

HALEY WORKFLOW ENGINE

CONCEPTS & IMPLEMENTATION

GUIDELINES



12TH January 2026

V0.4

Contents

1.	General coding instructions (<i>non-negotiable</i>)	1
1.1.	SQL + DAL discipline	1
1.2.	Formatting rules	1
1.3.	Transactions + cancellation (DbExecutionLoad).....	1
1.4.	“Ensure/Exists then Insert” rule	2
1.5.	Naming conventions (engine project)	2
2.	Concept (high level).....	3
	Objective (what this engine is)	3
3.	Core runtime flow (application → engine → application).....	4
3.1.	Trigger (application → engine)	4
3.2.	Engine evaluates + persists (engine → DB).....	4
3.3.	Emit outbound events (engine → application)	5
3.4.	Ack updates (application → engine)	5
3.5.	Reliability: Monitor (the heart of idempotency)	6
4.	Examples (end-to-end)	8
4.1.	Example A — Normal transition with ACK	8
4.2.	Example B — Pending ACK retry (30 seconds)	8
4.3.	Example C — Delivered but not processed reminder (5 minutes).....	9
4.4.	Example D — Stale state timeout automation	9
5.	Major engine components and roles	10
5.1.	WorkFlowDAL (DB gateway).....	10
5.2.	BlueprintImporter (IBlueprintImporter)	10
5.3.	BlueprintManager (IBlueprintManager).....	10
5.4.	StateMachine (IStateMachine).....	10
5.5.	PolicyEnforcer (IPolicyEnforcer)	11
5.6.	AckManager (IAckManager)	11
5.7.	LifeCycleMonitor (ILifeCycleMonitor) — scheduler	11
5.8.	WorkFlowEngine (IWorkFlowEngine, also implements ILifeCycleEngine)	12

5.9.	RuntimeEngine (IRuntimeEngine).....	12
6.	Consumer Choreography (Principles)	13
6.1.	Saga Choreography.....	13
6.2.	No Instance Owner	13
6.3.	“Transition Event” is a Fact, not a Command	13
6.4.	Duplicate/Conflicting Transitions are Prevented by the Engine	13
6.5.	Every Consumer Gets a Dispatch Record; Only Alive Consumers Get Notified	13
6.6.	Consumer Heartbeat	14
6.7.	Retry Attempts Are Not Burned While Consumer Is Down.....	14
6.8.	Per-Consumer Per-Instance Ordering Rule	15
6.9.	“At-Least-Once” Events	15
6.10.	State Moves Only When a Consumer Triggers the Next Event.....	15
6.11.	Each Consumer Maintains Its Own Scope.....	16
6.12.	Summary.....	16

1. General coding instructions (*non-negotiable*)

1.1. SQL + DAL discipline

- **SQL queries are first-class citizens:** every DB operation must be a named query constant in the relevant QRY_* class.
- WorkFlowDAL is the only DB entry point for the engine.
 - ✓ Engine/components **must not** call leaf DALs directly (read/write DALs are internal implementation details behind IWorkFlowDAL).
- Adding new DB capability:
 1. Add query constant to the relevant QRY_* class
 2. Extend the relevant DAL(s)
 3. Expose via IWorkFlowDAL

1.2. Formatting rules

- **Models:** keep constructors minimal; prefer object initializer style:
 - ✓ **Preferred:** new X { A = a, B = b }
 - ✓ **Not Preferred:** “everything in constructor”
- Method signatures and constructors must be in one line (no multi-line wrapping).
- SQL query strings must be in one line.

1.3. Transactions + cancellation (DbExecutionLoad)

- DbExecutionLoad is just: (CancellationToken + TransactionHandler).
- Public engine APIs accept **CancellationToken**.
- Internal DB methods accept **DbExecutionLoad**.
- Transaction pattern (engine-owned):

```
var transaction = _dal.CreateNewTransaction();  
  
using var tx = transaction.Begin(false);  
  
var load = new DbExecutionLoad(ct, transaction);
```

- ✓ Commit/Rollback guarded by a committed flag.

1.4. “Ensure/Exists then Insert” rule

- For “upsert” style operations, prefer:
 - ✓ EXISTS → return id
 - ✓ else INSERT → return id
- **Avoid the pattern:** `id = LAST_INSERT_ID(id)` / mutating primary key semantics.

