
| el::delta<TourLength, TwoOptLengthDelta>())
| el::runner<runners::FirstImprovement>("fi")
| el::runner<runners::SimulatedAnnealing<Classic>>(
"sa",
{.temperature = {.initial_temperature = 10.0, .final_temperature = 0.1}});

el::tui::run(
application,
{
.title = "TSP tester",
.seed = 2026, // the RNG for random solutions, moves and stochastic runners
.input_path = EASYLOCAL_TUTORIAL_INSTANCE, // loaded with the read_input hook
});
```

The `Input/Output`, `Move` and `Run` pages replace the old `State`, `Move` and `Run` menus. Use them to load or create solutions, inspect moves and their deltas, edit parameters and watch a search. [Chapter 14](#) describes the controls.

Here the `Move` page shows the best 2-opt move and its delta check, the counterpart of *Perform Best Move* followed by a cost check:

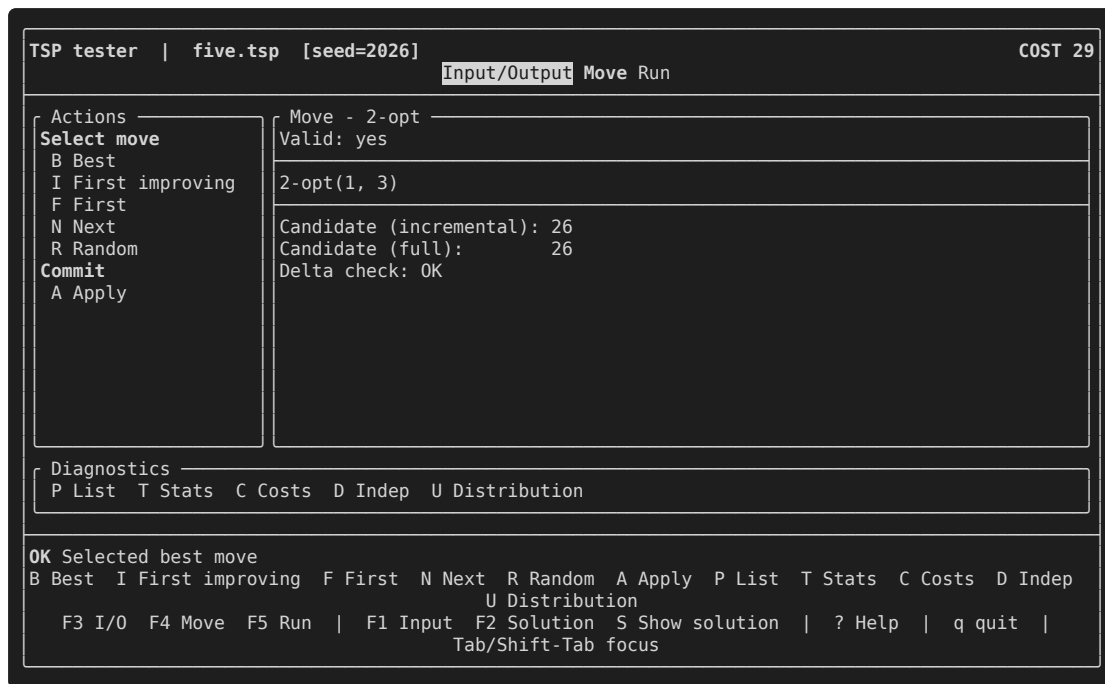


Figure 1: The Move page of the TextUI, with the best 2-opt move and its delta check

To inspect several neighborhoods on the same solution, use a launcher with one app per neighborhood. Build the apps from the same SolutionManager recipe; the launcher carries the Input and current solution between them:

EasyLocal 4

```
easylocal::tui::run_launcher(
{
    .title = "EasyLocal TSP Tester",
    .tester =
    {
        .seed = 0,
        .input_path = EASYLOCAL_TSP_INSTANCE_FILE,
        .solution_path = EASYLOCAL_TSP_SOLUTION_FILE,
    },
},
tsp::two_opt_app(),
tsp::swap_app(),
tsp::union_app());
```

`two_opt_app()` and `swap_app()` are two apps of the TSP example (`examples/tsp/apps.hpp`), one with the 2-opt neighborhood and one with the swaps ([chapter 14](#)).

13. A solve in two stages

A common solver first searches on hard constraints until it finds a feasible solution, then continues on the whole cost. EasyLocal 3 needed two sets of services for this: one with only the hard components, and one with all of them. The program connected the stages with `solverH.Solve()` followed by `solver.Resolve(stage_one.output)`.

In EasyLocal 4, the solver derives the hard-only stage from a single runner and its recipes. The example in `examples/tutorial/staged_main.cpp` treats TSP edges longer than 8 as violations:

EasyLocal 4

