HALEY WORKFLOW ENGINE

CONCEPTS & IMPLEMENTATION

GUIDELINES

12TH February 2026

V0.6

Contents

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
- Produces timeline/reporting; runtime tables are mainly for app activity/status storage
- Auto-trigger happens only when `timeout_event` is defined for the state.
- If a state is stale **without** `timeout_event`, the engine **does not auto-transition**, it only raises a notice.

`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. *Done outside transaction.*
2. **Resolve consumers** — At least one transition consumer must exist. Hook consumers are optional.
3. **Resolve policy** (`IPolicyEnforcer.ResolvePolicyAsync`) — gets the latest policy for the definition.
4. Ensure instance exists (`IStateMachine.EnsureInstanceAsync`) — creates or retrieves the instance. Policy is attached at instance creation time and never changes for that instance, even if the policy is updated later.
5. Apply transition (`IStateMachine.ApplyTransitionAsync`)
    a. validates transition exists for (from_state, event)
    b. updates instance current state using CAS *(**C**ompare **A**nd **S**et; basically Upsert)*
    c. inserts lifecycle timeline (lifecycle, lifecycle_data)

d. If transition not applied (e.g. already moved), commits and returns early — instance is still ensured.

6. **Resolve rule context from policy JSON** — Looks up the policy rule matching (to_state, via_event) and extracts: params, on_success_event, on_failure_event.
7. Create lifecycle ACK (`IAckManager.CreateLifecycleAckAsync`)
    a. One ack_guid, multiple consumer rows
    b. Linked via `lc_ack` (lifecycle_id -> ack_id)
8. 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. Important→ not dispatched yet.)
9. Create hook ACKs (`IAckManager.CreateHookAckAsync`)
    a. One ack_guid per hook, multiple consumer rows
    b. Linked via `hook_ack` (hook_id -> ack_id)
10. `Commit first.` Only after commit, the engine raises C# events via FireEvent. Failures during dispatch become notices; the monitor will re-send due to pending ACK rows.

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
    b. Handlers are invoked via fire-and-forget (`RunHandlerSafeAsync`). Failures are caught and raised as *EVENT_HANDLER_ERROR* notices.
2. `NoticeRaised`(`LifeCycleNotice`) — informational
    a. Retries, failures, duplicates, stale warnings, suspensions, etc.
    b. Notice errors are swallowed to prevent infinite loops.

Notice codes emitted by the engine:

Notice codes emitted by the engine:		
Code	**Kind**	**Description**
ACK_RETRY	Warn	ACK pending/delivered too long, event re-raised
ACK_SUSPEND	Warn	Max retries exceeded, instance suspended
ACK_FAIL	Warn	Max retries exceeded but instance not found
STATE_STALE / DEFAULT_STATE_STALE	OverDue	Instance stuck in state with no open ACKs and no policy timeout
MONITOR_ERROR	Error	Unhandled exception in monitor loop
EVENT_HANDLER_ERROR	Error	Exception in an EventRaised handler
TRIGGER_ERROR	Error	Exception during TriggerAsync

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, message?, retryAt?) OR
- `AckAsync`(**envCode**, consumerGuid, ackGuid, outcome, …) *which resolves consumerId internally.*

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**:

The monitor (`LifeCycleMonitor`) runs on a `PeriodicTimer` with a configurable interval. It guarantees single active execution via an Interlocked gate. Each tick calls `RunMonitorOnceInternalAsync`, which performs three jobs:

`A) Default staleness "OverDue" (no timeout policy) – runs first`

- Scans instances that have remained in the same state longer than `WorkFlowEngineOptions.DefaultStateStaleDuration.`
- Only applies to states where no policy timeout is configured (`timeout_minutes` IS NULL OR `timeout_event` IS NULL).
- **Critical filter:** Only raises `OverDue` when there are no open ACKs — all relevant lifecycle and hook ACKs for the current state are already Processed. If anything is Pending/Delivered, the ACK monitor already covers it.
- Excludes instances flagged as Suspended, Completed, Failed, or Archived.
- Fires `LifeCycleNotice` of kind `OverDue` with code DEFAULT_STATE_STALE per consumer, including: *consumerId, instanceId, instanceGuid, externalRef, defVersionId, currentStateId, stateName, lcId, staleSeconds.*
- No DB writes. No ACK created for the notice.
- In-memory throttling via ConcurrentDictionary<string, DateTimeOffset> keyed by `"{consumerId}:{instanceId}:{stateId}"` — prevents spamming the same notice every tick. Throttle interval equals `DefaultStateStaleDuration`. Dictionary has a safety cap of 200K entries (cleared if exceeded).

