

Reactive Dead Code Analysis Plan (Skip Runtime Alignment)

November 16, 2025

Abstract

This plan replaces the batch-only dead code analysis with a fully reactive pipeline built on the Skip runtime described in `docs/reactive_ocaml.tex`. It details the preparatory refactors (pure data extraction, deterministic summaries, graph factoring), the Skip collection design, integration with the existing build (rewatch) outputs, test strategy, and troubleshooting guidance. Each milestone enumerates concrete code changes, intermediate validation steps, and acceptance criteria so implementation proceeds from small refactors to a production-ready reactive service with < 200ms latency for file changes.

Contents

1 Overview of Changes and Outcome	1
1.1 Architectural Transformation	1
1.2 Key Refactors (Milestones 1–3)	2
1.3 Skip Integration Path (Milestone 4)	2
1.4 Expected Outcome	2
2 Goals and Non-Goals	2
3 Baseline Architecture and Constraints	3
3.1 Batch Pipeline Recap	3
3.2 Skip Runtime Constraints (<code>docs/reactive_ocaml.tex</code>)	3
3.3 Data Flow Diagram	3
4 Milestone 0: Runtime Readiness	4
4.1 Tasks	4
4.2 Acceptance	4
5 Milestone 1: Pure Data Extraction	4
5.1 Overview	4
5.2 Step-by-Step Refactor	5
5.3 Acceptance	6
6 Milestone 2: Deterministic File Summaries	6
6.1 Schema	7
6.2 Implementation Tasks	7
6.3 Acceptance	8

7 Milestone 3: Graph Store and Incremental Liveness	8
7.1 Graph Store Design	11
7.2 Pseudocode	11
7.3 Acceptance	12
8 Milestone 4: Skip Reactive Service Core	12
8.1 CMT Discovery and Input Declaration	12
8.2 Rewatch Integration	13
8.3 Skip Graph Implementation	13
8.4 Error Handling	15
8.5 Acceptance	15
9 Milestone 5: Integration (CLI, Watcher, LSP)	16
9.1 Build Coordination	16
9.2 CLI Mode	16
9.3 LSP Integration	17
9.4 Acceptance	17
10 Milestone 6: Validation and Rollout	17
10.1 Parity Harness	17
10.2 Performance Benchmarks	18
10.3 Rollout	18
11 Troubleshooting Guide	18
12 Summary	18

Implementation log. Each milestone has a matching entry in `docs/reactive_dead_code_log.md` summarizing the concrete code changes, validation steps, and commands that landed in the repository. Consult the log before starting a new milestone to ensure you're building on the verified baseline.

1 Overview of Changes and Outcome

1.1 Architectural Transformation

The batch-only analyzer mutates global tables and must rescan every `.cmt` to emit diagnostics. This plan migrates it to the Skip runtime so that:

- Collection becomes fully pure: every traversal returns `Collected_types.t`, enabling deterministic caching and parallel work.
- Per-file summaries, graph state, and diagnostics persist inside a Skip heap, so warm edits reuse cached work instead of reprocessing the whole project.
- The only tracked resource is `.reanalyze/manifest.json`; its digests describe all `.cmt/.cmti` files, so Skip re-evaluates only when manifests change.
- `reanalyze --reactive` runs once per manifest update (invoked by rewatch or a watch script): declare the graph, call `Reactive.exit`, observe diagnostics, exit. This matches the lifecycle in `docs/reactive_ocaml.tex` where each process initializes, declares, exits, and then reads results.

1.2 Key Refactors (Milestones 1–3)

- Thread a collector interface through `DeadValue`, `DeadType`, `DeadException`, and `DeadOptionalArgs`; adapt batch mode via `DeadCommon_sink` so existing tooling continues to work.
- Introduce `Collected_types`, `Collector_intf`, `Pure_collector`, `Summary`, and `Summary_cache` to produce deterministic per-file artifacts with JSON goldens.
- Build `Graph_store` and `Liveness` modules that incrementally recompute SCCs only within the frontier influenced by changed summaries.
- Add scoped helpers (`Common.with_current_module`, `ModulePath.with_current`) so Skip maps never depend on ambient mutable refs.

1.3 Skip Integration Path (Milestone 4)

- Rewatch writes `.reanalyze/manifest.json` after a successful build. A supervisor reruns `reanalyze --reactive` when that file changes.
- `Reactive_service.run_once` declares the Skip pipeline: manifest input → per-file summaries → graph store → diagnostic diffs. Individual `.cmt` files are read via regular OCaml IO inside the manifest map, relying on manifest digests for invalidation (Option A from `docs/reactive_ocaml.tex`).
- Diagnostics are pulled only after `Reactive.exit`; `Diagnostics_loop` streams them to the CLI or the LSP provider before the process exits.

