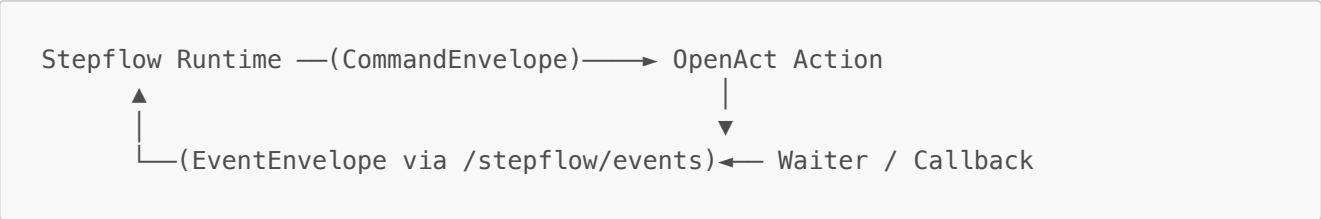
OpenAct Integration Guide (Command → Event)

This document describes how the OpenAct service integrates with Stepflow using the `aionix-protocol` envelopes. Stepflow already accepts these envelopes on both the command and event paths, so OpenAct only needs to adopt the same payload shapes.

1. Overview



- Commands are expressed as `CommandEnvelope` JSON; Stepflow runtime already constructs these when dispatching workflow tasks.
- OpenAct validates the command, executes the requested action, and returns:
  - **sync** results immediately when possible.
  - **async** acknowledgement + later emits an `EventEnvelope` to Stepflow's `/stepflow/events` endpoint (waiter behaviour).
- All request/response payloads should be validated with the `aionix-protocol` crate (Rust) or the generated SDKs for other languages.

2. Command Handling

OpenAct must expose a task execution endpoint (e.g. `POST /actions/{trn}/execute`) that takes a `CommandEnvelope`. Required fields:

Field	Purpose
`schemaVersion`	Presently `"0.1.0"`; reject unsupported versions.
`id`	Unique command ID (UUIDv7 recommended).
`timestamp`	RFC3339 issuance time.
`command`	Namespace verb such as `aionix.openact.action.execute`.
`source`	TRN of the caller (`trn:stepflow:{tenant}:engine`).
`target`	Action TRN to be executed.
`tenant`	Tenant/workspace identifier.
`traceId`	Propagated distributed trace.
`parameters`	Action-specific payload (usually `{ "input": ..., ... }`).
`correlationId`	Mirrors `command.id`; use it for downstream logging.

Command Validation Checklist

1. Parse with `aionix_protocol::parse_command_envelope` (Rust) or the generated validator.
2. Ensure the `target` TRN belongs to OpenAct and resolve it to an internal action.

3. Authorise using `tenant`, `actorTrn`, `authzScopes` if provided.
4. Derive execution context:

```
let action_trn = envelope.target;
let input = envelope.parameters.get("input").cloned().unwrap_or(Value::Null);
let trace_id = envelope.trace_id;
let idempotency_key = envelope.idempotency_key;
```

Command Response Patterns

Scenario	Response
Immediate completion	`200 OK` + payload `{ "status": "succeeded", "result": ... }`.
Job accepted, running asynchronously	`202 Accepted` + optional handle `{ "runId": "...", "phase": "running" }`.
Validation error	`400/422` with details.
Idempotency conflict	`409 Conflict`.

To allow Stepflow to correlate asynchronous work, include the following JSON fields in the response body:

```
{
  "phase": "running",
  "runId": "openact-run-1234",
  "heartbeatTimeout": 30,
  "statusTtl": 600
}
```

3. Waiter / Event Emission

When the action completes (success or failure), OpenAct must post an `EventEnvelope` to Stepflow's `/stepflow/events` endpoint. This is the waiter mechanism.

HTTP Endpoint

```
POST https://{stepflow-host}/api/v1/stepflow/events
Content-Type: application/json
Body: EventEnvelope
```

Use `aionix_protocol::parse_event_envelope` to validate before sending. On HTTP errors replay with exponential backoff; Stepflow's endpoint is idempotent (by `id`).

EventEnvelope Mapping

Field	Value
`specversion`	`"1.0"`
`id`	Unique event UUID

Field	Value
`source`	`trn:openact:{tenant}:executor` (or specific adapter TRN)
`type`	e.g. `aionix.openact.action.succeeded` / `.failed` / `.cancelled`
`time`	RFC3339 completion time
`data`	JSON payload: `{ "status": 200, ... }`
`aionixSchemaVersion`	`"1.1.0"`
`tenant`	Mirrors command tenant
`traceId`	Same as the command's `traceId`
`resourceTrn`	The action's TRN (`CommandEnvelope.target`)
`runId`	Identifier of the execution instance (derive from command parameters or backend handle)
`correlationId`	Command `id`
`relatedTrns`	Optional list (e.g. external resources touched)
`actorTrn`	If known

Example (success):

```
{
  "specversion": "1.0",
  "id": "018fb2e1-...",
  "source": "trn:openact:acme:executor",
  "type": "aionix.openact.action.succeeded",
  "time": "2025-10-10T12:36:20Z",
  "datacontenttype": "application/json",
  "data": {
    "status": 200,
    "durationMs": 480,
    "output": { "body": "..." }
  },
  "aionixSchemaVersion": "1.1.0",
  "tenant": "acme",
  "traceId": "00-4bf92f35-...",
  "resourceTrn": "trn:openact:acme:action/http/send-mail@v3",
  "runId": "openact-run-1234",
  "correlationId": "018fb0c2-...",
  "relatedTrns": [
    "trn:oss:acme:object/reports/2025-10-10.json@v1"
  ]
}
```

For a failure, change `type` to `aionix.openact.action.failed` and include `data.error`.

