<migration_spec>

1 executive summary

- current python system is a langgraph state machine for finops anomaly lifecycle: watcher → triager → analyst → hil gate → executor → verifier, with durable checkpoints and a human approval gate.
- migration target: go v1.25+ (use go 1.26 toolchain now; still satisfies “1.25+”). (go.dev)
- core architectural shift: replace langgraph durability/checkpointing with temporal workflows + activities (durable timers, retries, signals/updates for hil), while keeping the same phase semantics and state shape. temporal workflows must be deterministic; all nondeterminism moves to activities (aws api calls, kubecost/http, aws-doctor exec). (Temporal Docs)
- “deterministic-first” redesign (rip out dangerous parts):
    - llm is demoted to narrator/ui helper only; it cannot create/modify executable actions.
    - all triage + root-cause attribution primitives are computed from deterministic data sources (ce/cur/kubecost/etc), then *optionally* summarized.
    - executable plans are produced by a rule/policy engine + typed action templates + allowlists, and require explicit approval routing by risk. (mirrors the existing hil routing semantics.)
- required integrations baked in:
  t-class “evidence producer” activity (json output) for waste + trend signals. (Go Packages)
    - generative ui: ship a web ui that renders dynamic workflow-driven components using ag-ui/copilotkit patterns; optionally embed similar components in mcp-compatible surfaces. (AWS Documentation)

2 go project structure

repo layout (single mono-repo; go backend + optional web ui):

- go.mod (go 1.25+; pin toolchain to 1.26 for now)
- /cmd
    - /worker-finops (temporal worker: workflows + activities)
    - /api (http/grpc api for ui + operators)
    - /cli (operator cli: start workflows, query status, approve, export reports)
    - /mcp-aws-cost (mcp server: cost explorer + ri/sp + credits/marketplace/transfer)

    - /mcp-aws-cur-athena (mcp server: athena-cur line items)
    - /mcp-kubecost (mcp server: kubecost allocation fetcher)
    - /mcp-slack (mcp server: interactive approvals + notifications)
    - /mcp-jira (mcp server: tickets/audit trail)
- /internal
    - /domain
        - models.go (state structs/enums: anomaly, triage, analysis, action, execution, verification)
        - schema/ (jsonschema for action plans + evidence bundles)
    - /policy
        - engine.go (risk routing, allow/deny rules, tag guards, dry-run requirements)
        - rules/ (yaml/rego-ish rules; compiled at startup)
    - /triage
        - primitives.go (ri/sp drift, credits, marketplace, transfer, kubecost shift, deploy correlation)
        - scorer.go (severity + confidence computation)
    - /analysis
        - attribution.go (cur line item grouping, usage-type diffs, tag attribution)
        - planner.go (deterministic action template selection + parameterization)
        - narrator.go (optional llm summarizer; strict i/o contracts)
    - /executor
        - executor.go (plan execution, snapshots, idempotency keys, rollback plan synthesis)
        - snapshots.go (pre/post state capture adapters)
    - /verifier
        - health.go (slo/health checks, cw metrics)
        - savings.go (cost trajectory verification)
    - /connectors
        - /aws
            - ce.go, sts.go, cw.go, tagging.go, budgets.go, athena.go

        - /kubecost
            - client.go
        - /awsdoctor
            - runner.go (exec or embedded runner + json parsing)
    - /temporal
        - workflows/
            - anomaly_lifecycle.go
            - scheduled_detection.go
            - awsdoctor_sweep.go
        - activities/
            - fetch_costs.go, fetch_cur.go, fetch_kubecost.go, run_awsdoctor.go
            - plan_execute.go, verify.go, notify.go, ticket.go
        - codecs/ (payload encryption/compression if needed)
        - versioning/ (workflow version gates)
    - /mcp
        - server.go (common mcp server bootstrap)
        - tools.go (tool registration + typed request/response)
    - /observability
        - otel.go, logging.go, metrics.go
- /web (optional but recommended)
    - nextjs/react ui w/ copilotkit/ag-ui renderer + temporal-backed state panels
- /tests
    - /golden (fixtures ported from python repo: cost_timeseries, cur_line_items, ri/sp coverage, kubecost allocations)
    - workflow_tests.go (temporal test suite)
    - contract_tests_mcp.go

## 3 component migration map

python → go package mapping (from the current spec):

- finops_desk/models.py → internal/domain/models.go
    - pydantic models/enums → go structs + typed enums (string underlying)

    - “finopsstate” shape preserved (workflow_id, tenant_id, anomaly, triage, analysis, approval, executions, verification, economics, control flow).