1.4 Expected Outcome

- **Performance:** Cold runs match current batch latency; warm edits touch only changed summaries plus dependent SCCs (target p50 <200 ms, p95 <1 s).
- **Parity:** Batch mode remains the default baseline, and nightly local parity tests ensure reactive diagnostics are byte-identical.
- **Developer experience:** Editors receive near-real-time dead-code diagnostics via LSP, and scripts can simply call `reanalyze --reactive` after each build—no custom daemons or coordination files required.
- **Fallback:** If Skip mode fails, users fall back to `reanalyze -dce`; collector adapters keep batch behavior unchanged.

2 Goals and Non-Goals

- Maintain golden parity with current batch diagnostics for all analyses that depend on `DeadCommon` (dead values, optional args, exceptions).
- Recompute only the file summaries and SCC regions whose dependencies changed; target p50 latency < 200ms and p95 < 1 s for same project edits.
- Reuse Skip runtime primitives (`Reactive.init`, tracked resources, `Reactive.map`, `Reactive.exit`) so the tool matches `docs/reactive_ocaml.tex` constraints.

- Preserve CLI surface area: batch mode remains default for automation; reactive mode is opt-in until parity is proven.
- Non-goals: redesigning the AST, changing warning wording, or introducing new user-facing configuration.

3 Baseline Architecture and Constraints

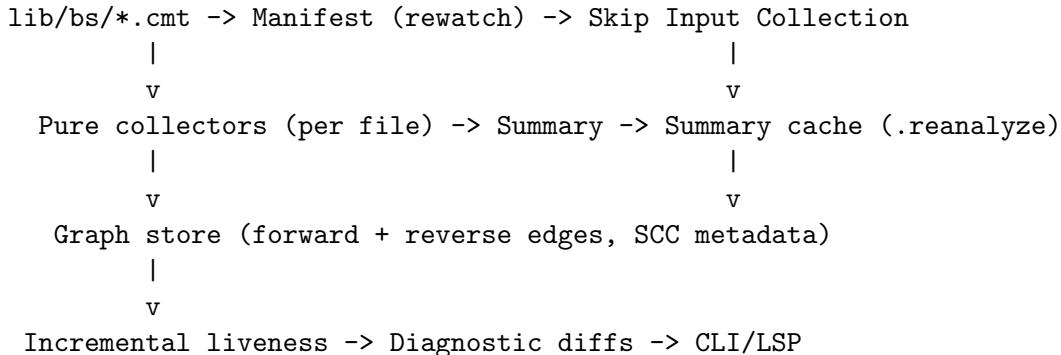
3.1 Batch Pipeline Recap

1. `Reanalyze.loadCmtFile` resolves each `.cmt/.cmti` to its source via `FindSourceFile.cmt`, sets `Common.currentSrc/currentModule/currentModuleName`, and invokes `DeadCode.processCmt`.
2. Modules under `analysis/reanalyze/src/Dead*.ml` mutate globals in `DeadCommon` (`decls`, `ValueReferences.table`, `TypeReferences.table`, `FileReferences`).
3. After all files, `DeadCommon.reportDead` resolves recursion, optional args, and emits diagnostics.

3.2 Skip Runtime Constraints (docs/reactive_ocaml.tex)

- Single initialization via `Reactive.init heap size`; Linux binaries must link `-no-pie -Wl,-Ttext=...` to keep pointers stable.
- Every file read occurs through trackers returned by `Reactive.input_files` or downstream maps; ad-hoc IO is prohibited.
- `Reactive.map` callbacks must be pure relative to inputs; any mutable globals (e.g., `Common.currentSrc`) must be encapsulated by helper functions that reset state per invocation.
- Graph declaration finishes with `Reactive.exit()`, after which downstream code observes derived arrays via `Reactive.get_array`.
- macOS cannot reopen heaps across fresh runs; service deletes stale `*.rheap` on start. Linux may reuse heaps if binary layout is constant.

3.3 Data Flow Diagram



4 Milestone 0: Runtime Readiness

This milestone exists to make Skip a first-class dependency of the analysis toolchain before any behavioural changes land. By wiring `rescript-editor-analysis` against the Skip runtime, providing a smoke-test binary, and recording baseline metrics, we de-risk future reactive work: later milestones can assume Skip artifacts are always built, the runtime loads on macOS, and we have before/after numbers for `reanalyze -dce`.

4.1 Tasks

1. Update `analysis/bin/dune`:
 - Add conditional linking of `libskip_reactive.a`.
 - On Linux, append `-ccopt -no-pie -ccopt -Wl,-Ttext=0x8000000`; on macOS, add startup hook that deletes stale heaps before `Reactive.init`.
2. Add `analysis/bin/reactive.repl` target that simply runs `Reactive.init`; `Reactive.exit`. Local smoke runs ensure Skip linkage stays healthy.
3. Capture baseline metrics (wall-clock, files processed) for `reanalyze -dce` on representative projects. Store JSON under `analysis/benchmarks/reactive_baseline.json` for later comparison.
4. Stub `--reactive` flag in `analysis/bin/main.ml` (behind experimental guard) pointing to a placeholder that exits with `Not_implemented`. This allows CLI plumbing to be reviewed early.

4.2 Acceptance

- Batch builds/tests pass with Skip runtime linked.
- Baseline metrics captured and committed.

