Turning OCaml Programs Reactive with the Skip Runtime

Codex

November 15, 2025

Abstract

Reactive is a prototype OCaml library that layers the Skip runtime's persistent, incremental computation model on top of idiomatic OCaml code. This paper documents the programming model exposed by `reactive.ml`, explains how it relates to the broader Skip philosophy of reactive services, and provides practical guidance for migrating an existing OCaml program into a fully reactive pipeline. We summarize key patterns and considerations, outline operational constraints such as fixed-address linking, and describe how to validate a reactive port end-to-end.

1 Introduction

Skip's native runtime was originally created to power fully reactive backend services that continuously maintain consistent views of data and stream deltas to clients. The reactive OCaml library makes the same execution model available from OCaml by exposing a lightweight API around Skip's persistent heap, deterministic forking model, and dependency tracking primitives. Rather than recomputing entire pipelines, a program declares immutable collections, composes pure maps between them, and lets the runtime cache results and re-run only the subgraphs affected by input changes.

Applying the model effectively requires more than calling a few functions: developers must understand how persistent heaps are managed, how computations are scheduled, how trackers enforce disciplined I/O, and how to structure code so that reactivity can be introduced incrementally. This paper provides comprehensive documentation for working with the reactive OCaml library, covering both the conceptual foundations from Skip's reactive service philosophy and practical guidance for building reactive OCaml applications.

2 Philosophy of Reactive Computation

Skip's RFC 008 characterizes a *reactive service* as a compute graph of reactive collections that continually maintains derived state and exposes it via reactive resources that can be mirrored by other services or clients. Key ideas that carry over to the OCaml binding are:

- **Declarative dependency graphs.** Developers define collections and deterministic transformations between them. Skip records the dependency edges automatically and maintains the graph across executions.
- **Persistent heaps with stable code addresses.** Cached results are stored in an on-disk heap (`.rheap`) whose format requires that code and data live at known addresses.
- **Strict control of effects.** External interactions happen through tracked resources. Anything not routed via the tracker API is invisible to the runtime and therefore unsafe.

- **Gradual adoption.** RFC 008 emphasizes wrapping existing REST resources with reactive mirrors to avoid flag-day rewrites. Likewise, OCaml applications can fence off legacy imperative code and progressively move stages behind reactive maps.

3 Runtime and Library Architecture

3.1 Persistent Heaps and Pointer Stability

OCaml objects must be linked with `libskip_reactive.a`, which bundles the Skip runtime, supporting C helpers, and the Skip LLVM object. On Linux, binaries must be linked with `-no-pie` and a fixed text address (`-Wl,-Ttext=0x8000000`) so that function pointers remain stable across runs. macOS cannot enforce a fixed text address; consequently, persistent heaps cannot be reused across separate executions. Within a single process tree (`fork()` descendants) heaps are reusable on both platforms.

Persistent heaps are opened via `Reactive.init file_name size`, which maps or creates the heap, registers a custom exception for write violations, and prepares the runtime's guard pages. Once initialized, the runtime protects the heap with `mprotect` when entering worker processes to catch accidental writes.

3.2 Collections, Trackers, and Maps

Collections are opaque identifiers (`type 'a t = string`) that reference named directories inside the heap. Inputs are declared up front through `Reactive.input_files`, which stores the list of file paths, synchronizes it with the cache, and returns a collection of trackers. Each tracker enforces that file reads happen through `Reactive.read_file`; the runtime hashes file contents and remembers which map invocation consumed which tracker.

`Reactive.map` prepares the computation, then forks worker processes so that pure computations run under copy-on-write semantics. Workers call the user function with a key and an immutable array of values and must return a list of key/array pairs. The runtime deduplicates output keys and produces a new collection. `marshalled_map` wraps the same mechanism but serializes values with `Marshal.to_string` so that closures or unsupported data types can be returned at higher cost.

3.3 Observation and Lifecycle

`Reactive.get_array` can only be called after `Reactive.exit`. Before exiting, the runtime is still in "graph building" mode and will raise `Toplevel_get_array`. After exit, code pointers are reprotected read-only, making it safe to reuse cached values. Exiting twice is a fatal error. `Reactive.union` merges two collections into a combined one when fan-in is required.