- finops_desk/policy.py → internal/policy/engine.go
    - risk scoring + routing preserved (low/low_medium/medium/high/critical) -/denylist, tag guards, per-tenant guardrails

- finops_desk/tools.py → internal/connectors/* + cmd/mcp-*
    - langchain @tool stubs tations (go mcp sdk) (GitHub)

- finops_desk/triage.py → internal/triage/*
    - deterministic evidence-first triage stays; llm removed from decision path

- finopnalysis/*
    - deterministic attribution + action planning; optional narrator

- finops_desk/executor.py → internal/executor/*
    - snapshots + dry-run + stop-on-failure + immutable logging preserved

- finops_desk/verifier.py → internal/verifier/*
    - verify: health dominates cost savings; rollback/escalate/monitor outcomes preserved

- finops_desk/graph.py → internal/temporal/wotional edges become temporal workflow branching + child workflows + durable timers, preserving topology.

- lus cmd/api for ui)

4 aws doctor integration plan

what aws-doctor provides (and why it’s valuable here):

- aws-doctor exposes waste + trend/cost comparison style outputs and suppor“output: tablel json”). (Go Packages)

integration points (temporal activities):

- activity: run_awsdoctor_waste(account, region, creds_ref) → awsdoctorWasteReport
    - invoked during triage for “is this just obvious waste?” and during analysis for action candidates.

- activity: run_awsdoctor_trend(account, region, creds_ref) → awsdoctorTrendReport
    - invoked in watcher/detection to enrich anomaly detection with “trend break” signals.

- workflow: awsdoctor_sweep_workflow(tenant, cadence)
    - scheduled workflow that runs waste+trend across accounts/regions and writes normalized findings into the same evidence store used by triage.

interface design (go):

- internal/connectors/awsdoctor/runner.go:
  - `type runner interface { waste(ctx, opts) (wasteReport, error); trend(ctx, opts) (trendReport, error) }`
  - impl a: exec "aws-doctor … --output json" and decode stdout into aws-doctor json structs. (use their json model types where possible) (Go Packages)
  - impl b: embed as go module (compile-time integration) and call internal functions; still treat output as immutable "evidence bundle".

data flow (deterministic-first):

- aws-doctor output is *evidence only*:
  - map findings → `evidence.awsdoctor.*` fields
  - planner converts evidence → candidate actions only via templates + policy (never "free-form").
- every candidate action must be:
  - tied to a concrete resource id/arn
  - validated against allowlists + tag guards
  - produced with a rollback recipe or rejected (mirrors current spec).

5 generative ui implementation

recommended approach (pragmatic + shippable):

- use copilotkit/ag-ui style generative ui on the web frontend:
  - backend emits a typed "ui intent + schema + data" payload (no raw jsx generation in the model).
  - frontend renders dynamic components based on the schema and current temp(AWS Documentation)
- optionally support "mcp ui" surfaces:
  - expose ui schemas via mcp tools so chat surfaces can render interactive controls (approve/deny, pick action params, request more evidence). (GitHub)

which interfaces should be generative (and why):

- anomaly inbox + drilldown: components depend on category/evidence completeness (ri/sp drift panel vs kubecost panel vs marketplace panel).
- approval queue: render risk-based action cards with diffs, snapshots, and explicit

acknowledgements (high risk requires extra confirm).

