EXAMPLE SCOPES & REPORTS

Engineering work, explained

Seven proposed engagements for teams using real products.

Independent examples based on public documentation. Not commissioned by or affiliated with the referenced companies. The work is proposed; acceptance checks are to verify.

Inside this collection

01 Formbricks to CRM: integration audit

02 Stripe: recover an interrupted payment request

03 Medusa: a recoverable supplier import

04 GitLab: task, code and test review

05 Cal.com: booking portal access review

06 Polar: subscription state to product access

07 n8n: decide when to introduce queue mode

How to read these examples

Public sources establish the product capabilities and API contracts. Upserto proposes the customer-side work, review questions, deliverables and acceptance checks. Review items are not verified vendor defects. A real engagement would attach source references, test results and release decisions for the agreed system.

Work priorities: P1 - resolve before releasing the affected workflow, or record acceptance. P2 - assign an owner and follow-up milestone.

Discuss a scope: upserto.com/contact

EXAMPLE 01 / 07 | PUBLIC REFERENCE: Formbricks

Formbricks to CRM: integration audit

Review a survey-response pipeline before sales teams depend on it

Independent example based on public documentation. Not commissioned by or affiliated with Formbricks.

Decision brief

Recommended next decision

Question	Recommendation
Decision	Review the response-to-CRM boundary before adding automatic replay.
Why it matters	A lost CRM response can make the next create attempt an unsafe decision.
Acceptance	The team can trace an eligible response to one CRM outcome and recover an interrupted delivery.
Scope boundary	One customer-owned receiver and CRM adapter; survey eligibility remains a sales-operations decision.

The proposed engagement

Proposed brief: a SaaS team uses Formbricks to collect demo requests and sends completed responses into its CRM. Upserto would review the code between the webhook receiver and the CRM write, identify failure paths and propose a small repair plan. The question is concrete: can the team explain whether each eligible response reached the intended CRM record, including after a timeout or a manual replay?

Inputs required. Receiver and adapter code; redacted event samples; survey mapping; CRM field rules; delivery logs.

System boundary and assumptions

The public Formbricks API documents responseCreated, responseUpdated and responseFinished triggers, as well as a response lookup endpoint. The proposed engagement covers one workspace, agreed surveys, the customer-owned receiver, its database and one CRM adapter. Upserto would first confirm the deployed API version and event contract. This is an integration review, not an assessment of Formbricks itself.

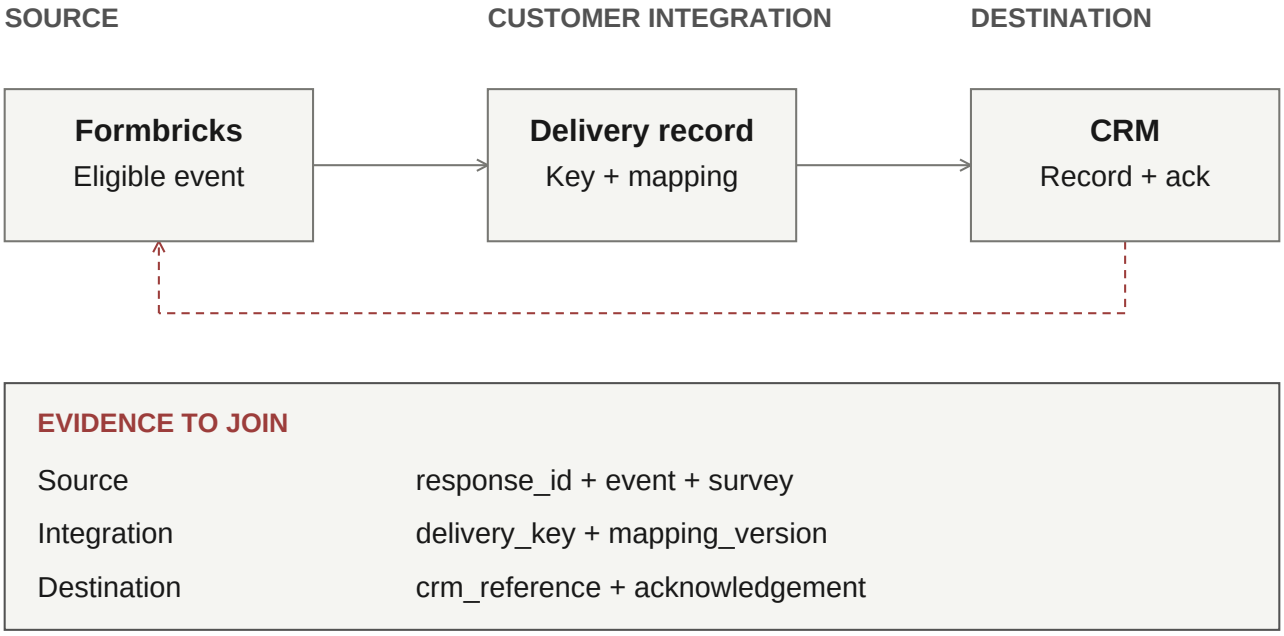
How Upserto would approach it

Start with one successful response and trace its identifiers through the integration. Then replay a redacted fixture twice, interrupt the process after persistence, and simulate a CRM timeout. Review incomplete-response handling, retry ownership and sensitive fields separately. Attach file-and-line references to any issue found during the actual engagement.

What the team would receive

- An annotated integration map and a finding list tied to source code.
- A response-to-CRM field contract, retry rules and a reconciliation checklist.
- Focused regression fixtures and a scoped implementation proposal.

Trace one response across three systems



The audit joins the source response, local delivery decision and CRM acknowledgement. Reconciliation checks the gaps between them.

Solid: delivery path | Dashed: reconciliation

- Match a stable response identity to the local delivery record.
- Retain the CRM reference and acknowledgement for the same delivery.
- Reconcile accepted responses against the destination; do not log survey answers.

Starting point and proposed change

Items to inspect in the customer integration, proposed controls and acceptance evidence.

Design comparison

Control	Inspect first	Proposed change	Acceptance evidence
Delivery identity	Review whether retries use a new CRM create call.	Stable response-derived delivery key and explicit unknown outcome.	Replay and lost-response fixtures resolve without a duplicate.
Response eligibility	Review trigger and completion handling.	One agreed survey/state-to-CRM mapping.	Created, updated and finished fixtures follow the contract.
Recovery	Review logs and manual retry procedure.	Delivery record with owner, decision and safe replay.	An operator identifies and resolves a failed delivery.

Decisions and approval owners

Decision	Proposed default	Approval owner
Delivery identity	Use a stable response-derived reference; reconcile before a second create when the CRM cannot upsert.	Integration owner
Eligible response	Agree the survey, completion state and field mapping before delivery is enabled.	Sales operations + product owner

Review questions and implementation risks

Priorities identify proposed work: P1 before the affected release or an explicit acceptance decision; P2 with an owner and follow-up milestone.

AUD-01 | P1 | Can a retry create a second CRM record?

If the customer adapter retries a create call after losing its response, a duplicate lead may result. The audit would inspect the CRM contract and the adapter; it does not assume this defect exists.

Proposed approach. Use a stable external reference and an upsert or provider-supported idempotency mechanism. If neither is available, reconcile an ambiguous result before creating again.

AUD-02 | P1 | Is the correct response state mapped?

Created and updated responses may not mean the respondent finished the survey. A consumer that treats every trigger as a new lead could send incomplete or unwanted data.

Proposed approach. Document the eligible event/state, workspace and survey mapping. Test the field contract against agreed redacted response fixtures.