5 Milestone 1: Pure Data Extraction

Why it exists. Skip and any future incremental engine need deterministic, side-effect-free inputs; the legacy analyzer mutates global tables during traversal. Milestone 1 introduces collectors and scoping helpers so every `.cmt` traversal can produce a pure snapshot (`Collected_types.t`) without touching `DeadCommon`. This isolates collection logic from reporting logic, enabling parity checks and—long term—the ability to run the same traversal both for batch mode (via a sink collector) and for Skip (via a pure collector) without diverging behaviour.

5.1 Overview

Goal: isolate AST traversal and data recording from `DeadCommon`'s global tables through incremental steps that keep batch behavior unchanged after each sub-step.

5.2 Step-by-Step Refactor

1. Step 1.1: Define collector types

```
(* Collected_types.ml *)
type optional_arg = { supplied: string list; maybe: string list }

type ref_edge_kind =
| Value
| Type
| Exception
| OptionalArgs of optional_arg

type decl = {
  name: Name.t;
  module_path: Path.t;
  loc: Location.t;
  decl_kind: Common.DeclKind.t;
}

type ref_edge = { from_: Location.t; to_: Location.t; kind: ref_edge_kind }

type file_edge = { from_file: string; to_file: string }

type t = {
  decls: decl list;
  refs: ref_edge list;
  file_edges: file_edge list;
}

type collector = t -> t

let empty = { decls = []; refs = []; file_edges = [] }
```

Document schema and invariants (e.g., loc must be non-ghost).

2. Step 1.2: Create collector interface

```
module type COLLECTOR = sig
  type t
  val empty : t
  val add_decl : t -> decl -> t
  val add_ref : t -> ref_edge -> t
  val add_file_edge : t -> file_edge -> t
end
```

Provide two implementations:

- `Collected_collector` storing data in `Collected_types.t`.
- `DeadCommon_sink` writing into `DeadCommon` by delegating to existing functions. This keeps batch mode working while the refactor proceeds.

3. Step 1.3: Thread collector through DeadValue

- Introduce `DeadValue.collectValueBinding ~collector` and `collectExpr ~collector` that perform the existing logic but call `Collector.add_decl/add_ref` instead of mutating globals directly.

- Example snippet:

```
let collectValueBinding ~collector super self vb =
  let collector = ref collector in
  (* existing pattern matching ... *)
  (match vb.vb_pat.pat_desc with
  | Tpat_var (id, { loc = { loc_start; loc_ghost } as loc }) when not loc_ghost ->
    let decl = { name; module_path; loc = vb.vb_loc; decl_kind = ... } in
    collector := C.add_decl !collector decl
  | _ -> ());
  collector := super.Tast_mapper.value_binding self vb;
  !collector
```

- Build `DeadValue.traverse_structure` `~collector` structure returning the updated collector. The function creates a ref cell for the collector and assigns call-back closures inside `Tast_mapper` to update the ref.
- Repeat for `DeadType`, `DeadException`, and `DeadOptionalArgs`: each module gains a `~collector` parameter. Until the pure pipeline lands, pass `DeadCommon_sink.collector` from `DeadCode.processCmt`.

4. Step 1.4: Common state helpers

- Add `Common.with_current_module` `~src` `~module_name` (`fun () -> ...`) wrapping assignments to `currentSrc/currentModule/currentModuleName`. Both batch and reactive traversals call this helper so state is set consistently.
- Update `Reanalyze.loadCmtFile` to invoke `Common.with_current_module` around `DeadCode.processCmt`.
- Add `ModulePath.with_current` : `(unit -> 'a) -> 'a` resetting `ModulePath.current` during traversal, ensuring collector runs are isolated.

5. Step 1.5: Parity verification

- After each sub-step (value collection, type collection, exceptions), run `reanalyze -dce` on fixtures to ensure diagnostics match.
- Add a local parity script that compares JSON outputs before/after refactor using `git diff --no-index`. Fail if any change appears.

5.3 Acceptance

- `DeadValue`, `DeadType`, `DeadException`, `DeadOptionalArgs` can return `Collected_types.t` without mutating `DeadCommon` when passed `Collected_collector`.
- Batch diagnostics remain unchanged.
- Unit tests cover representative constructs.

6 Milestone 2: Deterministic File Summaries

Why it exists. Collectors give us in-memory snapshots, but Skip needs stable artifacts that can be hashed, cached, and reloaded across runs. Milestone 2 defines a canonical summary schema

(positions, decl metadata, references, file edges), emits digested JSON blobs, and introduces a cache so every .cmt file has a reproducible summary. These summaries become the inputs to the graph store in Milestone 3 and the Skip service in Milestone 4; without them, we would have no persistent boundary between “per-file traversal” and “global liveness.”