`B) ACK re-dispatch (reliable delivery) – runs per consumer`

For each registered monitor consumer, scans both Pending and Delivered ACK statuses:

`Step 1: Push next_due for down consumers`

Before scanning dispatch items, the engine calls `PushNextDueForDownAsync`. This pushes next_due forward for consumers whose heartbeat is stale (older than *ConsumerTtlSeconds*), without incrementing trigger_count. This ensures retry attempts are not burned while a consumer is down.

`Step 2: Scan due dispatch items (lifecycle + hooks)`

For each due ACK (where `next_due <= now`):

- If TriggerCount >= MaxRetryCount:
- Mark ACK as Failed (next_due = NULL)
- Look up the instance by guid
- **If found**: Suspend the instance (set LifeCycleInstanceFlag.Suspended flag + message) and fire ACK_SUSPEND notice
- **If not found**: fire ACK_FAIL notice
- Otherwise:
  - ✓ Increment trigger count, set next next_due
  - ✓ Fire ACK_RETRY notice
  - ✓ Re-raise the same event (same ack_guid, same payload, policy context re-resolved from stored policy JSON)

`Escalation path summary`: Pending -> retry -> retry -> ... -> Failed ACK -> Suspend Instance -> ACK_SUSPEND notice.

Timings (from WorkFlowEngineOptions):		
Setting	**Default**	**Purpose**
AckPendingResendAfter	40 seconds	next_due for Pending status
AckDeliveredResendAfter	4 minutes	next_due for Delivered status
MaxRetryCount	configurable	Max attempts before failure/suspension
ConsumerTtlSeconds	30	Heartbeat freshness threshold
ConsumerDownRecheckSeconds	configurable	How far to push next_due for down consumers
MonitorPageSize	200	Paging size for dispatch queries

C) Policy-defined timeout automation

The engine stores timeout configuration in the timeouts table, imported from the policy JSON timeouts array (see Section 7). Each timeout entry specifies:

- state_name: which state this timeout applies to
- duration: timeout duration in minutes (parsed from ISO 8601 duration like "P2D" -> 2880 minutes, or direct timeout_minutes)
- mode: 0 = once, 1 = repeat
- event_code: the event to auto-trigger when the timeout fires

The QRY_LC_TIMEOUT.LIST_DUE_PAGED query finds instances where:

- The instance has been in the target state longer than duration minutes
- The instance is not Suspended/Completed/Failed/Archived
- The lc_timeout tracking table has no record for this lifecycle entry (mode=0 prevents re-fire)

When a timeout is due, the engine auto-triggers the specified `timeout_event` through the same transactional trigger pipeline as a normal application trigger. The `lc_timeout` table records that the timeout was processed (via INSERT IGNORE), ensuring idempotency.

4. Major engine components and roles

4.1. 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

4.2. 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 (hashed for deduplication), def_policy link
  - ✓ timeouts (state_name, duration, mode, event_code per policy)Should run entire import in one transaction.
- **Policy hash**: content is hashed (SHA-256 GUID) for deduplication. Only relevant fields are included in the hash material.
- **Policy timeouts**: parsed from ISO 8601 duration ("P2D", "PT30M") or direct timeout_minutes. Mode: 0=once, 1=repeat.

4.3. 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).
- Consumer management: `EnsureConsumerIdAsync`, `BeatConsumerAsync` for heartbeat

4.4. 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)

4.5. PolicyEnforcer (IPolicyEnforcer)

Role: Routes/policy evaluation + hook emission.