1.5. Naming conventions (engine project)

- Flag enums are prefixed: LifeCycleStateFlag, LifeCycleTransitionFlag, LifeCycleInstanceFlag, LifeCycleTransitionLogFlag, etc.
- DAL group classes are internal sealed (except the few agreed public ones like WorkflowDAL / DALUtil).
- Query placeholders when performing join queries: use the newer placeholders; Example: Joining two tables, where both table has ‘name’ field. In such case, use unique identifier for the field.
 - ✓ ENV_NAME, DEF_NAME (instead of reusing DISPLAY_NAME/NAME where they reduce readability)

2. Concept (high level)

Objective (what this engine is)

A dumb, reliable, DB-backed workflow/state engine that:

- Transitions an instance through states based on triggers (macro workflow). Triggers are nothing by event-contracts raised from the application. Applications doesn't know anything about transition or states. It has awareness only about the events.
- Emits outbound events for:
 - ✓ lifecycle transitions (macro)
 - Application has to decide which internal method to raise when a lifecycle transition is raised.
 - App should also take responsibility of sending the acknowledgements back to the engine.
 - Events to internal methods are **hard-coded** inside the application.
 - ✓ hooks/work-items (micro orchestration)
 - Directly tells the application which internal method to raise.
 - Benefit is that the orchestration can be controlled via json configs.
 - **No hard-coding** of events to internal methods.
- Tracks delivery/processing via ACKs and re-sends until completed
- Automatically triggers timeout events for stale instances
- Produces timeline/reporting; runtime tables are mainly for app activity/status storage

Non-goal: business logic. Business meaning lives in the application + JSON policies/routes, not inside the engine.

3. Core runtime flow (*application → engine → application*)

3.1. Trigger (application → engine)

Application calls:

- `TriggerAsync(LifeCycleTriggerRequest req, ct)`

Minimum req fields:

- `env_code, def_name`; *to find the target definition version*
- `external_ref` (application correlation key); *important as this is how both app and engine correlates*
- `event` (name or code); *to initiate the transition*
- `request_id` (recommended for idempotency);
- optional: `actor, payload`; *actor is for timeline tracking & payload is for carryforward to next stage*
- optional: `ack required` (if you keep this as a request flag); *default true, so we get ack*

3.2. Engine evaluates + persists (engine → DB)

Engine does, inside `one transaction`:

1. Load blueprint (`IBlueprintManager`) — cached read.
2. Ensure instance exists (`IStateMachine.EnsureInstanceAsync`)
3. Apply transition (`IStateMachine.ApplyTransitionAsync`)
 - a. validates transition exists for (`from_state, event`)
 - b. updates instance current state using CAS (*Compare And Set; basically Upsert*)
 - c. inserts lifecycle timeline (`lifecycle, lifecycle_data`)
4. Resolve policy (`IPolicyEnforcer.ResolvePolicyAsync`)
 - a. optionally attach policy to instance
5. Emit hooks (`IPolicyEnforcer.EmitHooksAsync`)
 - a. inserts hook rows (work items) Note: “*Emit hooks*” = *derive hook work-items from policy/routes and persist them, so they can be dispatched reliably (with ACK tracking), same as lifecycle events.*

- b. Prepare Hook Emissions in memory. (Emit hooks doesn't raise events immediately. Not inside the transaction.)
6. Create ACK records (**IAckManager**)
 - a. lifecycle ack (linked to lifecycle_id)
 - b. hook ack (linked to hook_id)
 - c. consumer tracking rows per consumer

Commit first.

Only after commit, the engine raises C# events.

3.3. Emit outbound events (engine → application)

The engine exposes **two C# events** only:

1. **EventRaised**(**ILifeCycleEvent**) — actionable
 - a. used for both lifecycle transition events and hook events
2. **NoticeRaised**(**LifeCycleNotice**) — informational
 - a. retries, failures, duplicates, stale warnings, etc.

Base event requirement (mandatory):

All actionable events (transition + hook) derive from one base event and MUST include:

- external_ref
- ack_guid
- plus event information (transition/hook details)

3.4. Ack updates (application → engine)

Application calls:

- **AckAsync**(consumerId, ackGuid, outcome, ...)