6.1 Schema

```
{
  "version": 2,
  "source_file": "src/Foo.res",
  "digest": "b2d7...",
  "decls": [
    {
      "name": "Foo.make",
      "kind": "Value",
      "module_path": ["Foo"],
      "loc": { "line": 12, "column": 4, "cnum": 1234, "bol": 1228 },
      "toplevel": true,
      "optional_args": ["callback"],
      "side_effects": true
    }
  ],
  "refs": [
    {
      "from": { "line": 18, "column": 6 },
      "to": { "line": 12, "column": 4 },
      "kind": "Value"
    },
    {
      "from": { "line": 25, "column": 8 },
      "to": { "line": 25, "column": 8 },
      "kind": "OptionalArgs",
      "optional": { "supplied": ["callback"], "maybe": [] }
    }
  ],
  "file_edges": [ { "from": "src/Foo.res", "to": "src/Bar.res" } ]
}
```

- **Version field** follows semantic versioning (major only). Increment when schema changes; maintain backward compatibility by upgrading readers.
- Store canonical JSON (sorted keys) and compute digest using `Digestif.blake2b` over the canonical bytes.
- Set `Summary.version = 2` (reflecting the absolute-position schema); bump the constant whenever new mandatory fields ship and regenerate cached artifacts.

6.2 Implementation Tasks

1. Implement `Summary.of_collected` covering:

- Conversion of `Collected_types.decl` to summary decls, including flags like `toplevel`, `optional_args`, `side_effects`.
- Flattening `ref_edge_kind` into JSON-friendly shapes.

- Deduplicating `file_edges` by sorting and calling `List.sort_uniq`.
2. Implement `Summary.to_json/from_json` with explicit error messages (`raise Summary.Invalid_format_of_string`). Provide property tests to ensure round-trips succeed.
 3. Create `Summary_cache`:
 - Cache path: `.reanalyze/summaries/<digest>.json`.
 - Write atomically by writing to temp file then `Unix.rename`.
 - Provide `read_or_recompute ~project_root summary` helper.
 4. Update `DeadCode.processCmt`: after obtaining `collected`, call `Summary.of_collected`. Feed the result into `DeadCommon_sink` (for batch) and optionally write to cache when `Common.Cli.cache_summaries` is set. This validates the new path without altering output.
 5. Extend verification by running `make test-analysis` (deadcode fixture project) plus the parity harness to confirm summaries remain stable between runs. Record any mismatches directly in the implementation log instead of relying on synthetic unit tests.

6.3 Acceptance

- Schema documented and versioned.
- Cache read/write validated by tests (including error cases like corrupt JSON).
- Batch diagnostics still match goldens.

7 Milestone 3: Graph Store and Incremental Liveness

This milestone is the first *algorithmic* change to the analysis. Milestone 2 gave us deterministic per-file summaries, but the batch pipeline still rebuilt the entire global graph after every change. Here we:

- assign stable declaration identifiers (file # line # column # kind # name) and persist forward/reverse edges, so we can answer “which declarations does this file affect?” without re-reading `.cmt` files;
- compute a **frontier** starting from the files whose summaries changed and walking reverse edges, so only declarations that might change liveness get reprocessed;
- run Tarjan’s strongly-connected-components (SCC) algorithm on just that frontier, cache each SCC’s hash (members + outgoing edges + summary digests), and re-run liveness only when the cache misses.
- surface an `-incremental-liveness` CLI flag so developers can build the graph, observe the recompute frontier, and keep it in lockstep with batch diagnostics even before the Skip runtime hosts it.

We treat the graph solver as experimental behind the `INCR_GRAPH_SOLVER=1` environment flag: by default the CLI builds/prints the frontier but reuses the batch liveness answers so parity can run on every change. As soon as the SCC hashing + heuristics catch up, flipping the flag enables the Tarjan-based recompute without touching the driver loop.

Tarjan’s algorithm is a linear-time DFS that assigns every node a discovery index, keeps a stack of “live” nodes, and emits each SCC exactly once when it discovers a node whose lowlink equals its index. This property makes it ideal for the dead-code solver: SCCs correspond to mutually recursive bindings, so we can reuse old SCC solutions unless the component structure changes. Without this graph/SCC layer, the Skip runtime would still be forced to re-run the whole liveness fixpoint on every edit, defeating the point of going reactive.

For now the `-incremental-liveness` flag drives a “shadow” path: summaries feed the in-process `Graph_store`, we log the size of the recompute frontier after every batch run, and the parity harness executes the same two-stage flow on curated fixtures. Once the Skip service lands we will replace the logging with actual incremental diagnoses, but keeping the flag in tree de-risks each intermediate refactor.

De-risking strategy. Because this milestone changes core liveness behaviour, we rely on the same safeguards as earlier work, just focused on the new graph layer:

- Every change must keep `make test-analysis` green. Those fixture projects exercise dead values, optional args, exceptions, and termination; running them after each graph-store change ensures batch behaviour stays identical.
- Run the collector parity harness (`dune exec analysis/bin/collector_parity.exe --tests/analysis_tests/tests-reanalyze/deadcode/lib/bs`) after wiring the new graph to confirm the SCC-based pipeline still matches the legacy `DeadCommon` diagnostics.
- Stage 2 of the parity harness now builds the graph, prints the incremental frontier, and compares incremental-vs-legacy deaths. Developers can flip `INCR_GRAPH_SOLVER` or `INCR_AFTER_LEGACY` in that command to experiment with the pure solver without regressing the default behaviour.
- Log intermediate graph/SCC hashes in the implementation log so we can trace regressions.