- action-plan editor: render only allowed params for the chosen action template; enforce validation client-side + server-side.
- verification dashboard: render cost trajectory graphs + health checks; show “rollback now” as a guarded update.

state management:

- source of truth = temporal workflow state (queried via api; mutations via temporal updates).
- ui issues “updates” for synchronous, audited mutations (approve/deny, override thresholds, attach owner/team) vs “signals” for async hints. (Temporal Docs)
- backend maintains a stable ui contract:
  - `ui_schema_version`
  - `components[]` (cards, tables, confirmations)
  - `actions[]` (mapped 1:1 to temporal updates)

6 agentic workflows implementation (temporal)

workflows to implement (mapping 1:1 with current phases):

- anomaly_lifecycle_workflow (the main state machine)
  - steps:
    1. watcher (mostly a “start” step; detection is usually a separate scheduled workflow)
    2. triage activity: compute deterministic category/severity/confidence + evidence bundle
    3. analysis activity: attribution + deterministic action-plan
    4. hil gate: wait for approval via temporal update/signal; enforce timeouts + escalation
    5. executor activity: snapshot → dry-run → execute → log + ticket
    6. verifier activity: timers + health + cost trajectory → close/rollback/escalate/monitor
- scheduled_detection_workflow (cron)
  - every n minutes/hours:
    - fetch cost times anomalies deterministically (z-score/thresholds)

        - start anomaly_lifecycle_workflow per anomaly

- awsdoctor_sweep_workflow (cron)
    - periodic sweep for waste/trend; attach findings to tenants/accounts.

hil implementation (replace langgraph interrupt/checkpoint):

- use temporal updates for approval decisions (auditable + synchronous), and signals for slack callbacks. ([Temporal Docs]())

- use dum "delay then execute"
    - verification "wait 1h then verify"

- determinism constraints:
    - no direct time/random/network in workflow code; use temporal APIs (workflow.now/side effects) or activities. ([Temporal Docs]())

example (illustrative, not complete):

```
func anomalyLifecycleWorkflow(ctx workflow.Context, input
anomalyInput) (domain.finopsState, error) {
  st := domain.newState(input)

  // triage (activities)
  err := workflow.ExecuteActivity(ctx, activities.Triage,
st).Get(ctx, &st)
  if err != nil { return st, err }

  // analysis
  err = workflow.ExecuteActivity(ctx,
activities.AnalyzeAndPlan, st).Get(ctx, &st)
  if err != nil { ret:contentReference[oaicite:34]
{index=34}ait for approval update
  workflow.SetUpdateHandler(ctx, "approve", func(ctx
workflow.Context, req approveReq) (approveResp, error) {
    // validate req + policy; mutate st.approval
deterministically
    return approveResp{Status: st.Approval}, nil
  })

  _ = workflow.Await(ctx, func() bool { return
st.Approval.IsTerminalOrApproved() })

  if !st.Approval.IsApproved() { return st, nil }
  // execute
  err = workflow.ExecuteActivity(ctx,
activities.ExecutePlan, st).Get(ctx, &st)
  if err != nil { return st, err }

  // verify
  err = workflow.ExecuteActivity(ctx, activities.Verify,
st).Get(ctx, &st)
  return st, err
}
```

7 dependencies and third-party libraries (go)

core:

- temporal go sdk (`go.temporal.io/sdk`) + samples for patterns/tests. (Temporal Docs)
- aws sdk for go v2 (cost explorer, sts, athena, cloudwatch, budgets, tagging)
- mcp go sdk (for tool servers) + mcp spec reference. (Model Context Protocol)
- aws-doctor as module dependency (and/or shipped binary), parse json outputs. (Go Packages)

observability:

- opentelemetry go (traces/metrics), structured logging (slog/zap), temporal observability hooks. (Temporal Docs)

validation + config:

- go-playground/validator (struct validation)
- yaml (gopkg.in/yaml.v3) for policy rules
- jsonschema validator for action plan schema

web ui (non-go):

- next.js/react + copilotkit/ag-ui renderer. (AWS Documentation)

8 migration phases (milestones)

phase 0 — parity harness + contracts

- freeze current python fixtures as golden inputs/outputs (triage category/severity/

confidence, planned actions, policy decisions).