ACK stages: (AckOutcome)

- **Delivered**: “I received the event” (fast); expected *in less than 30 seconds*.
- **Processed**: “I finished processing the event” (slow); expected *in less than 5 minutes*.
- Failed/Retry can exist as outcomes, but core persistence today is status + retry_count/last_retry.

AckOutcome vs AckStatus

- **AckOutcome** is what the application reports to the Engine. *i.o.w; Input from Application*
- **AckStatus** is what the database stores as a state of an ack record. *i.o.w; Persisted state*

Mapping:

- Outcome.Delivered → Status.Delivered
- Outcome.Processed → Status.Processed
- Outcome.Failed → Status.Failed
- Outcome.Retry → Status.Pending + MarkRetry(...) (increment retry_count/last_retry)

3.5. Reliability: Monitor (the heart of idempotency)

Monitor runs periodically and does **two major jobs**:

A) ACK re-dispatch (reliable delivery)

Continuously queries ack dispatch views and re-raises events:

1. Pending too long (delivery not confirmed)

- If an ack is Pending for more than **30 seconds**:
 - ✓ re-raise the same event again (same ack_guid, same payload)
 - ✓ mark retry (retry_count, last_retry)
 - ✓ raise a notice containing ackGuid + instance id + consumer id + kind

2. Delivered but not processed too long

- If an ack is Delivered but not Processed for more than **5 minutes**:
 - ✓ re-raise/remind (same event, same ack_guid)
 - ✓ mark retry
 - ✓ raise notice (ackGuid + instance id + consumer id + kind)

These timings are captured in **WorkFlowEngineOptions**.

B) Stale state automation (timeout events)

Monitor also scans instances:

- If an instance remains in a state beyond **its configured timeout_minutes**:

- ✓ raise notice: stale instance detected
- ✓ if blueprint defines timeout_event for that state:
 - auto-trigger that event as system-triggered (same pipeline as normal trigger)

4. Examples (end-to-end)

4.1. Example A — Normal transition with ACK

1. App triggers

- ✓ Env: 1, Def: "VendorPreQualification"
- ✓ ExternalRef: "VENDOR-00042"
- ✓ Event: "Submit"
- ✓ RequestId: "req-2026-01-04-0001"
- ✓ Payload: { ... }

2. Engine commits DB

- ✓ Ensures instance (def_version_id, external_ref)
- ✓ Current state moves: Draft → Submitted
- ✓ Inserts lifecycle row + lifecycle_data
- ✓ Creates one lifecycle ack and consumer rows (for resolved consumers)

3. Engine raises transition event

Application receives ILifeCycleEvent with:

- ✓ external_ref = "VENDOR-00042"
- ✓ ack_guid = "..." critical
- ✓ transition details: from_state, to_state, event, lifecycle_id, occurred_at
- ✓ optional policy info

4. App acks Delivered

- ✓ AckAsync(consumerId, ackGuid, Delivered)

5. App finishes work and acks Processed

- ✓ AckAsync(consumerId, ackGuid, Processed)

4.2. Example B — Pending ACK retry (30 seconds)

1. Engine dispatched event at T0 but app didn't ack Delivered.

2. At T0 + 30s, monitor finds:
 - ✓ status = Pending
 - ✓ last_retry older than threshold
3. Monitor re-raises the same event (same ack_guid)
4. Monitor raises notice:
 - ✓ Kind = AckRetryPending
 - ✓ includes: ack_guid, instance_id, external_ref, consumer_id, retry_count

4.3. Example C — Delivered but not processed reminder (5 minutes)

1. App acks Delivered quickly, but processing takes too long.
2. At Delivered + 5m, monitor re-raises event as a reminder.
3. Notice raised:
 - ✓ Kind = AckReminderProcessedPending
 - ✓ includes same fields + age

4.4. Example D — Stale state timeout automation

1. Instance enters state Review with:
 - ✓ timeout_minutes = 60
 - ✓ timeout_event = "AutoReject"
2. Monitor detects instance still in Review after > 60 minutes.
3. Monitor raises notice:
 - ✓ Kind = StateStale
 - ✓ includes: instance_id, external_ref, state, age, timeout_event
4. Monitor triggers system event "AutoReject" through the same transactional trigger pipeline.

5. Major engine components and roles

5.1. WorkFlowDAL (DB gateway)

Role: Single entry point to DB for all engine operations.