Milestone 3.1 – Annotation Semantics

Thread `ProcessDeadAnnotations.(annotateLive|annotateDead|doReportDead)` through the graph solver so declarative annotations (React components, `@dead`, etc.) behave exactly like the batch path. Finishing this milestone should remove the current block of “unexpected deaths” involving annotated React props/components.

Milestone 3.2 – Delayed Edge Replay

Ensure `DeadOptionalArgs.forceDelayedItems` and the exception replay run before we query the graph, so optional-arg and ghost exception references populate the edge set just like in batch mode. This should shrink the “missing deaths” set where optional parameters currently appear unused.

Status. Completed.

- Added `DeadException.replay_delayed_items` and `DeadOptionalArgs.replay_delayed_items`; these re-run the delayed-edge logic against any *pure* collector without consuming the shared queues. `Reanalyze.loadCmtFile` now calls them only when a summary/graph run is needed (`-cache-summaries` or `-incremental-liveness`), so batch runs never touch the new code.

- Taught `Collector.collected` to deep-copy the mutable optional-argument payloads before storing `Common.decls`. Summary-side replays can now mark arguments as used without mutating the legacy `DeadCommon` structures coming from the sink collector.
- Removed the previous attempt that flushed `DeadException`/`DeadOptionalArgs` queues at the end of every `DeadCode.processCmt`. The queues now drain once per batch run (as before); the summary path reads them through the replay helpers without disturbing the batch data.

Validation.

- `make test-analysis`
- `dune exec analysis/bin/collector_parity.exe -- tests/analysis_tests/tests-reanalyze/deadcode`
 - Stage 1 (collector parity) remains clean.
 - Stage 2 (`INCR_GRAPH_SOLVER=1`) still reports 115 “unexpected deaths”; those correspond to the remaining sub-milestones (file ordering, reference normalisation, module bookkeeping).

Milestone 3.3 – File Ordering Parity

Carry the legacy `orderedFiles` rank into each node and order SCC processing by that rank, preventing premature deaths when dependencies live in other files. Completing this step should eliminate mismatches caused purely by traversal order.

Status. Completed.

- `Graph_store` now retains the per-file dependency DAG emitted by the collector and recomputes the same topological order that `DeadCommon.iterFilesFromRootsToLeaves` derives. Every time summaries update file edges or digests we invalidate and recompute the rank table so it stays in sync with the legacy ordering.
- The incremental solver fetches those ranks, sorts the frontier before feeding it to Tarjan, and sorts every SCC’s member list with `Decl.compareUsingDependencies`, mirroring the batch traversal (files processed from leaves to roots, then bottom-to-top inside each file).

Validation.

- `make test-analysis`
- `INCR_GRAPH_SOLVER=1 dune exec analysis/bin/collector_parity.exe -- tests/analysis_tests/test`
 - Stage 1 remains clean.
 - Stage 2 still reports exactly 115 unexpected deaths, all tracked for milestones 3.4–3.5. No file-order-related regressions remain.

Milestone 3.4 – Reference Normalisation

Apply the same location rewriting (`Current.lastBinding`, ghost filtering, point locations) before counting incoming references so self-ref edges and ghost locations do not skew the solver. Expect the long tail of prop/record mismatches to drop after this milestone.

Milestone 3.5 – Module Bookkeeping

Call `DeadModules.(markDead|markLive)` from the graph solver so module-level liveness state and derived warnings stay in sync. Once this lands, module-level discrepancies (e.g. nested components) should disappear.

Each mini-milestone targets a single discrepancy category; running the parity harness after each completion should show a strictly decreasing mismatch count. Each step removes a distinct block of discrepancies that the parity harness currently reports, letting us watch the diff set shrink monotonically as we implement the missing behaviour.

7.1 Graph Store Design

1. Data structures

- `decl_id = <source file>##<line>##<column>##<kind>##<name>.`
- Hashtables: `forward_edges : decl_id -> decl_id list, reverse_edges : decl_id -> decl_id list, file_to_decls : string -> decl_id list, decl_info : decl_id -> summary_decl.`
- SCC cache: `decl_to_scc : decl_id -> int, scc_states : int -> { members; live; digest }.`

2. Frontier computation

```
let frontier graph changed_files =
  let seed_decls = List.concat_map (fun file -> Hashtbl.find file_to_decls file) changed_files
  in
  let rec bfs acc queue =
    match queue with
    | [] -> acc
    | id :: rest when DeclSet.mem id acc -> bfs acc rest
    | id :: rest ->
        let acc = DeclSet.add id acc in
        let revs = Hashtbl.find_opt reverse_edges id |> Option.value ~default:[] in
        bfs acc (revs @ rest)
    in
  bfs DeclSet.empty seed_decls |> DeclSet.elements
```

This yields all declarations whose liveness can change due to the modified files.