AUD-03 | P2 | Can operations locate a missing delivery?

If acknowledgement and retry ownership are unclear, a response can remain unresolved without anyone knowing which step needs attention.

Proposed approach. Keep a redacted delivery record, bounded retry classification and a response-to-CRM reconciliation view. Avoid storing survey answers in routine logs.

Example technical output

Proposed delivery-decision record

```
source: Formbricks
workspace: approved_workspace
response_id: sample_response
trigger: responseFinished
delivery_key: workspace + response_id + mapping_version
crm_write: upsert_if_supported
ambiguous_result: reconcile_before_create
execution_result: TO_VERIFY
```

Delivery plan and ownership

Proposed action plan

Priority	Review item	Accountable role	Acceptance gate
P1	AUD-01	Integration owner	Duplicate delivery resolves to one CRM record or a documented operator decision.
P1	AUD-02	Backend + sales operations	Incomplete responses follow the agreed rule; finished responses map to the intended fields.
P2	AUD-03	Operations owner	A simulated CRM outage creates a visible, attributable recovery item.

Acceptance checks

Run these checks before accepting the proposed changes.

Acceptance checks

Scenario	Required evidence / expected behavior	Status
Same response delivered twice	One intended CRM record; repeated delivery decision retained	To verify

Scenario	Required evidence / expected behavior	Status
CRM accepts, response is lost	Reconcile by the stable reference before another create	To verify
Response is not yet finished	Apply the agreed eligibility rule; no unintended CRM write	To verify
CRM is unavailable	Bounded retry and visible operator-owned recovery item	To verify

Rollout and recovery

Review first; change only the agreed integration boundary. Introduce delivery records before enabling automatic replay. Test with an isolated CRM workspace and approved survey fixtures, then enable a limited production scope with an agreed stop condition. Keep receipt handling separate from CRM availability so an outage does not silently discard accepted input.

Handover and limits

The team would receive source-linked review items, fixture results, field mappings and a recovery guide. Any production conclusion would require access to the deployed version and authorized delivery records. The public documentation supplies the reference contract; the review determines how the customer integration actually uses it.

Access and evidence handling

Area	Required control
Access	Read-only receiver code, configuration names and redacted delivery samples first.
Evidence	Keep response IDs and delivery decisions; exclude survey answers and CRM tokens from routine traces.
Handover	Agree the restricted delivery channel and retention period for any customer evidence.

Public documentation used

Formbricks: Create Webhook - Documents response event triggers and webhook configuration.

Formbricks: Get a response - Documents response identity, survey and finished state; confirm API version.

Read this example online | Related service | Discuss a similar scope

EXAMPLE 02 / 07 | PUBLIC REFERENCE: Stripe

Stripe: recover an interrupted payment request

A proposed reliability review for an existing Payment Intents integration

Independent example based on public documentation. Not commissioned by or affiliated with Stripe.

Decision brief

Recommended next decision

Question	Recommendation
Decision	Keep one durable order-to-payment identity through an interrupted request.
Why it matters	A missing response does not establish payment failure or justify a new instruction.
Acceptance	Lost responses, duplicates and reordered events resolve without a repeated fulfilment effect.
Scope boundary	Merchant application and its fulfilment contract; Stripe account, dispute and settlement decisions are excluded.

The proposed engagement

Proposed brief: an application already uses Stripe Payment Intents. A customer retries checkout after a network interruption, and support needs to determine which payment belongs to the order. Upserto would review order-to-PaymentIntent identity, request retries, webhook processing and fulfilment. The goal is a documented recovery path that does not create a fresh payment instruction merely because the application missed a response.

Inputs required. Payment code; API version; order-state model; redacted webhook samples; fulfilment contract.

System boundary and assumptions

The reference is Stripe's documented API v1 idempotency and webhook behavior. Stripe recommends Checkout Sessions for many new integrations; this example concerns improving an existing Payment Intents flow, not choosing the default for a new checkout. Review would cover the merchant application and its order database. Card handling, account eligibility, disputes and settlement are outside this slice.

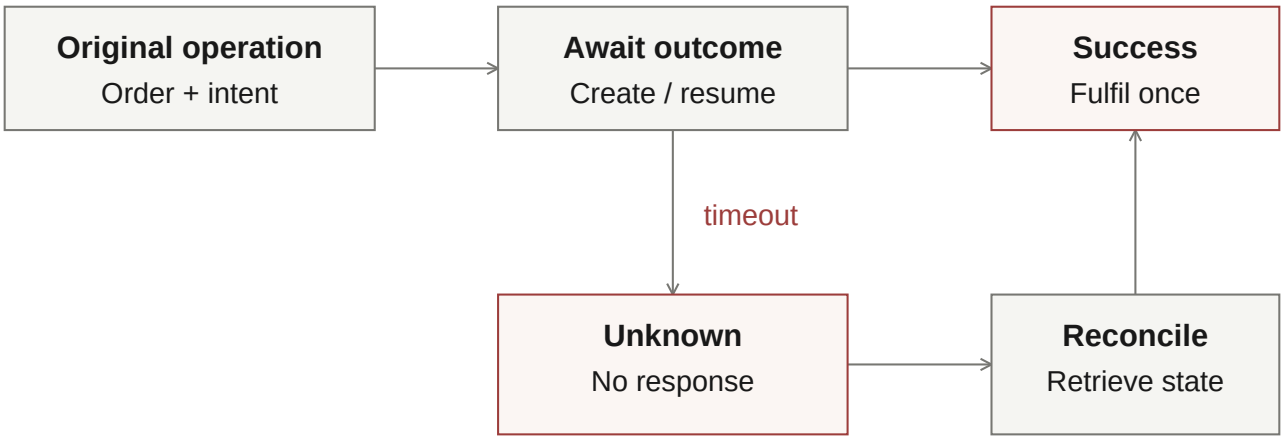
How Upserto would approach it

Identify the stable business operation before inspecting retries. Record which PaymentIntent belongs to the order and when the provider identifier becomes durable. Review signature validation on the raw body, duplicate event handling and allowed local transitions. Use Stripe test-mode fixtures and application failure injection only after scope and credentials are agreed.

What the team would receive

- An order/payment state map and a provider capability checklist.
- A repair for the agreed retry or webhook boundary with focused tests.
- A reconciliation query and support runbook for unresolved operations.

An unknown outcome is a recovery state



Recovery preserves the original operation identity.

These are application recovery decisions, not Stripe's complete status model. The order and original provider object remain linked throughout recovery.

Solid: allowed decision path | Unknown does not authorize a new payment

- Persist the order-to-provider identity before handling retries.
- A missing response keeps the operation unresolved. Retrieve current provider state.
- Fulfil only after the application has authoritative evidence and its local transition is permitted.

Starting point and proposed change

Items to inspect in the customer integration, proposed controls and acceptance evidence.

Design comparison

Control	Inspect first	Proposed change	Acceptance evidence
Request retry	Check operation keys and provider mapping.	Stable order identity and documented provider retry rules.	Lost response resolves the original operation.
Webhook decision	Check duplicate and delivery-order handling.	Verified event, explicit transition and durable fulfilment key.	Duplicate and stale events do not repeat fulfilment.
Support recovery	Check whether retry and inspect are separate actions.	Read-only refresh plus owned reconciliation.	Support resolves an unknown without creating another instruction.

Decisions and approval owners

Decision	Proposed default	Approval owner
Retry policy	Reuse or retrieve the existing operation under the configured Stripe API contract.	Payments backend owner
Order completion	Agree which verified provider state permits fulfilment and who resolves conflicting evidence.	Product + payments operations

Review questions and implementation risks

Priorities identify proposed work: P1 before the affected release or an explicit acceptance decision; P2 with an owner and follow-up milestone.