4 Programming Model Reference

The public API surfaces the following primitives (types copied directly from `reactive.mli` for clarity):

- `init`
  type: `filename -> int -> unit`. Create or open a persistent heap with a fixed upper bound in bytes. Must be called before any other function.

- `input_files`
  type: `filename array -> tracker t`. Declare the set of input files. Skip records and sorts them; cached runs require the same set.

- `read_file`
  type: `filename -> tracker -> string`. Read file contents through the tracker supplied by `input_files`, ensuring the dependency is tracked.

- `map`
  type: `'a t -> (key -> 'a array -> (key * 'b array) array) -> 'b t`. Apply a pure transformation to every key's values, producing a new collection. Runs under forked worker processes and cannot be called recursively.

- `marshalled_map`
  type: `'a t -> (key -> 'a array -> (key * 'b array) array) -> 'b marshalled t`. Variant of `map` that serializes outputs so closures or custom types can be cached.

- `unmarshal`
  type: `'a marshalled -> 'a`. Deserialize values produced by `marshalled_map`.

- `get_array`
  type: `'a t -> key -> 'a array`. Access cached arrays after `exit`. Calling it earlier raises `Toplevel_get_array`.

- `union`
  type: `'a t -> 'a t -> 'a t`. Merge two collections, useful when fusing branches or joining independent maps.

- `exit`
  type: `unit -> unit`. Seal the heap, flush caches, and transition to observation mode. Required before calling `get_array`.

5 Migrating an OCaml Program

The easiest way to make an existing binary reactive is to follow a disciplined set of transformation steps.

5.1 Audit Inputs and Effects

1. **Identify stable inputs.** Files, command-line data, or database exports that should trigger incremental invalidation become entries in the array passed to `input_files`.

2. **Fence off side effects.** Anything that relies on wall-clock time, random numbers, network calls, or mutable globals must either be converted into deterministic data or moved outside the reactive pipeline.

3. **Design trackers.** Each file read should map to one or more trackers so that the runtime knows when to re-run a node.

5.2 Stage Computations into Maps

Walk the original pipeline and wrap each pure stage in its own `Reactive.map`:

- **Reader maps** use `read_file` exactly once per tracker and emit the parsed representation.
- **Transformation maps** accept the normalized data and compute derived metrics, such as uppercasing and reversing text, or fanning out words into length buckets.
- **Aggregators** combine branches via `union` or by emitting multiple keys per input.

Because maps run out-of-process, they must avoid capturing mutable OCaml state except through their arguments. Closures can be emitted only through `marshalled_map`. Raw closures passed through regular `map` will be rejected by the runtime.

5.3 Exit and Observe

Once the graph is declared, call `Reactive.exit()`. Only after exiting can downstream code fetch arrays and integrate with non-reactive subsystems (database writes, HTTP responses, etc.). Attempting to call `get_array` before exiting will raise `Toplevel_get_array`. Forking after exit is allowed and lets children reuse cached data without reopening the heap, as long as the parent remains alive.

5.4 Worked Transformation

Listing 1 sketches how an imperative file-processing script can be rewritten.

```
(* Imperative baseline *)
let summarize files =
  files
  |> Array.map (fun file ->
       let content = Stdlib.input_all (open_in file) in
       let metrics = analyze content in
       (file, metrics))

(* Reactive version *)
let summarize_reactive files =
  Reactive.init "analysis.rheap" (512 * 1024 * 1024);
  let inputs = Reactive.input_files files in
  let parsed =
    Reactive.map inputs (fun key trackers ->
      let raw = Reactive.read_file key trackers.(0) in
      [| (key, [| parse raw |]) |])
  in
  let metrics =
    Reactive.map parsed (fun key arr ->
      let summary = analyze arr.(0) in
      [| (key, [| summary |]) |])
  in
  Reactive.exit ();
  Array.map (fun file -> (file, Reactive.get_array metrics file)) files
```

Listing 1: From imperative to reactive

6 Common Patterns and Considerations

Multi-file fan-out When processing multiple input files, map functions should validate that output keys can differ from input keys. For large workloads, allocate sufficiently large heaps during initialization.