Recommended Workflow

1. Persist the command (idempotency check, state `RUNNING`).
2. Execute the action. If asynchronous, store the handle (runId / externalTrn).
3. Upon completion, build and post the `EventEnvelope`.

4. Update internal state + send any additional domain events if needed.

4. Optional: Server-Side Waiter Queue

To avoid losing events, encapsulate the waiter as a simple outbox:

```
async fn emit_event(event: &EventEnvelope) -> Result<(), anyhow::Error> {
    let client = reqwest::Client::new();
    client
        .post("https://stepflow/api/v1/stepflow/events")
        .json(event)
        .send()
        .await?
        .error_for_status()?;
    Ok(())
}
```

If the HTTP call fails, persist the envelope in a retry queue and retry with backoff.

5. OpenAct Orchestrator Runtime

OpenAct persists asynchronous executions before returning a `202 Accepted`. Two tables in the SQLite store keep the state durable even if the service restarts:

Table	Purpose	Key Columns
`orchestrator_runs`	One row per command execution (`run_id`). Tracks `status`, `phase`, `heartbeat_at`, `deadline_at`, serialized `result` / `error`, and metadata (traceId, correlationId).	`run_id` primary key
`orchestrator_outbox`	Pending `EventEnvelope` payloads waiting to be delivered back to Stepflow (or another orchestrator).	`id` (auto increment), `run_id`, `next_attempt_at`, `attempts`

Callback Endpoint

For long-running jobs, external systems (or internal pollers) can report completion through OpenAct's callback API:

```
POST /api/v1/orchestrator/runs/{runId}/completion
Content-Type: application/json
Body: { "status": "succeeded" | "failed" | "cancelled", "result": {...}, "error":
{...} }
```

- `status` – required; case-insensitive (`succeeded`, `failed`, `cancelled`).
- `result` – optional JSON payload used when the run succeeds.
- `error` – optional JSON payload describing the failure / cancellation reason.

The handler performs three steps:

1. Load the persisted run and update `status` / `phase` / `result` / `error`.
2. Create an appropriate `EventEnvelope` (`aionix.openact.action.succeeded|failed|cancelled`).

3. Enqueue the event into `orchestrator_outbox` for the dispatcher to deliver.

Background Workers

Two background tasks run inside the server; they are spawned automatically when the REST or unified server starts:

Task	Purpose	Key Env Vars (defaults)
`OutboxDispatcher`	Sends pending events to `OPENACT_STEPFLOW_EVENT_ENDPOINT` with retry/backoff.	`OPENACT_OUTBOX_BATCH_SIZE` (50), `OPENACT_OUTBOX_INTERVAL_MS` (1000), `OPENACT_OUTBOX_RETRY_INITIAL_MS` (30000), `OPENACT_OUTBOX_RETRY_MAX_MS` (300000), `OPENACT_OUTBOX_RETRY_FACTOR` (2.0), `OPENACT_OUTBOX_RETRY_MAX_ATTEMPTS` (5)
`HeartbeatSupervisor`	Scans `orchestrator_runs` for stale heartbeats, marks them `TIMED_OUT`, and enqueues timeout events.	`OPENACT_HEARTBEAT_BATCH_SIZE` (50), `OPENACT_HEARTBEAT_INTERVAL_MS` (1000), `OPENACT_HEARTBEAT_GRACE_MS` (5000)

Both tasks emit structured logs via `tracing`; hook these into your logging/metrics pipeline to monitor delivery success, retries, and timed-out runs.

State Transitions

1. `RUNNING` – persisted when the command is accepted; `heartbeat_at` is set immediately.
2. `SUCCEEDED` / `FAILED` – set either by the Stepflow handler (sync results) or by the callback API.
3. `CANCELLED` – callback API reports a cancellation (optional for orchestrators that support it).
4. `TIMED_OUT` – heartbeat supervisor detects `heartbeat_at` older than the grace period.

Every terminal state transition enqueues a corresponding event into the outbox so that the orchestrator is notified.

6. Testing Checklist

- Unit tests validating `CommandEnvelope` → action dispatcher (use the schema validator).
- Unit tests building `EventEnvelope` for each outcome.
- Integration test with Stepflow sandbox:
    1. POST command to OpenAct's execute endpoint.
    2. Simulate async completion → POST to `/api/v1/orchestrator/runs/{runId}/completion`.
    3. Confirm the outbox dispatcher emits `EventEnvelope` to Stepflow and the workflow continues.

7. References

- Protocol schemas: https://github.com/aionixone/aionix-protocol
- Stepflow event handler: `crates/stepflow-http-server/src/handlers/events.rs`
- Runtime ingestion: `crates/stepflow-runtime/src/services/events.rs`
- Runtime command dispatch (builder): `crates/stepflow-runtime/src/runtime/task_handler.rs`

With this guide, OpenAct can implement envelope-based command ingestion and waiter-driven event emission that matches Stepflow's expectations.