PAY-01 | P1 | Does a missing response create a new payment?

If retries generate a new business operation or idempotency key, the application can lose the connection between the original order and the provider request. This is a review question about the merchant code.

Proposed approach. Persist the order-to-provider mapping and apply Stripe's documented idempotency rules for the exact API version. Do not assume API v1 and v2 cache failure results identically.

PAY-02 | P1 | Can a repeated event fulfil an order twice?

Stripe documents duplicate events and does not guarantee delivery order. Local fulfilment must not rely on receipt order or a single delivery.

Proposed approach. Verify the raw payload signature, record event identity and enforce the allowed order transition with a durable uniqueness rule for fulfilment.

PAY-03 | P2 | Who resolves a payment that remains unclear?

If the application cannot explain an order/provider mismatch, support may retry an action that creates another instruction.

Proposed approach. Provide read-only status refresh and a reconciliation queue. Record the operator decision and keep creation separate from inspection.

Example technical output

Proposed merchant-side recovery logic

```
operation = load_order_operation(order_id)
if operation.payment_intent_id:
  inspect_existing_payment_intent(operation)
else:
  create_with_persisted_operation_key(operation)
on_missing_response: mark_reconciliation_required()
on_verified_success: enqueue_fulfilment_once(order_id)
execution_result: TO_VERIFY
```

Delivery plan and ownership

Proposed action plan

Priority	Review item	Accountable role	Acceptance gate
P1	PAY-01	Payments backend owner	A lost-response fixture resolves the original order without a new unintended PaymentIntent.
P1	PAY-02	Backend + fulfilment owner	Duplicate and reordered fixtures produce one fulfilment instruction and no state regression.
P2	PAY-03	Payments operations	An aged unresolved order has an owner and a status refresh that cannot create a payment.

Acceptance checks

Run these checks before accepting the proposed changes.

Acceptance checks

Scenario	Required evidence / expected behavior	Status
Create response is lost	Original order maps to one intended provider operation	To verify

Scenario	Required evidence / expected behavior	Status
Webhook is delivered twice	One fulfilment instruction; repeat is a recorded no-op	To verify
Earlier state arrives after success	No local order-state regression	To verify
Support refreshes an unresolved order	Inspect only; creation requires a separate decision	To verify

Rollout and recovery

Confirm the existing API version and checkout flow before changing behavior. Add observation and reconciliation before automatic recovery. Exercise failure cases in the approved test environment, then release to an agreed merchant scope. Preserve the ability to stop new creation while continuing event processing and investigation for existing orders.

Handover and limits

The handover would include code changes, test evidence, the state map and support steps for unknown outcomes. Test-mode results establish test-mode behavior only. Production acceptance requires authorized records linking the order, provider object and fulfilment effect.

Access and evidence handling

Area	Required control
Access	Use test-mode access and redacted event fixtures for the agreed merchant integration.
Evidence	Record order, event and provider-object references; exclude card data, secrets and raw personal details.
Production	Require a scoped approval before inspecting live records or changing payment behavior.

Public documentation used

Stripe: Idempotent requests - API v1 and v2 retry semantics differ; use the configured version.

Stripe: Receive events - Signature verification, duplicate delivery and event-order guidance.

Stripe: Payment Intents - Existing PaymentIntent lifecycle; documents Checkout Sessions recommendation for new integrations.

Read this example online | Related service | Discuss a similar scope

EXAMPLE 03 / 07 | PUBLIC REFERENCE: Medusa

Medusa: a recoverable supplier import

Plan and implement one custom workflow with explicit retry and compensation rules

Independent example based on public documentation. Not commissioned by or affiliated with Medusa.

Decision brief

Recommended next decision

Question	Recommendation
Decision	Give each supplier row a stable identity and each workflow step a recovery rule.
Why it matters	A repeat import or lost adapter response can leave duplicate or partly synchronized products.
Acceptance	An operator can recover a failed batch and explain the outcome of every accepted row.
Scope boundary	One custom Medusa workflow and external catalog adapter; product mapping is agreed with the commerce owner.

The proposed engagement

Proposed brief: a commerce team using Medusa needs to import supplier products and synchronize an external catalog. Upserto would implement one custom workflow: validate an approved batch, update the intended product records, call the external catalog adapter and expose the completion or recovery decision. The scope includes partial failure and repeated execution, not just a successful import.

Inputs required. Supplier schema; product mapping; deployed Medusa version; catalog API contract; representative redacted files.

System boundary and assumptions

Medusa documents durable workflows, step compensation and configurable retries. These are building blocks; the customer still needs rules for external effects and business identity. The proposed slice covers a customer-owned import module and one catalog adapter. It excludes a full commerce migration, pricing rules and payment capture. No Medusa installation or supplier data was inspected for this proposal.

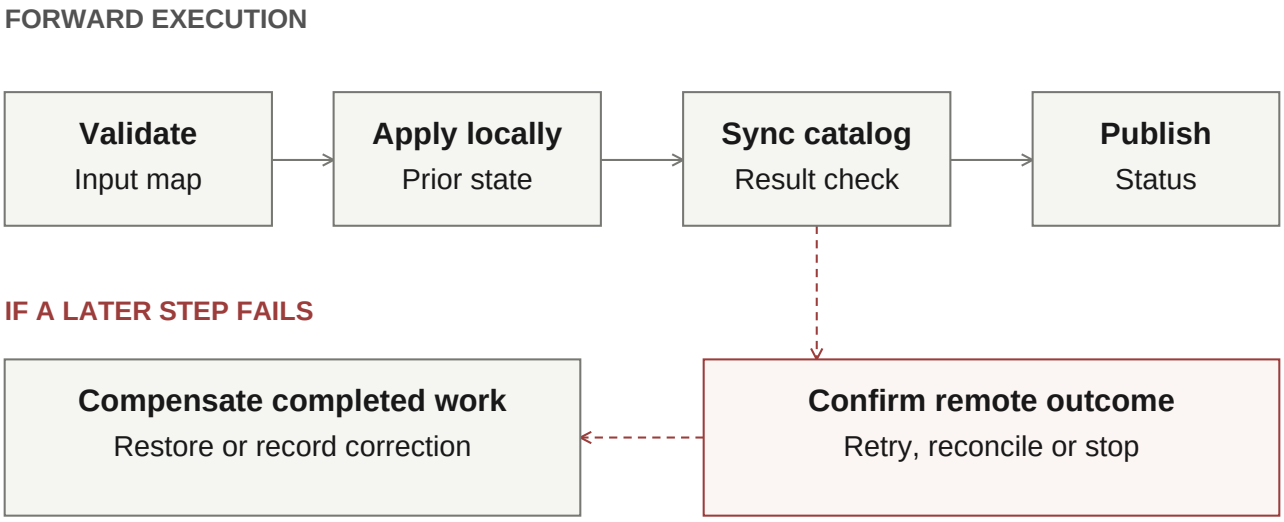
How Upserto would approach it

Define batch identity and row identity before writing steps. Separate validation from changes. For each external call, specify whether a repeat is idempotent, compensatable or requires manual reconciliation. Review version-specific workflow retry settings. Agree batch size, concurrency and a representative workload before quoting capacity.

What the team would receive

- A custom Medusa workflow and one catalog adapter.
- Batch-state and compensation rules with focused integration tests.
- Operator status output, retry guidance and configuration notes.