- `ResolvePolicyAsync`(definitionId): Gets the latest policy for a definition.
- `ResolvePolicyByIdAsync`(policyId): Gets a specific policy by ID (used for instance-attached policies).
- `ResolveRuleContextFromJson`(policyJson, toState, viaEvent): Reads policy rules and returns RuleContext containing params, *on_success_event, on_failure_event* for the matched rule.
- `ResolveHookContextFromJson`(policyJson, toState, viaEvent, hookCode): Returns hook-specific context with params, on_success_event, on_failure_event, not_before, deadline.
- `EmitHooksAsync`(blueprint, instance, transition): Reads policy rules, matches by (state, via event), creates hook rows in DB, and returns LifeCycleHookEmission objects with: HookId, HookCode, OnEntry, OnSuccessEvent, OnFailureEvent, NotBefore, Deadline, Params.
- **Param resolution:** Policy params array defines a catalog of named parameter sets (code + data). Rules and emit entries reference param codes. The enforcer resolves codes to actual `LifeCycleParamItem` objects.

4.6. 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) with initial status and next_due scheduling.
- **Insert-only rule:** EnsureConsumersInsertOnlyAsync — only inserts missing consumer rows, never overwrites existing status/next_due. Prevents rescheduling on restart or re-attach.
- Updates statuses on AckAsync calls from the app.
- Provides monitor dispatch methods:
  - ✓ ListDueLifecycleDispatchAsync — returns due lifecycle ACKs with full event context (policy re-resolved from stored JSON)
  - ✓ ListDueHookDispatchAsync — returns due hook ACKs with full event context (policy hook context re-resolved)
  - ✓ CountDueLifecycleDispatchAsync, CountDueHookDispatchAsync
- MarkRetryAsync — explicit retry scheduling.
- **Critical rule:** ACK identifiers are stable — re-dispatch always uses the same ack_guid for the same lifecycle/hook item.

4.7. LifeCycleMonitor (ILifeCycleMonitor) : scheduler

Role: Timer + safety gate.

- Runs on a configurable periodic interval (PeriodicTimer)
- Guarantees single active execution (Interlocked gate on _runGate)
- Calls WorkFlowEngine.RunMonitorOnceInternalAsync(ct) each tick
- *Monitor itself is generic and doesn't have any business logic. The logic lies within the workflow engine*.
- Handles monitor exceptions (raised as MONITOR_ERROR notice)
- Supports manual RunOnceAsync(ct) for testing

- Implements IAsyncDisposable

4.8. WorkFlowEngine (IWorkFlowEngine, also implements ILifeCycleEngine)

Role: The orchestrator + API surface.

- Owns all core components:
  - ✓ BlueprintManager, BlueprintImporter, StateMachine, PolicyEnforcer, AckManager, RuntimeEngine, Monitor
- Exposes public operations:
  - ✓ `TriggerAsync` — full transactional trigger pipeline
  - ✓ `AckAsync` — two overloads (by consumerId or by envCode+consumerGuid)
  - ✓ `RegisterConsumerAsync`, `BeatConsumerAsync` — consumer registration and heartbeat
  - ✓ `InvalidateAsync` — cache invalidation after imports
  - ✓ `StartMonitorAsync`, `StopMonitorAsync` — monitor lifecycle
  - ✓ `GetTimelineJsonAsync` — produces a complete JSON timeline for an instance
  - ✓ `UpsertRuntimeAsync`, `SetRuntimeStatusAsync`, `FreezeRuntimeAsync`, `UnfreezeRuntimeAsync` — runtime activity management
- Raises only two C# events:
  - ✓ `EventRaised` (actionable)
  - ✓ `NoticeRaised` (informational)
- Owns transactions for Trigger: DB changes in transaction -> commit -> then raise outbound events
- Implements IAsyncDisposable (stops monitor, disposes DAL)

4.9. RuntimeEngine (IRuntimeEngine)

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

- `UpsertAsync`(RuntimeLogByNameRequest): Create or update runtime activity rows (by name).
- `SetStatusAsync`(runtimeId, status): Update activity status.
- `SetFrozenAsync`(runtimeId, frozen): Freeze/unfreeze a runtime activity (exposed as `FreezeRuntimeAsync`/UnfreezeRuntimeAsync on the engine).
- Runtime rows link to: instance_id, lc_id (lifecycle), state_id, activity, status, actor_id.