Key management Maps should treat the key argument as the canonical identifier when emitting derived keys to maintain consistency across stages.

Dependent stages Building pipelines with multiple dependent stages (e.g., word extraction followed by length calculation) requires careful reuse of keys at different logical layers.

Nested maps Calling `map` inside another `map` is not supported. Nested reactive graphs must be flattened or expressed through separate top-level maps.

Heap reuse limitations On macOS, cached heaps cannot be reopened by a fresh process; cleanup scripts should delete `*.rheap` files after each run unless debugging requires preserving them.

Tracker discipline Every file read must pass through the tracker array supplied by `input_files`; ad-hoc I/O violates dependency tracking and will compromise the reactive guarantees.

7 Operational Playbook

7.1 Build and Link

- Build the reactive library to produce `reactive.cmxa` and `libskip_reactive.a`. Linking your own program requires including `-cclib -lstdc++`.
- macOS binaries rely on custom Mach-O segments; Linux requires explicit linker flags to disable PIE and fix the text segment.
- The runtime bundles its own `main`; `runtime64_specific.cpp` strips symbols via `objcopy` when building the static library.

7.2 Process Discipline

Skip relies on `fork()` to isolate worker maps and to terminate them if the parent exits. Never call `map` while already inside another `map`; the runtime tracks this via the `toplevel` flag and raises `Can_only_call_map_at_toplevel`. Because `fork()` duplicates the process image, the code must remain single-threaded (no multicore OCaml runtime) and should avoid holding OS resources open across map boundaries unless they are read-only descriptors.

7.3 Heap Hygiene

On macOS, always delete stale heaps between executions; the runtime will otherwise exit with an error requesting manual cleanup. On Linux, heaps can be reused across program restarts as long as the binary layout is unchanged. Use cleanup scripts that delete `*.rheap` files by default and offer a keep-heaps option for debugging.

7.4 Testing Strategy

Reactive unit tests are regular OCaml binaries that link against the reactive library. Each test should link with `reactive.cmxa` and the static runtime library. Test harnesses should check for expected failures such as child processes intentionally aborting. Mirroring this workflow in downstream projects ensures regressions are caught early, especially around platform-specific invariants.

8 Best Practices and Anti-Patterns

- **Always initialize once.** The runtime tracks whether `init` has run and refuses to proceed otherwise.
- **Respect tracker usage.** Use the tracker array supplied to `map`; do not allocate new file handles or call `read_file` without the matching tracker.
- **Emit immutable data.** Values returned from `map` are assumed immutable. Modifying them afterward leads to undefined behavior because multiple keys may share the same cached array.
- **Use `marshalled_map` sparingly.** Serialization defeats structural sharing and increases heap footprint. Prefer encoding results as primitive data.
- **Expose deterministic keys.** Keys determine cache reuse. If keys depend on the execution environment (timestamps, random numbers), the runtime will never hit its cache.
- **Guard the imperative boundary.** After `exit`, copy data out before mutating it, especially when handing arrays to legacy code.

9 Future Directions

Bringing Skip's full reactive service model to OCaml would involve exposing replication tokens, diff streams, and authentication mechanisms described in RFC 008. The current prototype already models collections as DAG nodes; adding APIs for `diff` and `mirror` would let OCaml programs act as first-class reactive resources inside a larger Skip deployment. Another avenue is improving developer ergonomics by generating `map`-heavy boilerplate or by offering lint rules that detect nested map attempts or unchecked `get_array` calls.

10 Conclusion

Reactive OCaml offers a practical path toward incrementalizing existing workloads by reusing the Skip runtime's proven abstractions. By enforcing disciplined I/O through trackers, executing pure maps under forked workers, and persisting results into stable heaps, applications can scale to large data sets while avoiding redundant recomputation. The programming model provides a template for structuring pipelines, while Skip's broader reactive service philosophy illustrates how those pipelines integrate into end-to-end systems. With careful adherence to the guidelines in this document, developers can confidently port OCaml code to a reactive architecture that is both efficient and predictable.