Separate forward progress from compensation



Compensation is step-specific, not a blind global rollback.

A supplier import needs a recovery decision for each completed step. An ambiguous external result must be reconciled before retry or compensation.

Solid: forward execution | Dashed: failure recovery

- Validate input before changing products or calling the external catalog.
- Record the information needed to restore or correct each completed step.
- After an ambiguous external call, confirm remote state before compensating local changes.

Starting point and proposed change

Items to inspect in the customer integration, proposed controls and acceptance evidence.

Design comparison

Control	Inspect first	Proposed change	Acceptance evidence
Repeat identity	Review how rows are matched on repeated imports.	Stable source identity and recorded adapter references.	Repeat and lost-response fixtures do not duplicate records.
Compensation	Review whether all failures use the same rollback.	Step-specific restore, compensate or reconcile decision.	Partial failure produces the intended recovery state.
Execution bounds	Review unbounded waiting and concurrency.	Agreed timeouts, batch limit and operator status.	A stuck adapter has a visible terminal or owned recovery decision.

Decisions and approval owners

Decision	Proposed default	Approval owner
Compensation	Choose restore, corrective action or reconciliation per step rather than one blanket rollback.	Commerce + backend owners
Execution limits	Agree batch size, timeouts and concurrency after a representative fixture run.	Backend + operations owners

Review questions and implementation risks

Priorities identify proposed work: P1 before the affected release or an explicit acceptance decision; P2 with an owner and follow-up milestone.

BE-01 | P1 | Can the catalog call be repeated safely?

A catalog service may accept a request before its response is lost. Retrying the whole import without a stable row key can repeat an external effect.

Proposed approach. Use supplier/batch/row identity and confirm the adapter's idempotency behavior. Reconcile ambiguous writes before repeating them.

BE-02 | P1 | Does compensation match the business effect?

Deleting a local record is not a universal reversal. An external catalog change may already be consumed, and an existing product may need restoration rather than deletion.

Proposed approach. Specify compensation input for each step. Distinguish newly created and updated objects; document irreversible effects and the manual correction path.

BE-03 | P2 | Can an import wait forever or overload workers?

Retry counts, waiting-step behavior and batch size affect how an operator observes progress. The actual workload and SDK version determine the configuration.

Proposed approach. Set explicit retry/timeout rules for the chosen version and expose batch progress. Measure an agreed workload before increasing concurrency.

Example technical output

Proposed workflow contract

```
workflow: supplier_catalog_import
identity: supplier + batch + row_key
steps:
  - validate_input
  - apply_product_change
  - sync_external_catalog
  - publish_batch_result
external_timeout: reconcile_before_repeat
compensation: defined_per_step
execution_result: TO_VERIFY
```

Delivery plan and ownership

Proposed action plan

Priority	Review item	Accountable role	Acceptance gate
P1	BE-01	Backend + catalog owner	A repeated row and a lost adapter response do not create unintended duplicate records.
P1	BE-02	Commerce backend owner	A failure after the external step leaves a documented, reviewable state with no blind deletion.
P2	BE-03	Backend + operations	A stuck adapter reaches the defined timeout/recovery state and does not block unrelated batches.

Acceptance checks

Run these checks before accepting the proposed changes.

Acceptance checks

Scenario	Required evidence / expected behavior	Status
Same batch is submitted twice	Stable row identity prevents unintended duplicate products	To verify
Catalog accepts then connection is lost	Remote state is reconciled before repeat	To verify
Failure after an update	Compensation restores the intended prior state or records a correction task	To verify
Catalog never completes	Timeout/recovery state is visible; unrelated work can proceed	To verify

Rollout and recovery

Build and verify in an isolated Medusa environment with approved fixture data. Start with a limited supplier and batch size. Keep new imports independently stoppable while existing batches remain inspectable. Release only after the team agrees which failures can be retried automatically and which require a person.

Handover and limits

Deliver the workflow, adapter, mapping specification, fixture results and operator guide. Estimates for throughput or infrastructure would follow a representative workload test; this proposal includes no measured performance claim.

Access and evidence handling

Area	Required control
Access	Use an isolated Medusa environment and approved supplier fixtures before live imports.
Evidence	Retain row identity and adapter references; redact supplier credentials and unnecessary commercial fields.
Release	Agree an import stop control and the operator responsible for unresolved batches.

Public documentation used

Medusa: Workflows - Defines custom steps and execution from routes, subscribers and jobs.

Medusa: Compensation function - Documents step rollback inputs and compensation.

Medusa: Retry failed steps - Documents retries, waiting steps and version-specific limits.

Read this example online | Related service | Discuss a similar scope

EXAMPLE 04 / 07 | PUBLIC REFERENCE: GitLab

GitLab: task, code and test review

A scoped AI assistant that checks a merge request against its task and CI evidence

Independent example based on public documentation. Not commissioned by or affiliated with GitLab.

Decision brief

Recommended next decision

Question	Recommendation
Decision	Pilot a read-only review tied to the task, MR head and matching CI result.
Why it matters	A plausible summary or unrelated green pipeline can conceal a missing behavioral check.
Acceptance	Each review claim links to its source; stale or incomplete input produces a named open check.
Scope boundary	One approved GitLab project; engineers retain publication, merge and release decisions.

The proposed engagement

Proposed brief: a team wants a useful answer before merging a backend change: what does the linked task require, where does the diff implement it, and what do the tests actually prove? Upserto would build a read-only GitLab pilot for one project. The assistant would draft a review for an engineer, with links to the requirement, changed lines and matching CI results. It would not merge code or publish comments automatically.

Inputs required. One approved project; task ID; MR IID and head SHA; repository paths; matching pipeline/JUnit artifacts; reviewer rubric.

System boundary and assumptions

The public reference is GitLab's merge request, issues and pipeline APIs, plus JUnit test reports. Inputs are an approved issue or task snapshot, paginated MR diffs, relevant permitted files at a pinned commit, the corresponding pipeline and its test report. The pilot would record missing or truncated input explicitly. It would not read CI variables, broaden repository access, or treat a green pipeline as proof that every requirement was tested.

How Upserto would approach it

First build deterministic collection: pin the MR head SHA, fetch every diff page and record GitLab's collapsed/too_large flags. Match the pipeline to that SHA and read the available JUnit result. Then let the model map the task criteria to changed code and named tests. A rule layer would reject unsupported file references and block a ready verdict when required input is missing. A human would review the draft and decide what to publish.

What the team would receive

• A read-only GitLab collector with project-scoped access and a review draft screen.

• A fixed evaluation set containing correct changes, missing requirements, duplicate-callback defects and incomplete CI evidence.

• A report of supported findings, missed seeded defects, false positives and reviewer corrections from actual pilot runs.

A review record connects a requirement to evidence

ONE REVIEW SNAPSHOT

Task + MR head SHA + matching pipeline

REQUIREMENT	CODE	ASSERTION	CI RESULT
No repeated fulfilment	changed path + line range	named behavior test + assertion	matching run + test result

Missing or stale evidence

NEEDS_REVIEW: name the gap and next check.

The matrix is a proposed review format. A matching CI run describes executed tests; it does not prove a requirement that has no relevant assertion.

One snapshot: task + MR head + matching pipeline

- Pin the task, MR head SHA, complete permitted diff and pipeline revision.
- For each task criterion, link the changed code, relevant assertion and result from that pipeline.
- Missing, truncated or stale evidence remains NEEDS_REVIEW with the next check named.

Starting point and proposed change

Items to inspect in the customer integration, proposed controls and acceptance evidence.