- Does not participate in workflow correctness except for reporting and timeline.

4.10. Timeline

The engine provides `GetTimelineJsonAsync`(instanceId) which produces a comprehensive JSON document containing:

- **Instance metadata**: id, guid, external_ref, def_version, current_state, last_event, created, modified
- **Timeline**: ordered lifecycle entries, each with: from_state, to_state, event, event_code, actor, and nested runtime activities for that lifecycle step
- **Other Activities**: runtime entries not linked to any lifecycle step (lc_id = 0 or orphaned)

This is built via a single SQL query (QRY_INSTANCE.GET_TIMELINE_JSON_BY_INSTANCE_ID) using JSON aggregation.

5. Consumer Choreography (Principles)

5.1. 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.

5.2. 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.

5.3. "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).

5.4. 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").

5.5. 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.

5.6. 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)
- Engine API: `RegisterConsumerAsync`(envCode, consumerGuid), `BeatConsumerAsync`(envCode, consumerGuid)

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

5.7. Retry Attempts Are Not Burned While Consumer Is Down

When the monitor is about to dispatch/resend:

- Check heartbeat via `PushNextDueForDownAsync`
- If consumer is down:
  - ✓ Do not send
  - ✓ Do not increment trigger_count
  - ✓ Push next_due forward by ConsumerDownRecheckSeconds
- 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
  - ✓ **Suspend the instance** with a descriptive message
  - ✓ Fire ACK_SUSPEND notice

5.8. 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".

5.9. "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.

5.10. 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.
- Engine may emit **OverDue** as a *hint* when an instance is stuck after processing is complete, but it still won't move state automatically.

5.11. 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.

5.12. 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 (retries not burned).
- Max retries exceeded -> ACK failed -> instance suspended -> notice fired.
- Policy timeouts auto-trigger events; default staleness only fires advisory notices.
- Catch-up delivery is ordered, but consumers must reconcile against current state/timeline before acting.

6. JSON Policy Reference

6.1. Policy Structure

```
{
  "policy_name": "vendorregistration.policy",
  "for": { "definition": "vendorregistration", "version": 1 },
  "params": [
    {
      "code": "PARAMS.REG.PQ.VALIDATE",
      "data": { "approval": { ... } }
    }
  ],
  "rules": [
    {
      "state": "CheckVendorRegistered",
      "complete": { "success": 1001, "failure": 1002 },
      "emit": [
        {
          "event": "APP.REG.CHECK.VENDOR_REGISTERED",
          "complete": { "success": 1001, "failure": 1002 },
          "params": ["PARAMS.REG.PQ.VALIDATE"]
        }
      ]
    },
    {
      "state": "Overdue",
      "via": 1010,
      "emit": [
        {
          "event": "APP.REG.OVERDUE.NOTIFY",
          "complete": { "success": 1001, "failure": 1002 }
        }
      ]
    }
  ],
  "timeouts": [
    {
      "state": "PendingPQValidation",
      "timeout": "P2D",
      "timeout_event": 1010
    },
    {
      "state": "AwaitingApproval",
      "timeout_minutes": 60,
      "timeout_mode": "repeat",
      "timeout_event": 1011
    }
  ]
```

```
}
```

6.2. Policy Attachment Lifecycle

1. **Import time:** `BlueprintImporter` parses policy JSON, hashes it, and stores it. Timeouts are extracted into the timeouts table. Policy is linked to the definition via def_policy.
2. **Instance creation:** When an instance is first created, the latest policy for the definition is attached (instance.policy_id). This policy ID never changes for that instance.
3. **Runtime:** When a transition occurs, the engine reads the instance's attached policy (not the latest) to resolve rules, params, and hooks. This ensures behavioral consistency for in-flight instances even if the policy is updated.
4. **Re-dispatch:** Monitor re-resolves policy context from the stored policy_json column (carried alongside the ACK dispatch data), ensuring retries use the same policy that was active when the event was first created.