```

// One hierarchical cost: edges longer than 8 are violations (hard), the
// length is the objective (soft). EasyLocal 3 needed a SolutionManager
// for each set of components (all, hard only); with_hard_cost() derives
// the hard-only runner from this one.
auto sm = el::solution_manager<TourManager>()
    | el::cost::hard_soft(
        el::cost::apply(
            [](double longest) { return std::max(0.0, longest - 8.0); },
            el::component<MaxEdge>()),
        el::component<TourLength>());

// The explorer and its deltas are written once: the hard-only stage
// ignores the delta of the soft TourLength.
auto descent =
    el::make_runner<runners::FirstImprovement>(runners::FirstImprovementParameters{})
    | sm
    | (el::neighborhood<TwoOptExplorer>()
        | el::delta<TourLength, TwoOptLengthDelta>());

```

This program reads its own parameters because `cli::run` dispatches named runners, while this search uses a solver. `load_and_apply` provides parameter loading ([chapter 10](#)). Set parameters that depend on the instance after loading the Input and before the solver copies the runner, as you would with `SetParameter` after parsing in EasyLocal 3:

EasyLocal 4

```

// Parameters that depend on the instance, set once it is read, as
// EasyLocal 3's SetParameter("max_evaluations", ...) after the parsing.
const auto tsp = el::load_input<Tsp>(main_parameters.instance);
if (main_parameters.evaluations_per_city != 0)
    descent.parameters().max_evaluations =
        main_parameters.evaluations_per_city * tsp.cities();

```

`solvers::two_stage()` replaces the two solvers and the Resolve that passed the solution from one to the other:

EasyLocal 4

```

// EasyLocal 3's two solvers, one per SolutionManager, and the Resolve that
// passed the solution from one to the other: the descent on the hard cost
// from a random tour until it is feasible, then on the whole cost.
auto stages =
    el::solvers::two_stage(descent)
        .initialization(el::initialization::random)
        .seed(main_parameters.seed);
const auto result = stages.solve(tsp);

```

- The first stage evaluates only the components of the hard branch of `cost::hard_soft`, and stops as soon as the hard cost is zero; the second continues from its solution on the whole cost.
- A delta attached to a soft component, `TwoOptLengthDelta` here, is ignored by the first stage and used by the second.
- One runner serves both stages; `two_stage(first, second)` takes two, for example with different parameters or neighborhoods.
- `result.iterations` and `result.evaluations` add up both stages, and `result.stages` reports each one.
- `two_stage()` is a pipeline of two stages, `(stage("first", descent) & until_feasible()) | stage("second", descent)` in `solvers`: a solver of more stages, each with its own runner, neighborhood

and cost, is written the same way (see [Solvers](#)). A stage can also be repeated: `& attempts(10)` on the first one restarts it from new random tours while the tour is not feasible.

The report reads each cost branch separately. To replace EasyLocal 3's `cc.Cost(output)`, construct a component from the Input and call `evaluate`:

EasyLocal 4

```
std::cout << "violations " << result.cost.hard() << ", length " << result.cost.soft()
<< '\n'
<< "iterations " << result.iterations // of both stages
<< ", termination " << el::to_string(result.termination) << '\n';
// The value of each component: components are plain classes, constructed
// from the Input (EasyLocal 3's cc.Cost(out) on each component).
std::cout << "TourLength " << TourLength{tsp}.evaluate(result.solution)
<< ", MaxEdge " << MaxEdge{tsp}.evaluate(result.solution) << '\n';
el::write_solution(tsp, result.solution, std::cout);
```

EasyLocal 4 — output