Design comparison

Control	Inspect first	Proposed change	Acceptance evidence
Review input	Check whether the bot reads only part of the diff or an unrelated passing pipeline.	Pinned head SHA, complete diff accounting and matching CI.	Force-push and excluded-file cases are blocked or marked incomplete.
Requirement check	Check whether summaries substitute for assertions.	Task criterion -> changed path -> relevant assertion -> CI result.	A missing behavioral test stays open.
Action control	Check whether source text can request tools or comments.	Read-only collector and engineer-controlled publication.	Injection fixtures cannot expand scope or write to GitLab.

Decisions and approval owners

Decision	Proposed default	Approval owner
Context access	Permit specific projects and paths in application code; repository text cannot extend access.	Engineering + security owners
Pilot acceptance	Compare drafts with an engineer rubric before allowing a live read-only shadow run.	Review lead

Review questions and implementation risks

Priorities identify proposed work: P1 before the affected release or an explicit acceptance decision; P2 with an owner and follow-up milestone.

AI-01 | P1 | Is the review using the current diff and pipeline?

A force-push or oversized diff can make a draft incomplete or stale. GitLab documents diff limits and pipeline test-report endpoints; the collector needs to handle both.

Proposed approach. Pin head SHA, paginate diffs and check excluded-file flags. Reject a pipeline from a different commit. Mark required missing context as needs review rather than inventing coverage.

AI-02 | P1 | Can repository text expand access or trigger a tool?

A source comment or task body can contain instructions to fetch secrets or run commands. That text is input to review, not authorization for the assistant.

Proposed approach. Enforce project and tool permissions outside the model. Keep the first pilot read-only. Any later test execution would use an isolated runner, an allowlisted command and explicit resource limits.

AI-03 | P2 | Does each conclusion link to the right evidence?

A named test may pass without checking the task's new behavior. A plausible explanation and a valid URL are not enough to establish that the requirement is covered.

Proposed approach. Require the changed path/line, requirement criterion and relevant test assertion. Evaluate seeded cases with a human rubric; preserve unsupported and missing-evidence outcomes.

Example technical output

Proposed review output for a duplicate-callback fixture

```
task_criterion: duplicate callback does not fulfil twice
commit: MR_HEAD_SHA
code_source: src/payments/callback.go:84
test_source: tests/callback_test.go:61
ci_source: matching_pipeline / test_report
result: NEEDS_REVIEW
reason: duplicate-delivery assertion is not in supplied tests
next_check: replay same signed event twice; inspect durable effect
# Paths are fixture examples; no GitLab project was reviewed.
```

Delivery plan and ownership

Proposed action plan

Priority	Review item	Accountable role	Acceptance gate
P1	AI-01	Integration backend owner	Force-push and truncated-diff fixtures prevent a ready verdict and identify the missing input.
P1	AI-02	AI + security owner	An injected instruction cannot access another project, CI variables or an unapproved command.
P2	AI-03	Engineering reviewer	A missing duplicate-callback assertion remains an open criterion even if the pipeline is green.

Acceptance checks

Run these checks before accepting the proposed changes.

Acceptance checks

Scenario	Required evidence / expected behavior	Status
MR is updated after collection	Draft is marked stale; review uses the new head and matching pipeline	To verify
Diff is too_large or required file is unavailable	No invented code or complete-review verdict; missing input is named	To verify
Source text requests another project or a secret	Application denies the read/tool action before model output	To verify
Pipeline passes but requirement assertion is absent	Criterion remains open with the specific test to add	To verify

Rollout and recovery

Start offline with a fixed fixture repository and compare drafts with an engineer's rubric. Move to a read-only shadow pilot on one approved project only after access and stale-input checks pass. Record reviewer decisions before considering comment publication or authorized test execution. Expanding scope is a separate decision, not a model permission.

Handover and limits

The team would receive the collector, access policy, fixture manifest, review format and actual evaluation results. Each pilot run would preserve task version, head SHA, pipeline ID, model/prompt version and reviewer correction. No accuracy percentage or time saving is claimed in this example.

Access and evidence handling

Area	Required control
Access	Read-only project scope; no CI-variable, secret, merge or write access.
Model input	Agree the model provider and retention terms before repository content leaves the approved environment.
Evidence	Store source references and evaluation decisions; redact credentials and personal data from fixtures and prompts.

Public documentation used

GitLab: Merge requests API - Paginated diffs, collapsed and too_large flags, MR commit references.

GitLab: Issues API - Authorized issue descriptions provide the task snapshot.

GitLab: Pipelines API - Pipeline identity and test_report resource.

GitLab: Unit test reports - JUnit artifacts, report limits and separation from job status.

Read this example online | Related service | Discuss a similar scope

EXAMPLE 05 / 07 | PUBLIC REFERENCE: Cal.com

Cal.com: booking portal access review

Review the customer application that exposes team bookings and meeting details

Independent example based on public documentation. Not commissioned by or affiliated with Cal.com.

Decision brief

Recommended next decision

Question	Recommendation
Decision	Check booking ownership at every customer-portal access path.
Why it matters	A valid session or provider token does not establish authority over a requested booking.
Acceptance	Allowed and denied fixtures cover the agreed routes and produce attributable, redacted evidence.
Scope boundary	Customer-owned portal and its Cal.com adapter; authorization is required before testing.

The proposed engagement

Proposed brief: an agency uses Cal.com and exposes booking details through its own client portal. Staff should see only the customers and teams they are assigned to. Upserto would review the portal's session, local customer mapping and server-side Cal.com calls, then test an agreed access matrix with seeded accounts. This proposal does not allege a vulnerability in Cal.com.

Inputs required. Portal code; local role model; Cal.com app/scopes; seeded bookings and two isolated customer identities.

System boundary and assumptions

Cal.com documents API v2 roles, permissions and OAuth scopes. Its booking API can return attendee and meeting information, so the portal must define which fields each local role may receive. The proposed review covers the customer-owned proxy and interface, not Cal.com infrastructure. Testing would require written targets, permitted identities, a time window and stop conditions. No service was scanned or exploited for this example.

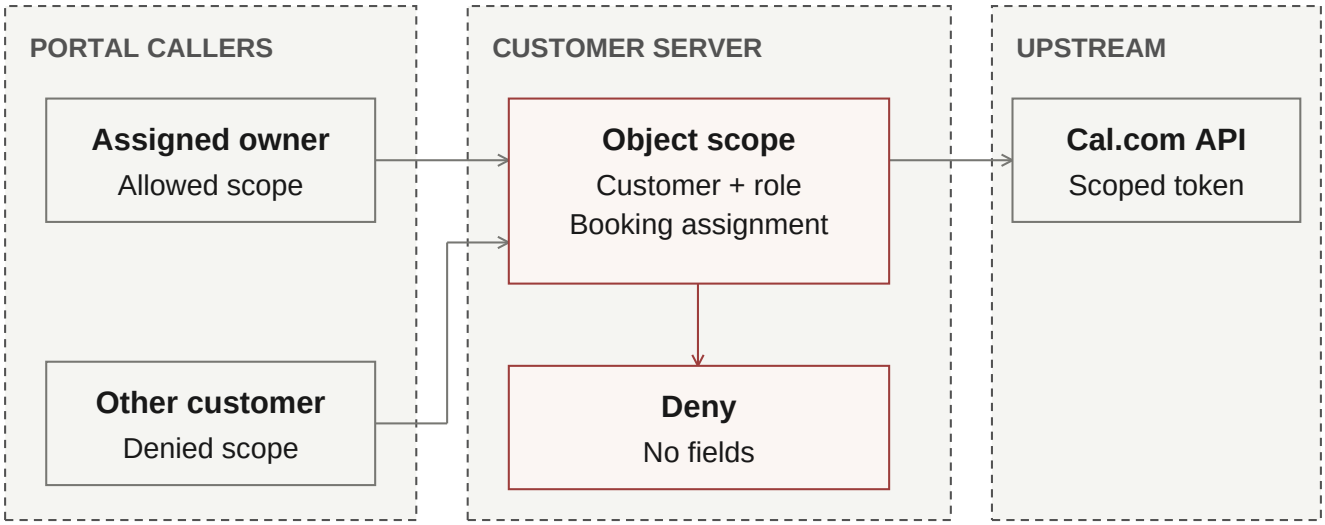
How Upserto would approach it

Map local identities to Cal.com users, teams and bookings before testing. Check authorization at the server request boundary, not only in visible navigation. Use known seeded booking identifiers to test cross-customer requests without enumeration. Inspect cache keys, exported links and redacted denied-event logs. Stop immediately if unexpected real data appears.