3. Incremental Tarjan

- Run Tarjan only on the subgraph induced by `frontier`. Non-frontier SCCs keep their cached `scc_state`.
- For each SCC, compute a hash of (members, outgoing edges, summary digests). If unchanged, reuse previous liveness result.
- `Liveness.solve_scc`: reuse logic from `DeadCommon.resolveRecursiveRefs` by parameterizing over callbacks to fetch references.

7.2 Pseudocode

```

let recompute_liveness graph changed_files =
  let frontier = frontier graph changed_files in
  let subgraph = induce graph frontier in
  let sccs = Tarjan.compute subgraph in
  List.iter (fun scc ->
    let key = hash_scc scc in
    match Hashtbl.find_opt graph.scc_cache key with
    | Some old when old.members = scc.members && old.out_edges = scc.out_edges -> ()
    | _ ->
        let result = Liveness.solve_scc scc in
        Hashtbl.replace graph.scc_cache key result;
        emit_diffs scc.members result)

```

7.3 Acceptance

- Frontier computation + SCC recompute documented and implemented.
- Existing integration suites (`make test-analysis`) cover cases where SCCs shrink/grow, optional-arg edges toggle liveness, and exception references propagate; parity harness runs remain green.

8 Milestone 4: Skip Reactive Service Core

Why it exists. Once summaries and the incremental graph exist, we need a long-running process that actually declares them to Skip, ties them to tracked files, and enforces tracker discipline. Milestone 4 is where `reanalyze --reactive` comes to life: it discovers `.cmt/.cmti` files, declares them via `Reactive.input_files`, maps each one to its summary/graph contribution, and ensures diagnostics are only observed after `Reactive.exit`. This milestone proves we can run the analyzer inside Skip’s tracker model without relying on ad-hoc I/O or one-off batch scripts.

8.1 CMT Discovery and Input Declaration

Discovery Phase Skip runtime requires ALL input files to be declared upfront via `Reactive.input_files` (per `docs/reactive_ocaml.tex`, line 91: “Declare the set of input files. Skip records and sorts them; cached runs require the same set”). Ad-hoc file IO inside maps violates tracker discipline (line 171: “ad-hoc I/O violates dependency tracking”).

1. At service startup, discover all `.cmt/.cmti` files under `lib/bs`:

```

(* Cmt_discovery.ml *)
let discover_cmt_files ~project_root =
  let lib_bs = Filename.concat project_root "lib/bs" in
  let rec walk acc dir =
    if Sys.is_directory dir then
      Sys.readdir dir
      |> Array.fold_left (fun acc entry ->
          walk acc (Filename.concat dir entry)) acc
    else if Filename.check_suffix dir ".cmt" || Filename.check_suffix dir ".cmti" then
      dir :: acc
    else acc
  in
  walk [] lib_bs |> Array.of_list

```

2. Discovery happens ONCE at startup before `Reactive.input_files`. If new CMT files appear, service must restart (input file list is fixed per `docs/reactive_ocaml.tex`).
3. For restart coordination, rewatch writes `.reanalyze/build.stamp` with monotonic build ID. Reactive service watches this (outside Skip) and exits on change, triggering supervisor restart.

8.2 Rewatch Integration

1. Add `rewatch/src/build/notify.rs`:

```
pub fn notify_reanalyze(state: &BuildState) -> anyhow::Result<()> {
    let stamp_path = state.project_context.root_dir.join(".reanalyze/build.stamp");
    let build_id = state.build_id.to_string();
    std::fs::create_dir_all(stamp_path.parent().unwrap())?;
    std::fs::write(&stamp_path, build_id)?;
    Ok(())
}
```

2. Invoke after successful build in `rewatch/src/build/compile.rs`:

```
if run_successful {
    if let Err(e) = notify::notify_reanalyze(&command_state.build_state) {
        log::warn!("Failed to notify reanalyze: {e:?}");
    }
}
```

3. Reactive service polls `build.stamp` in background thread; when content changes, exits gracefully (code 0), signaling supervisor restart with fresh CMT discovery.

8.3 Skip Graph Implementation

Tracker Discipline Critical constraint from `docs/reactive_ocaml.tex` (line 171):

“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.”

Implementation:

1. Declare ALL discovered CMT files as inputs:

```
let all_cmt_files = Cmt_discovery.discover_cmt_files ~project_root in
let cmt_inputs = Reactive.input_files all_cmt_files in
```

2. Each CMT becomes a key. Skip invokes map once per key, passing: (1) `key`: CMT path, (2) `trackers`: array with ONE tracker for this file.
3. Read CMT using ONLY the tracker (per `docs/reactive_ocaml.tex` line 93):

```
let cmt_bytes = Reactive.read_file key trackers.(0) in
```