```
$ easylocal_tutorial_staged --main.instance five.tsp --main.seed 1
violations 0, length 26
iterations 3, termination local optimum
TourLength 26, MaxEdge 8
0 4 2 3 1
```

Checking the port

The compiler catches many integration errors before you run the program: incompatible hook signatures, delta types that cannot be added to their components, missing costs and neighborhoods that do not support the chosen runner. Diagnostics point to the composition and explain what is needed (step 10).

Compilation cannot catch a wrong delta formula or a change in neighborhood enumeration and sampling. EasyLocal 3 checked these through the `MoveTester` menu. EasyLocal 4 also exposes the checks as functions, so you can repeat them automatically across instances at every build.

Check deltas during a search. Compile with `-DEASYLOCAL_VERIFY_DELTAS` (or set it through `target_compile_definitions`). After each applied move, the search compares the incremental cost with a full evaluation. The first disagreement stops the run and identifies the component:

```
EASYLOCAL_VERIFY_DELTAS: the delta of cost component #1 disagrees with its full
evaluation after a move
```

Full evaluation slows the search considerably, so enable this in debug builds or tests and disable it for performance measurements ([chapter 5](#)). It is especially useful when move fields change meaning during the port. Here, EasyLocal 3's `TwoOpt` identifies the two *edges* to remove, while the tutorial uses *positions* to locate the segment to reverse.

Test components individually. A *fixture* contains an Input and the solutions to check. Include a tour with cities out of their original order to expose deltas that confuse positions with city identifiers. Each check constructs the relevant component and returns a report for your test to assert on ([chapter 11](#)):

Check	What it verifies
<code>check_solution_manager(f)</code>	the fixture's solutions, the initial one and random ones are valid
<code>check_cost_component<Component>(f)</code>	evaluating the same solution twice gives equivalent values
<code>check_neighborhood<NHE>(f)</code>	enumerated and sampled moves are valid, keep the solution valid and change it; random moves are enumerated ones, drawn from the generator given
<code>check_delta_cost_component<NHE, Component, Delta>(f)</code>	after each valid move, value + delta equals the full re-evaluation

These functions live in `easylocal::testing`. Use `run_checks(...)` to run several checks and obtain a program exit status. They cover the work of EasyLocal 3's `ComponentTester` and the cost checks in `MoveTester`, making those checks part of a repeatable test suite.

Check the whole app. `check(app, input)` builds the app's services and runs the checks above on initial and random solutions. Its report counts how many passed:

EasyLocal 4

```
const auto report = el::check(application, tsp); // also: check(app, input, solution)
el::print_report(std::cout, report);
if (!report)
    return 1;
```

```
EasyLocal tsp check: 503 checks passed
composition: solution_managers=1, cost_components=1, neighborhoods=1, delta_bindings=1,
runner_registrations=2
```

It also validates the app's setup: runner names must be distinct and usable on the command line, runners must be constructible from their parameters, and every parameter must declare a domain. The domain check supports [tuning](#); the full list of checks is in [chapter 15](#).

Check a particular solution. A `Session` holds an `Input` and current solution, like the tester. It offers the three familiar `MoveTester` checks: *Check Neighborhood Costs*, *Check Move Independence* and *Check Random Move Distribution*. Each returns counters you can inspect or assert on:

EasyLocal 4

```
// Each check enumerates the neighborhood of the current solution and
// returns a struct of counters (Session::..._result).

// The delta evaluation of each move against the full evaluation of the
// solution it leads to: moves (enumerated), mismatches (the two costs
// differ), invalid (moves that are not valid or lead to an invalid
// solution).
const auto costs = session.check_neighborhood_costs();

// What each move does to the solution: moves, null_moves (moves that leave
// it unchanged), repeated_states (moves that lead to a solution an earlier
// move reached), invalid.
const auto independence = session.check_move_independence();
```

```
// random_move against the enumeration: neighborhood_size (valid enumerated
// moves), samples (draws, 20 per move by default), out_of_neighborhood
// (draws that return no move or one the enumeration does not contain),
// unseen (enumerated moves never drawn), min_frequency and max_frequency
// (how often the least and the most drawn moves came up).
const auto sampling = session.check_random_move_distribution();

if (costs.mismatches != 0 || costs.invalid != 0 || sampling.out_of_neighborhood != 0)
    return 1;
```

The TextUI exposes these three checks on its Move page and `check(app, input)` on its Input/Output page (step 12). You can investigate a problem interactively, then use the same checks in a regression test.

Parameters, configuration and tuning

In EasyLocal 3, each `Parameter<T>` was registered in a `ParameterBox` and parsed as `--box::name`. You used `IsSet()` to check for a value and `SetParameter("name", value)` to change it.

In EasyLocal 4, related parameters live in a **parameter block**: a plain struct whose fields are `easylocal::parameter` and whose `parameter_schema()` describes them. Algorithms, temperature policies, tabu lists, cost expressions and neighborhoods use these blocks. Your classes can use them too:

EasyLocal 4