- Consolidates all read/write DALs
- Enforces query-first discipline (QRY_*)
- Exposes CreateNewTransaction() for engine-owned transactions

5.2. BlueprintImporter (IBlueprintImporter)

Role: Import-time canonicalization (JSON → SQL).

- Loads definition JSON: states/events/transitions/categories/timeouts
- Loads policy JSON: routes/hooks/emit rules
- Writes normalized SQL rows:
 - ✓ environment, definition, def_version
 - ✓ state, events, transition
 - ✓ policy, def_policy link (and any route tables you maintain)
- Should run entire import in one transaction.

5.3. BlueprintManager (IBlueprintManager)

Role: Runtime blueprint read + cache.

- Builds LifeCycleBlueprint from DB (states/events/transitions lookup maps)
- Caches latest by (env_code, def_name) and by def_version_id
- Supports invalidation after imports
- Important cache rule: do not “poison” cache with a canceled task (avoid capturing ct inside cached lazy).

5.4. StateMachine (IStateMachine)

Role: Pure state transition executor (dumb by design).

- Ensures instance exists for (def_version_id, external_ref)
- Resolves event (by code or name)
- Validates transition exists
- Updates instance state using CAS to avoid race conflicts
- Writes lifecycle log/timeline (lifecycle, lifecycle_data)
- Returns ApplyTransitionResult (Applied/Reason/From/To/Event/LifeCycleId)

5.5. PolicyEnforcer (IPolicyEnforcer)

Role: Routes/policy evaluation + hook emission.

- Resolves policy for a given state after transition
- Emits hooks/work-items when policy says so:
 - ✓ creates hook rows in DB
 - ✓ prepares hook emission objects with metadata/payload
- Does not implement business logic; only reads policy JSON and translates to DB + emitted events.

5.6. AckManager (IAckManager)

Role: ACK persistence + consumer tracking + monitor-ready dispatch lists.

- Creates ack records and links:
 - ✓ lc_ack (lifecycle_id → ack_id)
 - ✓ hook_ack (hook_id → ack_id)
- Creates consumer status rows (ack_consumer)
- Updates statuses on AckAsync calls from the app
- Provides monitor methods to list “pending dispatch” items from ack dispatch queries

Critical rule: ACK identifiers are stable:

- re-dispatch always uses the same ack_guid for the same lifecycle/hook item.

5.7. LifeCycleMonitor (ILifeCycleMonitor) — scheduler

Role: Timer + safety gate.

- Runs on a periodic interval (PeriodicTimer)
- Guarantees single active execution (Interlocked gate)
- Calls `WorkFlowEngine.RunMonitorOnceAsync(ct)` each tick; (*Very crucial concept. Monitor itself is generic and doesn't have any business logic associated. The logic lies with in the workflow engine itself*)
- Handles monitor exceptions (typically by raising a notice)

5.8. WorkFlowEngine (IWorkFlowEngine, also implements ILifeCycleEngine)

Role: The orchestrator + API surface.

- Owns all core components:
 - ✓ BlueprintManager, Importer, StateMachine, PolicyEnforcer, AckManager, RuntimeEngine, Monitor
- Exposes the public operations:
 - ✓ TriggerAsync, AckAsync, cache invalidation, and monitor run
- Raises only two C# events:
 - ✓ actionable events (EventRaised)
 - ✓ informational notices (NoticeRaised)
- Owns transactions for Trigger:
 - ✓ **DB changes in transaction**
 - ✓ **commit**
 - ✓ **then raise outbound events** (failures become notices; monitor re-sends)

5.9. RuntimeEngine (IRuntimeEngine)

Role: Observability + app activity/status storage (**not “smart workflow”**).

- Allows application to store activity/runtime rows:
 - ✓ statuses, frozen, lc_id linkage, payload/data
- Supports timeline/reporting
- Does not participate in workflow correctness except for reporting.

6. Consumer Choreography (Principles)

6.1. Saga Choreography

A decentralized, event-driven process for managing distributed transactions where each microservice (or consumers) react independently and make the state transition of a target instance after completing their work while the other microservices will remain as observers.

6.2. No Instance Owner

An instance is not owned by one application. Instead, each application owns only the steps it is responsible for.

- Any consumer may listen to state transitions.
- Only the consumer(s) responsible for a specific next step should attempt the next transition.