Skip tracks content hash and invalidates when file changes.

```

Reactive Pipeline
(* Reactive_service.ml *)
let run project_root =
  let heap_path = Filename.concat project_root "reanalyze.rheap" in
  Reactive.init heap_path (1024 * 1024 * 1024);

  (* Discover all CMT files upfront *)
  let all_cmt_files = Cmt_discovery.discover_cmt_files ~project_root in
  Log_.info "Tracking %d CMT files" (Array.length all_cmt_files);

  (* Declare ALL as tracked inputs - mandatory per reactive_ocaml.tex *)
  let cmt_inputs = Reactive.input_files all_cmt_files in

  (* Stage 1: Parse CMT -> Summary (one invocation per CMT file) *)
  let summaries =
    Reactive.map cmt_inputs (fun cmt_path trackers ->
      (* MUST use tracker - no ad-hoc IO per reactive_ocaml.tex:171 *)
      let cmt_bytes = Reactive.read_file cmt_path trackers.(0) in
      match Cmt_format.read_cmt_bytes cmt_bytes with
      | exception exn ->
          Diagnostics.report_cmt_error cmt_path exn;
          []
      | cmt_infos ->
          match FindSourceFile.cmt cmt_infos.cmt_annot with
          | None -> Diagnostics.report_no_source cmt_path; []
          | Some source_file ->
              let collected =
                Common.with_current_module ~src:source_file
                  ~module_name:(Paths.getModuleName source_file >| Name.create)
                  (fun () -> ModulePath.with_current (fun () ->
                      Pure_collectors.from_cmt cmt_infos))
              in
              let summary = Summary.of_collected collected in
              Summary_cache.write ~project_root summary;
              [| (source_file, summary) |])
    in

  (* Stage 2: Aggregate to build unified graph *)
  (* Use special "__all__" key to collect summaries from all files *)
  let graph_inputs =
    Reactive.map summaries (fun source_file summary_arr ->
      [| ("__all__", summary_arr) |])
  in

  let graph =
    Reactive.map graph_inputs (fun key summary_arrays ->
      if key <> "__all__" then [] else
      let graph = Graph_store.create () in
      Array.iter (fun arr ->
        Array.iter (fun (src, summary) ->
          Graph_store.add_summary graph ~source_file:src summary
        ) arr
      ) summary_arrays;
      [| (key, graph) |])
  in

  (* Stage 3: Compute liveness *)
  let diagnostics =
    Reactive.map graph (fun key graph_arr ->

```

```

if key <> "__all__" then [] |> else
  let graph = snd graph_arr.(0) in
  let changed = Graph_store.get_dirty_files graph in
  let diff = Liveness.recompute graph changed in
  [| (key, diff) |]
in

Reactive.exit ();
Diagnostics_loop.run diagnostics

```

Key Design Decisions

1. **Per-CMT tracking:** Every CMT gets own tracker. File change re-runs only that key's map invocation.
2. **Aggregation key:** "__all__" collects summaries to build unified graph. Valid pattern per docs/reactive_ocaml.tex line 167 (multi-file fan-out).
3. **Zero ad-hoc IO:** ALL CMT reads use Reactive.read_file with trackers. FindSourceFile.cmt reads from already-loaded cmt_infos (no file IO).
4. **Summary cache:** Written for debugging but NOT used for invalidation. Skip's content hashing provides caching.

8.4 Error Handling

- **Corrupt CMT:** Catch exception from Cmt_format.read_cmt_bytes, emit diagnostic, return empty. Skip caches error; if fixed, hash changes and map re-runs.
- **Missing source:** FindSourceFile.cmt returns None. Emit diagnostic, return empty.
- **Collector failure:** Wrap Pure_collectors.from_cmt in try-catch, convert to diagnostic. Graph treats missing summaries as files with no declarations.
- **Service restart:** Fatal errors exit non-zero. Supervisor restarts service, which rediscovers CMT files.

8.5 Acceptance

- **Tracker discipline verified:** Every CMT read through Reactive.read_file with assigned tracker. Zero ad-hoc IO. Validated by inspecting Skip dependency graph showing all CMT files as tracked resources.
- **Content-based invalidation:** Single .res change (CMT rebuild) causes Skip to re-run ONLY that file's map, then propagate through graph/liveness.
- **Restart protocol:** Adding/removing modules requires service restart via build.stamp detection.
- **Performance:** Warm edits complete in <200ms p50, reusing cached summaries for unchanged files.

9 Milestone 5: Integration (CLI, Watcher, LSP)

Why it exists. A reactive engine isn't useful unless developers can run it the same way they run batch mode. Milestone 5 threads the Skip runtime through the actual user entry points: the CLI flag, watch mode, and the editor integration. It adds the supervisor/watch restart story, wires diagnostics into the existing output formats, and gives LSP clients a near-real-time source of dead-code warnings. In short, it turns the Skip graph into a user-facing feature rather than a behind-the-scenes prototype.

9.1 Build Coordination

- Rewatch writes `.reanalyze/build.stamp` after successful builds (Milestone 4). The reactive service detects this change and restarts to discover new/removed CMT files.
- Per `docs/reactive_ocaml.tex` Section “Process Discipline” (line 183): process stays single-threaded and declares graph once. The service runs continuously, watching `build.stamp` in a background thread (outside Skip). When stamp changes, exits gracefully for supervisor restart.
- Supervisor (systemd/launchd/manual script) automatically restarts the service, which rediscovers CMT files and rebuilds the Skip graph with fresh inputs.