```
struct ExcessParameters
{
    easylocal::parameter<double, "bound"> bound{8.0};

    static consteval auto parameter_schema()
    {
        return easylocal::config::fields(
            easylocal::config::field<&ExcessParameters::bound>(
                "Longest edge that is not a violation",
                easylocal::config::range(0.0, easylocal::unlimited)));
    }
};
```

The default of a field is its member initializer: `bound{8.0}` here, where EasyLocal 3 set the default in the constructor of the parameter. A field written `bound{}` has no default and must be set, which the parameters whose value belongs to the problem are: the temperatures of Simulated Annealing and the budget of a search, which EasyLocal 3 gave arbitrary values.

The schema is read at compile time. It gives each field a **description** for `--help` and the tester, and a **domain** of allowed values. For example, use `config::range(0.0, 1.0).open()`, `config::one_of(...)` or `easylocal::unlimited` for an unrestricted domain.

It can also express relationships between fields. Use `.only_if(...)` for a parameter that applies only under a condition, or a requirement such as `config::require(value<"final_temperature"> < value<"initial_temperature">, "...")`. You declare these rules once; the framework enforces them wherever parameters are applied. This replaces validation written by hand after parsing ([chapter 10](#)).

A **parameter set** collects the blocks under paths. `runner.configuration()` uses search for the algorithm, cost for the cost expression and neighborhood for the neighborhood. You can add a prefix: `configuration.add("descent", descent.configuration())` produces paths such as `--descent.search.*`.

An app collects the parameters automatically for its registered runners. With `cli::run`, use `--runners.<name>.*` for an algorithm, and `--cost.*` or `--neighborhood.*` for the parts shared by all runners.

EasyLocal 3	EasyLocal 4
Parameter<double> cooling("cooling_rate", "...", box)	a field of the block, and a line of its parameter_schema()
--SA::cooling_rate 0.95	--runners.sa.temperature.cooling_rate 0.95, listed by --help with its domain and its current value
the weights given to each CostComponent constructor	--cost.weights, the weights of the cost::sum, set without recompiling
parameter.IsSet()	nothing to ask: every field has a default, and the diagnostics say what was rejected
runner.SetParameter("max_evaluations", n)	runner.parameters().max_evaluations = n, before the runner is bound (step 13)
CommandLineParameters::Parse(argc, argv, ...)	cli::run(app, argc, argv), or config::load_and_apply(argc, argv, set) in a program that is not built on an app
—	the parameter window of the interactive tester, which edits the same set

Overrides are validated and applied together. If any value is invalid, all parameters stay unchanged. The command-line program reports the rejected path and reason, then exits with status 2.

The same schema also provides configuration and tuning support, saving you from maintaining separate descriptions of the parameters:

- **Configuration files.** `--config <file>` reads `path = value` lines and accepts `#` comments. Command-line values take precedence. The optional `ConfigTOML` component supports TOML, where tables supply path prefixes: `[solver.search.temperature]` can contain `cooling_rate`. Keep a configuration file alongside your results to record an experiment's settings ([chapter 10](#)).
- **Tuning.** In a program using `cli::run`, `--tuning.irace <dir>` exports an `irace` scenario from the parameters with finite domains. It includes domains, conditions, requirements as `[forbidden]` expressions and a stub to run the program. Since the scenario comes from the program's schema, you do not have to keep a separate parameter file in sync with its command-line switches ([chapter 13](#)).
- **Domain checks.** `check(app, input)` catches parameters with no declared domain. Choose a useful domain for tuning: for a temperature, `config::range(1.0, 100.0).log()` is more useful than `unlimited`.

What EasyLocal 4 adds

Once the port works, you can add capabilities that previously required application-specific code:

- **Inspect a search with tracing.** `easylocal::trace` records events such as evaluated and accepted moves, incumbent updates, temperature changes and termination. Use `memory_recorder` in code or save a trace with `--trace run.jsonl`; JSON Lines and binary ELTR are supported. Tracing is selected at compile time, so disabled tracing adds no recording overhead. Use `scripts/eltr.py` to decode saved runs ([chapter 17](#) and [Tracing](#)).