- define stable json schemas:
  - evidence bundle schema
  - action plan schema
  - workflow state export schema

phase 1 — domain + deterministic engines in go

- implement internal/domain + internal/policy + internal/triage + internal/analysis (planner only)
- port tests first (golden tests from python fixtures)

phase 2 — temporal backbone

- implement workflows + activities with temporal test suite
- implement approval flow with updates/signals (no slack yet)
- add replay testing gates in ci to catch nondeterminism regressions. (Temporal Community Forum)

phase 3 — real data connectors + mcp servers

- implement mcp servers for aws ce/cur/cw/tagging/budgets + kubecost
- wire activities to call mcp tools (or call aws sdk directly; mcp becomes "external tool boundary" either way)

phase 4 — aws-doctor integration

- ship runner (binary + json) and activity wrapper
- incorporate aws-doctor findings into evidence ordering (triage) and action candidate set (analysis)

phase 5 — generative ui

- ship api endpoints for:
  - list workflows/anomalies
  - query state + evidence
  - submit approval/update
  - fetch ui schema

- build ui that renders cards dynamically per category/evidence completeness

phase 6 — production hardening + cutover

- task-queue partitioning per tenant / environment
- rate limits + budgets for expensive activities
- authn/z (oidc) + per-tenant iam assume-role
- shadow-run: run go in parallel, compare outcomes, then flip to go as source of truth

9 testing strategy

unit tests (pure deterministic):

- triage primitives:
  - ri/sp coverage drift classification (ce getreservationcoverage/ getsavingsplanscoverage). (GitHub)
  - credits/refunds/fees attribution (ce record types + cur line items). (GitHub)
  - marketplace + data transfer attribution (ce group by service/usage type; cur fields). (GitHub)
  - kubecost cost shift attribution via allocation api. (CopilotKit)

workflow tests (temporal):

- use temporal go sdk test suite:
  - approval update handling
  - retries/backoff
  - durable timer paths (low_medium delay, verify delay)
  - rollback path

contract tests (mcp tools):

- for each mcp server:
  - validate request/response json against schema
  - run against local fixtures (no aws creds needed)

integration tests (optional):

- sandbox aws account:
  - read-only cost explorer/cur

    - “dry-run only” executor with tag guards

security/regression:

- determinism checks: workflow replay tests in ci to prevent nondeterministic code changes. (Temporal Community Forum)
- policy fuzz: generate random action plans; ensure policy engine blocks forbidden/unsafe combos

10 deployment and devops considerations

temporal:

- run temporal self-hosted via docker compose for dev; temporal cloud for prod if you want less yak shaving.
- separate task queues:
    - finops-detect (read-heavy)
    - finops-anomaly (stateful lifecycle)
    - finops-exec (restricted permissions; tighter concurrency)
- workflow versioning strategy:
    - use temporal version gates for backward-compatible changes; never break in-flight histories. (Temporal Community Forum)

aws + secrets:

- per-tenant iam assume-role (sts) with explicit permission boundaries:
    - detection/triage: cost explorer + athena read + cw read + tagging read
    - executor: narrow write permissions by action type; default deny
- store creds refs in temporal state, never raw secrets (use kms/asm/ssm)

observability:

- otel traces across api → temporal client → worker activities; emit:
    - anomaly counts by category/severity
    - approval latency
    - realized savings vs predicted
    - agent-compute unit economics (ported from python economic tracker idea).
- audit trail:
    - immutable event log per action (pre-snapshot, dry-run result, execution, post-

snapshot, verification), plus ticket links.

notes on “missing finops primitives” (to stop hallucinations)

- implement these as first-class evidence collectors (no llm):
    - ri coverage + utilization (cost explorer apis) (GitHub)
    - savings plans coverage + utilization (cost explorer apis) (CopilotKit)
    - credits/refunds/fees (ce record types + cur line item types) (GitHub)
    - marketplace charges (ce service grouping + cur product fields) (GitHub)
    - data transfer (ce usage-type grouping; “datatransfer” usage patterns) (GitHub)
    - kubecost allocation diffs (namespace/workload cost shift) (CopilotKit)
    - aws-doctor waste/trend as an additional evidence stream (json) (Go Packages)

</migration_spec>