9.2 CLI Mode

1. `analysis/bin/main.ml`:

```
if Common.Cli.reactive then
  Reactive_service.run_forever ()
else
  Reanalyze.cli ()
```

2. `Reactive_service.run_forever`:

- Initialize heap via `Reactive.init`
 - Discover and declare all CMT files via `Reactive.input_files`
 - Declare reactive graph (summaries -*i* graph -*i* diagnostics)
 - Call `Reactive.exit()`
 - Enter monitoring loop:
 - a. Read diagnostics via `Reactive.get_array`
 - b. Print/publish diagnostics
 - c. Sleep/poll `build.stamp` in background thread
 - d. On stamp change, exit gracefully (code 0) for supervisor restart
3. Provide `--reactive-heap=<bytes>` flag. Heap persists across restarts on Linux (per `docs/reactive_ocaml.tex` line 189), deleted on startup on macOS (line 188).
 4. Streaming output: Print diagnostics in same format as batch mode (file, location, message).

9.3 LSP Integration

- Implement `analysis/lsp/Reactive_provider.ml` targeting LSP 3.17. Diagnostics use `textDocument/publishDiagnostics` with payload:

```
{  
  "jsonrpc": "2.0",  
  "method": "textDocument/publishDiagnostics",  
  "params": {  
    "uri": "file:///abs/path/Foo.res",  
    "version": 42,  
    "diagnostics": [  
      {  
        "range": { "start": { "line": 9, "character": 2}, "end": { "line": 9, "character": 5} },  
        "severity": 2,  
        "source": "reanalyze",  
        "message": "foo is never used"  
      }  
    ]  
  }  
}
```

- Maintain `file -> Diagnostic.t list`; when diffs arrive, recompute per-file arrays and emit the JSON payload above. Multiple clients are supported by broadcasting via the existing session manager. Add a 50ms debounce.

9.4 Acceptance

- CLI watch mode mirrors batch output after each build.
- LSP clients receive timely diagnostics with proper versioning.

10 Milestone 6: Validation and Rollout

Why it exists. Even with a working reactive service, we need confidence it matches batch diagnostics, meets latency targets, and can be rolled out safely. Milestone 6 adds the parity harness (batch vs reactive diffs), performance benchmarks, and the staged rollout plan. It's where we prove the new pipeline is both correct and faster for real projects, and where we define the fallback/rollback story so the reactive mode can ship without risking regressions.

10.1 Parity Harness

- Implement `tests/reactive_parity.ml`:

```
let run_batch project =  
  Reanalyze_cli.run ~project ~mode:'Batch'  
let run_reactive project =  
  Reactive_runner.run_once ~project:(project)  
let assert_parity project =  
  let batch = run_batch project in  
  let reactive = run_reactive project in  
  Alcotest.(check diagnostics) (project `parity) batch reactive
```

- Diagnostics comparison normalizes ordering and formatting (sort by file/line, compare message text).
- Run parity harness nightly on: core repo, partner repo A, partner repo B. Record failures in local logs for investigation.

10.2 Performance Benchmarks

- Benchmark script runs both pipelines on 10/100/1000-file fixtures, measuring: manifest read, summary generation, graph recompute, diagnostic emission. Record p50/p95.
- Compare against baseline metrics; fail if p50 or p95 regress more than 10%.

10.3 Rollout

1. **Phase 1:** internal opt-in via VS Code flag.
2. **Phase 2:** enable reactive mode by default for ReScript repo devs; keep batch fallback flag.
3. **Phase 3:** public beta announced in release notes; encourage feedback via GitHub issues. Automatic rollback if parity job fails.

11 Troubleshooting Guide

- **Heap exhaustion:** expose CLI flag to grow heap; log heap usage when exceeding 80%. Document that macOS removes heaps on restart.
- **Manifest corruption:** instruct users to delete `.reanalyze/manifest.json` and rerun `make`. Provide CLI flag `--reanalyze-force-cold` to ignore cache.
- **Tracker violations:** add instrumentation to detect direct file IO during reactive runs (wrap `open_in` and warn if called inside map).
- **Cross-platform linking:** include section in README describing required linker flags and how to install Skip runtime dependencies.

12 Summary

Milestone	Key Deliverables
M0 Readiness	Skip linkage, baseline metrics
M1 Pure collection	Collector interfaces, refactored <code>Dead*</code> , unit tests
M2 Summaries	<code>Summary.ml</code> , cache, goldens
M3 Graph + liveness	<code>Graph_store</code> , frontier + SCC cache
M4 Skip service	Reactive pipeline, manifest writer, error handling
M5 CLI/LSP	<code>--reactive</code> mode, watcher, LSP provider
M6 Rollout	Parity harness, perf benchmarks, troubleshooting

Reactive dead code analysis built on Skip runtime