6.3. “Transition Event” is a Fact, not a Command

A transition event means:

“The workflow moved from State A → State B via Event X”

All consumers can receive this notification and decide what to do:

- Some consumers may perform side effects (audit, dashboard updates, enrichment, etc.).
- Some consumers may perform business work and then trigger the next transition (only if it is within their scope).

6.4. Duplicate/Conflicting Transitions are Prevented by the Engine

Multiple consumers may receive the same event, but the engine prevents chaos through:

- CAS (compare-and-swap) update on `instance.current_state`
- Transition validation against the definition

If a consumer triggers an event that no longer matches the current state, the transition will not apply (treated as “already moved / not applicable”).

6.5. Every Consumer Gets a Dispatch Record; Only Alive Consumers Get Notified

For every dispatchable lifecycle/hook event, the engine creates:

- ack
- ack_consumer rows for the logical consumer(s)

Dispatch behavior:

- If a consumer is **alive** (heartbeat fresh): it is notified and can process/ack.
- If a consumer is **down** (heartbeat stale): it is **not notified**, but its ack_consumer row remains pending.

When the consumer comes back online, the system can deliver the missed events. This supports “store-and-forward” without losing events.

6.6. Consumer Heartbeat

Each application registers itself and updates heartbeat periodically:

- Table: `lcstate.consumer`(env, consumer_guid, last_beat)
- A consumer is considered **alive** if `now - last_beat <= TTL` (e.g., TTL = 3 × expected beat interval)

Heartbeat is used only to decide **when to dispatch**, not to decide workflow correctness.

6.7. Retry Attempts Are Not Burned While Consumer Is Down

When the monitor is about to dispatch/resend:

- check heartbeat
- if consumer is down:
 - ✓ **do not send**
 - ✓ **do not increment trigger_count**
 - ✓ push next_due forward (e.g., recheck in 60s)

This ensures:

- dead services don't burn max retries
- retries represent actual delivery attempts

When max attempts is exceeded (*while the consumer was alive and dispatch was attempted*):

- set ack_consumer.status = Failed
- set ack_consumer.next_due = NULL

6.8. Per-Consumer Per-Instance Ordering Rule

When a consumer is catching up after downtime, events must be sent **in the same order they occurred** for that instance.

- Recommended ordering key: `lifecycle.id` (*monotonic per instance progression*)
- Monitor should dispatch backlog in ascending order per (*consumer, instance*).
- This keeps consumer replay deterministic and avoids “future event arrives before past event”.

6.9. “At-Least-Once” Events

Consumers must treat events as “At-Least-Once” and Potentially Stale

Even with ordered delivery, a consumer may process slowly and later discover the instance has progressed further. Therefore, consumers must follow a simple rule:

Before performing any action that could trigger a state change, the consumer must re-check current instance state (or timeline) and confirm the work is still applicable.

Practical options:

- read current instance.current_state + last_event
- or call GetTimelineJson(instanceGuid) and reconcile

If the instance already moved past the expected point:

- consumer should ack the event as Processed (or Delivered) and **skip side effects that are no longer valid.**

This prevents late consumers from “acting as if time froze” while they were down.

6.10. State Moves Only When a Consumer Triggers the Next Event

The **engine does not magically advance state**. A transition occurs only when some consumer calls TriggerAsync(...).

If a required business action belongs to Consumer B and Consumer B is down:

- state does **not** progress because the triggering event is never produced
- other consumers may still receive the transition notification but will not attempt transitions outside their scope. This is intended behavior and naturally enforces dependency without needing complex central ownership rules.

6.11. Each Consumer Maintains Its Own Scope

Each consumer defines (internally) which steps it is responsible for, such as:

- “When entering State 2 via Event 1001, I perform action X and then trigger Event 1002”
- “When entering State 3, I only log and update dashboard”

Consumers that are not responsible for the next step:

- do not trigger transitions
- may still perform observer actions

This keeps the engine generic and avoids embedding “who owns what” into the workflow definition itself.

6.12. Summary

- All consumers can receive transition notifications.
- Only responsible consumers trigger the next transition.
- Engine prevents duplication via CAS + transition validation.
- Down consumers are not notified but retain backlog records.
- Catch-up delivery is ordered, but consumers must reconcile against current state/timeline before acting.