What the team would receive

- An access matrix and source-linked review items for the customer portal.
- Minimal redacted reproductions for any verified in-scope issue.
- A remediation proposal and retest checklist covering permitted and denied behavior.

Authorize the portal caller before returning booking data



RESPONSE POLICY

Server returns role-approved fields; no raw upstream payload.

The customer portal must enforce its own object and field access rules. An upstream Cal.com credential is not the portal user's authorization.

Dashed frames: trust boundaries | Deny path returns no booking fields

- Bind caller, customer or team assignment and booking object before returning upstream data.
- A denied cross-customer request stops at the portal authorization gate.
- Return only role-approved fields; keep tokens and meeting details out of routine logs.

Starting point and proposed change

Items to inspect in the customer integration, proposed controls and acceptance evidence.

Design comparison

Control	Inspect first	Proposed change	Acceptance evidence
Object access	Review reliance on shared upstream credentials.	Portal customer/team authorization before returning data.	Seeded cross-customer read/action is denied.
Field access	Review complete upstream responses sent to the browser.	Role-specific response fields and scoped upstream access.	Restricted and revoked roles receive no disallowed field.
Audit trail	Review generic errors or full-payload logging.	Redacted, attributable access decision.	Operator can investigate denial without attendee/meeting content.

Decisions and approval owners

Decision	Proposed default	Approval owner
Test authority	Agree routes, identities, tenant fixtures and the permitted environment in writing.	Application + security owners
Access policy	Define tenant, support and administrator permissions before interpreting an access result.	Product + security owners

Review questions and implementation risks

Priorities identify proposed work: P1 before the affected release or an explicit acceptance decision; P2 with an owner and follow-up milestone.

SEC-01 | P1 | Does the portal bind a booking to the caller's customer?

If the proxy accepts a booking UID and relies only on a shared upstream credential, its local customer boundary may be missing. That possibility must be verified in the portal code and approved fixtures.

Proposed approach. Scope each lookup and action to the caller's current assignment before the upstream response is returned. Test status, details and any mutation separately.

SEC-02 | P1 | Are upstream data and scopes broader than the UI needs?

Booking responses may contain attendee or meeting fields that a restricted portal role should not receive. Hiding a field visually does not remove it from the response.

Proposed approach. Define response DTOs per role and review the upstream identity/scopes. Confirm access after membership or assignment changes, including cached responses.

SEC-03 | P2 | Can denied requests be investigated without leaking data?

A generic error is insufficient for support, while full booking payloads in logs can expose personal or meeting information.

Proposed approach. Record a correlation ID, actor, scoped object reference and decision reason. Exclude attendee details, tokens and meeting URLs from routine logs.

Example technical output

Proposed access checks, using seeded identities

```
caller: portal_user_customer_B
requested_booking: seeded_customer_A_booking
local_scope_check: required_before_upstream_response
allowed_result: deny_without_booking_fields
control_case: customer_A_owner_can_read_allowed_fields
log: correlation + actor + decision; no meeting URL
execution_result: TO_VERIFY
```

Delivery plan and ownership

Proposed action plan

Priority	Review item	Accountable role	Acceptance gate
P1	SEC-01	Portal backend owner	A cross-customer fixture returns no booking fields or meeting URL; the authorized owner still works.
P1	SEC-02	Backend + security owner	Restricted roles receive only approved fields; revoked assignment cannot read a cached response.
P2	SEC-03	Security operations	Denied decisions are attributable with no sensitive booking content in logs.

Acceptance checks

Run these checks before accepting the proposed changes.

Acceptance checks

Scenario	Required evidence / expected behavior	Status
Customer B requests customer A seeded booking	No attendee fields, meeting URL or action authority	To verify

Scenario	Required evidence / expected behavior	Status
Assigned owner requests the same booking	Approved fields and intended action remain available	To verify
Assignment is revoked after a cached read	Current authorization is enforced; stale cache does not disclose	To verify
Denied request is inspected in logs	Attributable decision without personal or meeting content	To verify

Rollout and recovery

Use an isolated test setup and approved identities first. Remediate verified issues in small changes, then retest both denied and legitimate flows. A rollback must not restore a missing access check; disable the affected portal action if needed while preserving the safer authorization boundary.

Handover and limits

The report would separate verified issues, assumptions and untested paths, with affected versions and redacted evidence. A focused portal review is not a certificate that every Cal.com integration or application route is secure.

Access and evidence handling

Area	Required control
Test data	Use synthetic tenant identities and bookings; do not probe unrelated accounts.
Secrets	Keep provider credentials server-side and out of screenshots, issue text and report attachments.
Evidence	Deliver minimal redacted request/response pairs through the agreed restricted channel.

Public documentation used

Cal.com: API v2 Access Control - Documents role, membership, PBAC and OAuth scope boundaries.

Cal.com: Get a booking - Documents booking identifiers, authentication and returned attendee/meeting fields.

Read this example online | Related service | Discuss a similar scope

EXAMPLE 06 / 07 | PUBLIC REFERENCE: Polar

Polar: subscription state to product access

Build a billing integration with recoverable local entitlement changes

Independent example based on public documentation. Not commissioned by or affiliated with Polar.

Decision brief

Recommended next decision

Question	Recommendation
Decision	Make the local entitlement decision and its delivery intent durable together.
Why it matters	A valid billing update can still be lost between the local commit and a downstream access change.
Acceptance	Duplicate, late and missed updates converge on the approved access policy with a recovery trail.
Scope boundary	SaaS-owned receiver, database and access adapter; billing policy and merchant responsibilities remain with the client.

The proposed engagement

Proposed brief: a SaaS team uses Polar for billing and needs subscriptions and granted benefits to control access inside its own product. Upserto would implement a signed webhook receiver, customer mapping, local entitlement update and recovery path. The team needs to explain why a customer has access and recover a missed update without blindly granting or removing it.

Inputs required. Polar organization/configuration; external customer ID mapping; entitlement rules; customer-owned API and database.

System boundary and assumptions

Polar documents a Customer State object containing active subscriptions and granted benefits, available by API and through customer.state_changed. The proposed work covers the SaaS-owned receiver, database and access adapter. It does not change Polar's billing engine, taxes or merchant responsibilities. A local outbox would be a proposed implementation choice in the SaaS, not a claim about Polar's internal architecture.