- **Observe and stop a run.** `run_control` combines a `std::stop_token` with a progress observer. Pass it with `with(control)`, or set a target and time limit with `stop_at(cost)` and `timeout(s)`. Every runner uses `search_run` for progress and cancellation, so a frontend can monitor and stop searches from another thread without changes to each algorithm.

- **Expose the app over HTTP.** The optional REST component lets clients submit runs, poll progress, cancel searches and fetch solutions using the same app definition ([chapter 16](#)).
- **Search several objectives.** `cost::objectives` defines a Pareto cost vector, and the run returns the non-dominated solutions it found. `runners::ParetoLateAcceptanceHillClimbing` searches for a front ([chapter 3](#)).
- **Compose solvers from stages.** Give each stage its own runner, cost and neighborhood, then add repetition or conditions as needed. You can express the search strategy without writing a Solver subclass ([chapter 9](#)).

A porting checklist

Before relying on the port's results:

1. Run `check(app, input)` on a representative instance and the smallest one.
2. Run a search with `EASYLOCAL_VERIFY_DELTAS` and resolve any disagreement.
3. Save a solution, read it back and verify that its cost matches the printed cost. This checks that the I/O hooks in step 4 agree.
4. Choose deliberately between `cost::hard_soft` and a weighted sum (step 6). Check feasibility and account for the difference in Simulated Annealing's behaviour.
5. Use `--help` to verify parameter values against the old program. Names and defaults may differ.
6. Check reproducibility with `--seed`, then compare the two versions across several seeds.
7. Declare a domain for every parameter you add, so it passes check and can be considered for [tuning](#).

Reference: the two frameworks side by side

The concepts

Use this table to find where each EasyLocal 3 concept belongs in EasyLocal 4 and which step explains the migration.

EasyLocal 3	EasyLocal 4
Input, Solution and Move (the State of RandomState, PrintState)	the same three values, plain: no base class, no back pointer to the Input (steps 1 to 3)
SolutionManager (RandomState, GreedyState, CheckConsistency)	SolutionManager: random_solution, initial_solution, is_valid — construction and validity only, no cost (step 5)
CostComponent::ComputeCost	a cost component's evaluate, with no base class; the recipe attaches it, component<C>() (step 6)
PrintViolations	an optional describe(solution) of the cost component, which returns the text instead of printing it
the hard flag and the weight of a component, HARD_WEIGHT	a cost expression in the recipe: cost::hard_soft, cost::sum, cost::weighted, cost::in_order, cost::apply, cost::objectives; hard is a level, not a big weight (step 6)
CostStructure, DefaultCostStructure<CFtype>, the pair violations/objective	nothing to declare: the cost and its type follow from the components and the expression
DeltaCostComponent::ComputeDeltaCost, AddDeltaCostComponent	a delta cost component's delta_evaluate, bound to its component by delta<C, D>() in the recipe (steps 8 and 10)
NeighborhoodExplorer (FirstMove, NextMove, RandomMove, MakeMove, FeasibleMove)	a NeighborhoodExplorer with a cursor (first_move/next_move) or a moves() generator, plus random_move, make_move, is_valid (step 7)
MultimodalNeighborhoodExplorer, ActiveMove	neighborhood_union, whose move is a variant of the children's moves, with random_biases (step 9)
ParallelNeighborhoodExplorer (TBB)	no counterpart (roadmap)
Kicker, KickerTester	no counterpart (roadmap)
Runner subclasses, MoveRunner	algorithm classes in easylocal::runners with one run member, composed with the recipes into a runner (step 10)
Solver (SimpleLocalSearch, MultiStartSearch, ...), SetRunner, Solve, Resolve	easylocal::solvers and their pipelines, or an app whose runners a Session runs by name (steps 10, 11 and 13)
Tester, MoveTester	tui::run(app, options), the TextUI adapter (step 12), and the checks of check and of a Session
ComponentTester	easylocal::testing, which checks a component against a fixture in a unit test
ParameterBox, Parameter<T>, CommandLineParameters, SetParameter	a parameter block per component, with a schema, domains and conditions, gathered in a parameter_set (the parameters)
Random::Uniform, Random::SetSeed, the global generator	an RNG passed to the members that need one, and --seed for the run's
Boost.program_options, TBB	no dependency beyond the standard library: Core is header-only and parses the command line itself