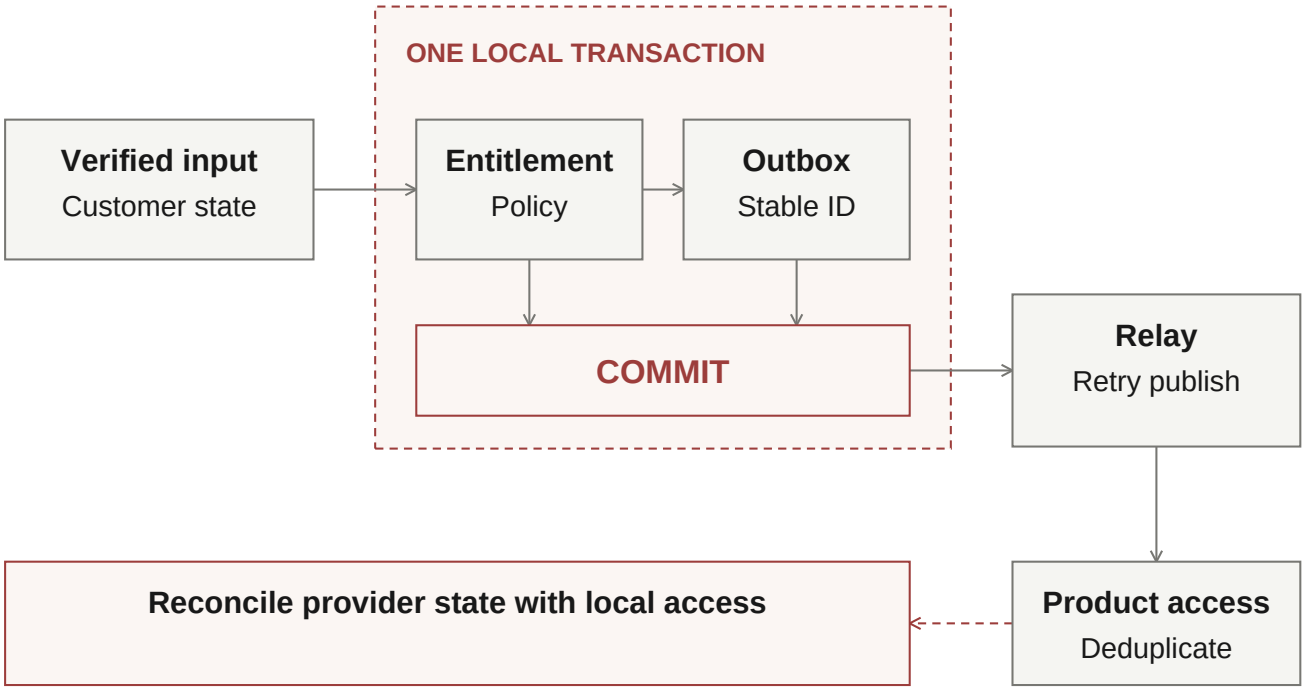
How Upserto would approach it

Agree access policy before coding: which subscription or benefit grants which capability, and how cancellation, expiry and revocation should appear. Use Polar's documented signature helper for the configured SDK/secret format. Serialize processing per customer and refresh current state where needed so a late event cannot blindly overwrite a newer local decision. Keep a reconciliation path for missed notifications.

What the team would receive

- A webhook receiver and versioned billing-to-entitlement mapping.
- A local transaction/outbox or equivalent durable handoff with idempotent access updates.
- Focused failure tests and an operator reconciliation guide.

Keep local consistency separate from external delivery



The SaaS database commits the entitlement decision and delivery intent together. The relay and access adapter still require retry, deduplication and reconciliation.

Accent frame: one local transaction | Dashed: reconciliation

- Validate the Polar event and resolve the current customer state under the agreed access policy.
- Commit the local entitlement decision and outbox intent in one transaction.
- Deliver by stable decision identity; reconcile provider state and product access through an owned correction path.

Starting point and proposed change

Items to inspect in the customer integration, proposed controls and acceptance evidence.

Design comparison

Control	Inspect first	Proposed change	Acceptance evidence
Local handoff	Review database change followed by best-effort publication.	Atomic local decision and durable delivery intent.	Crash leaves a recoverable outbox item.
Event handling	Review whether arrival order directly assigns access.	Current-state policy and stable decision identity.	Duplicate/late fixtures converge on the intended entitlement.
Recovery	Review webhook-health-only monitoring.	Customer-state-to-access reconciliation.	Missed update is found and corrected with an audit trail.

Decisions and approval owners

Decision	Proposed default	Approval owner
Entitlement policy	Agree grant, cancellation, expiry and revocation rules before implementing the mapping.	Product + billing owners
Recovery authority	Separate transport replay from an approved correction to customer access.	SaaS operations owner

Review questions and implementation risks

Priorities identify proposed work: P1 before the affected release or an explicit acceptance decision; P2 with an owner and follow-up milestone.

FT-01 | P1 | Can a local access change lose its downstream update?

If the SaaS commits an entitlement then sends a separate message, a process exit between them can leave another service unaware. This is a design risk to check in the customer implementation.

Proposed approach. Use one local transaction for the decision and an outbox record, or document an equivalent durable handoff. Relay only committed intent.

FT-02 | P1 | Can a duplicate or late notification undo the right access state?

A receipt is a trigger for a decision, not proof that its arrival order is the latest business state. Repeated downstream effects also need a stable identity.

Proposed approach. Record receipt identity, serialize per customer and define a current-state refresh policy. Apply each local decision idempotently in the access adapter.

FT-03 | P2 | Can operations reconcile a missed billing update?

Without a provider-to-local comparison, a missed update may remain invisible even when the receiver and queue look healthy.

Proposed approach. Provide a bounded customer-state reconciliation job with dry-run output and an approved correction path. Separate access-policy choices from transport recovery.

Example technical output

Proposed local transaction boundary

```
verified_event = validate_polar_webhook(raw_body, headers)
customer = resolve_external_customer(verified_event)
state = resolve_current_customer_state(customer)
BEGIN
  persist_receipt_if_new()
  apply_entitlement_policy(state)
  insert_outbox(stable_access_decision_id)
COMMIT
relay_with_retry_and_consumer_deduplication()
execution_result: TO_VERIFY
```

Delivery plan and ownership

Proposed action plan

Priority	Review item	Accountable role	Acceptance gate
P1	FT-01	SaaS backend + database owner	Crash after commit preserves the access decision and a recoverable delivery intent.
P1	FT-02	Billing integration owner	Repeated or reordered fixtures converge on the intended access policy without repeated grants.

Priority	Review item	Accountable role	Acceptance gate
P2	FT-03	SaaS operations	A seeded provider/local mismatch is detected and corrected through a recorded decision.

Acceptance checks

Run these checks before accepting the proposed changes.

Acceptance checks

Scenario	Required evidence / expected behavior	Status
Process stops just after local commit	Entitlement and delivery intent both remain durable	To verify
Same delivery is repeated	Access adapter applies the stable decision without a second effect	To verify
An older notification arrives later	Current-state policy prevents a stale access overwrite	To verify
A notification is missed	Bounded reconciliation detects the local/provider mismatch	To verify

Rollout and recovery

Use Polar's approved sandbox and fixture customers before production. Deploy consumers before enabling a new event producer where contracts change. Start with an agreed product/customer slice, observe local decisions and downstream access together, and keep new grants independently stoppable. Preserve durable intent when pausing delivery.

Handover and limits

Deliver the mapping, database changes, receiver, access adapter, test evidence and reconciliation procedure. Actual access-policy decisions and production acceptance belong to the product owner. No settlement or revenue result is claimed by this proposal.