The design changes behind these names have practical consequences:

- **Smaller component interfaces.** Concepts check capabilities at compile time, with no virtual dispatch or required base classes. Implement only the members your algorithms and tools use: an explorer used solely for Simulated Annealing needs random moves but no enumeration.
- **Separate state for each search.** Services are constructed for an immutable Input and owned by the bound runner. Their constructors can precompute instance data; solution and move values need no Input reference ([problem model](#)).
- **Reusable cost definitions.** Recipes compose cost components and deltas. The expression keeps hard/soft roles and weights together, making it easier to try another cost without changing the components. With `cost::hard_soft`, soft improvements cannot outweigh a hard violation.
- **Explicit randomness.** Passing RNGs into services makes the source of random choices visible and lets each run control its own generator.
- **Less work in custom runners.** `search_run` supplies counters, budgets, cancellation, progress and events, so each algorithm can focus on its search decisions.
- **Earlier integration checks.** Missing capabilities are reported where components are composed, with diagnostics describing what to provide.

The runners and the solvers

EasyLocal 3	EasyLocal 4
FirstDescent	runners::FirstImprovement (the scan restarts from the first move)
SteepestDescent	runners::BestImprovement (ties go to the first best move, not drawn at random)
HillClimbing	runners::HillClimbing
LateAcceptanceHillClimbing	runners::LateAcceptanceHillClimbing (the history records the current cost, see Runners)
GreatDeluge	runners::GreatDeluge
SimulatedAnnealing (max_neighbors_sampled, max_neighbors_accepted)	runners::SimulatedAnnealing<temperature::Hybrid>, the same schedule; temperature::Classic is the textbook one, which cools on samples alone and ends at the final temperature
SimulatedAnnealingOnlyCutoff, SimulatedAnnealingFixedTemperature, SimulatedAnnealingTimeBased	SimulatedAnnealing with temperature::Cutoff, FixedTemperature, TimeBased
SimulatedAnnealingWithReheating, ... TimeBasedSimulatedAnnealing	temperature::Reheating<Hybrid>>, Reheating<TimeBased>; every schedule with an initial temperature can be reheated
TabuSearch, FirstImprovementTabuSearch	runners::TabuSearch, runners::FirstImprovementTabuSearch, and AspirationPlusTabuSearch, EliteCandidateTabuSearch
min_tenure, max_tenure	a tabu list policy: runners::tabu::FixedLength (one tenure), RandomTenure (drawn in a range), and also Cyclic, Reactive, Frequency
max_idle_iterations	max_idle_iterations, in the runner's parameter block, as every other limit
the Inverse function given to TabuSearch	inverse(solution, move, tabu_move) in the explorer
SimpleLocalSearch, MultiStartSearch	solvers::LocalSearch, solvers::MultiStart
a solve in stages, written by hand with Resolve	solvers::two_stage, or a solvers pipeline of stages
—	runners::ParetoLateAcceptanceHillClimbing, for a multi-objective cost

Each algorithm is a class template with its own parameter block, listed in [Runners](#). To write your own, build on `search_run` ([chapter 8](#)).

For a concrete extension of SA, follow [Advanced: customizing Simulated Annealing](#). It shows EL4's actual search loop, enables the existing Reheating policy, then writes a new policy that reheats after consecutive rejected moves. The example keeps the framework's delta evaluation, best-solution tracking, configuration, cancellation and tracing.

What is not there yet

These EasyLocal 3 features do not yet have a direct counterpart:

EasyLocal 3	Planned
ShiftingPenaltyRunner	Shifting penalty
SimulatedAnnealingWithLearning	Adaptive neighborhood selection
SampleTabuSearch	Candidate strategies of Tabu Search
kickers, VariableNeighborhoodDescent	Kicks, Iterated Local Search and Variable Neighborhood Descent
TokenRingSearch	Cooperative runners, for the runners that exchange solutions; a fixed order of runners is a pipeline today
GRASP (GreedyState(alpha, k))	not planned: a greedy randomized construction is a <code>random_solution</code> of its own, searched by <code>solvers::MultiStart</code>
ParallelNeighborhoodExplorer	Parallel search
the modelling layer (AutoState, expressions)	A modelling layer, a brainstorming item

Roadmap items are plans rather than commitments; see [API stability](#).

Next steps

The [tutorial](#) builds the same TSP one capability per chapter, and the [reference](#) describes every component in full.