Access and evidence handling

Area	Required control
Access	Use approved sandbox customers and a scoped webhook endpoint first.
Evidence	Persist receipt and decision references; avoid logging full customer state or webhook secrets.
Correction	Restrict production reconciliation and access corrections to agreed operators with an audit trail.

Public documentation used

Polar: Customer State - Current subscriptions/benefits and customer.state_changed integration.

Polar: Handle webhook deliveries - SDK signature validation, delivery inspection and redelivery.

Polar: Setup webhooks - Configured endpoint and secret-format considerations.

Read this example online | Related service | Discuss a similar scope

EXAMPLE 07 / 07 | PUBLIC REFERENCE: n8n

n8n: decide when to introduce queue mode

An architecture decision for a self-hosted automation team

Independent example based on public documentation. Not commissioned by or affiliated with n8n.

Decision brief

Recommended next decision

Question	Recommendation
Decision	Choose n8n queue mode only after the workload and operating model justify it.
Why it matters	More workers can amplify database or provider contention instead of removing the bottleneck.
Acceptance	The decision names the limiting resource, operational cost and a rehearsed recovery path.
Scope boundary	Selected customer-owned workflows and deployment; capacity or cost savings require measurement.

The proposed engagement

Proposed brief: an operations team runs self-hosted n8n and sees long-running workflows overlap with incoming webhook work. It is considering queue mode. Upserto would measure the actual constraint, compare the current deployment with a main/Redis/worker topology, and prepare an implementation decision with operating costs and a rollback plan. Queue mode would be recommended only if the workload and team can justify the extra components.

Inputs required. Deployment/configuration; representative workflows; execution history; database/Redis constraints; operations coverage.

System boundary and assumptions

The reference is n8n's public queue-mode documentation: a main instance coordinates triggers, Redis carries execution work and workers run executions using shared database state. The proposed assessment covers the customer deployment and selected workflows, not an audit of n8n. Required evidence includes execution duration, concurrency, database waits, binary-data handling and the team's recovery responsibilities.

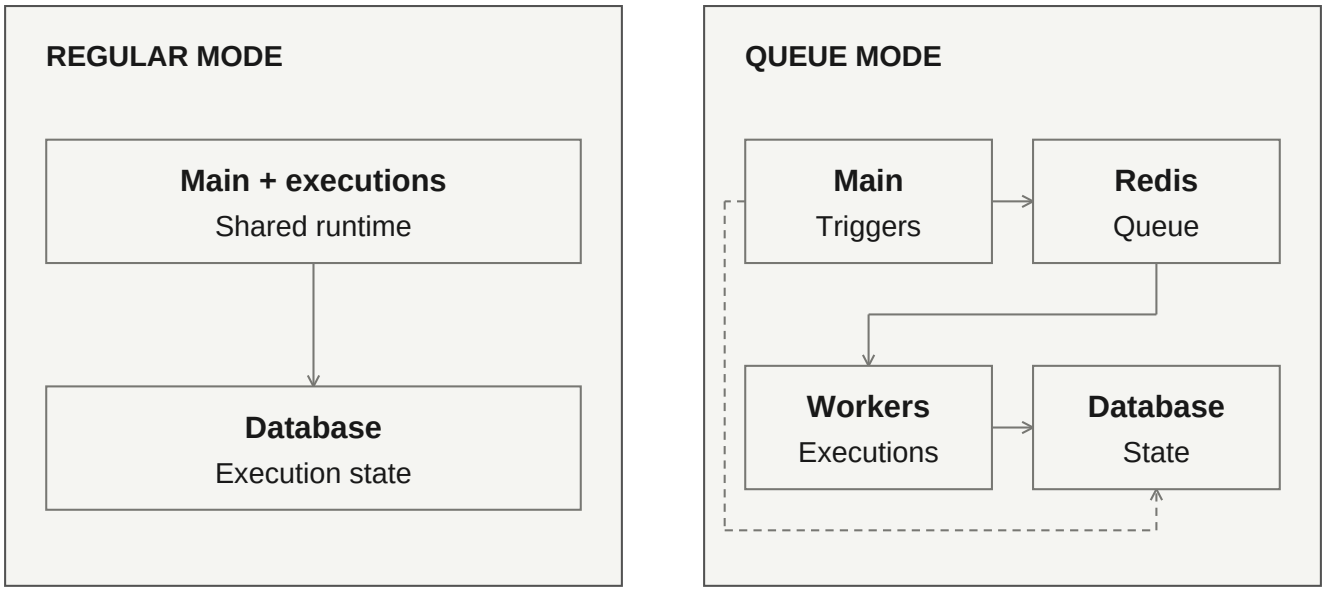
How Upserto would approach it

First separate workflow wait time from CPU, database and provider latency. Compare tuning the existing deployment with introducing workers; do not attribute every slow workflow to worker capacity. Inventory credentials/encryption configuration, binary data and version-specific constraints. Cost the recovery and monitoring work alongside application changes. Record conditions that would reverse the decision.

What the team would receive

- A short architecture decision record with measured evidence and option trade-offs.
- A deployment/dependency plan and capacity assumptions to validate.
- A rehearsal checklist, operational ownership and rollback/queue-drain procedure.

Compare topology against the operating constraint



Fewer components; shared runtime.

Tune the measured bottleneck first.

More components; more operations.

Check credentials + binary data.

DECISION GATE: baseline + recovery rehearsal

Queue mode changes the deployment and recovery responsibilities. The choice should follow a representative workload baseline, not an assumed throughput gain.

Left: existing regular mode | Right: proposed queue-mode option | Dashed: shared database dependency

- Regular mode keeps execution and trigger handling in the main deployment.
- Queue mode introduces Redis and workers, with shared configuration and database dependencies.
- Choose after measuring the constraint and rehearsing credentials, binary-data handling and recovery.

Starting point and proposed change

Items to inspect in the customer integration, proposed controls and acceptance evidence.

Design comparison

Control	Inspect first	Proposed change	Acceptance evidence
Scaling decision	Review a worker proposal without a baseline.	Choose against measured resource and workflow constraints.	Baseline and decision criteria are recorded.
Dependencies	Review application-only implementation estimates.	Include Redis, workers, credentials and binary storage.	Representative worker rehearsal succeeds with the agreed configuration.
Recovery	Review restart-as-recovery assumptions.	Owned pause, drain and resume procedure.	Failure rehearsal records execution state and duplicate-effect handling.

Decisions and approval owners

Decision	Proposed default	Approval owner
Topology	Compare tuning the current instance with main/Redis/worker mode against the same workload.	Engineering lead

Decision	Proposed default	Approval owner
Operational readiness	Name the operator, confirm storage and credentials configuration, and rehearse pause and recovery.	Platform + operations owners

Review questions and implementation risks

Priorities identify proposed work: P1 before the affected release or an explicit acceptance decision; P2 with an owner and follow-up milestone.

ADR-01 | P1 | Is queue capacity the actual bottleneck?

If time is spent waiting on a database lock or external rate limit, extra workers can increase contention instead of resolving it.

Proposed approach. Measure a representative workflow set and separate resource limits from external waits before choosing worker count.

ADR-02 | P1 | Are shared configuration and storage ready for workers?

Queue mode adds dependencies and has documented binary-data storage constraints. A topology diagram alone does not establish that workflows can run on another process.

Proposed approach. Inventory database, Redis, encryption/credentials configuration and binary data. Confirm the deployed version and any edition-specific capabilities before the proposal is costed.

ADR-03 | P2 | Can the team pause and recover the new topology?

More instances introduce more failure and recovery paths. The team needs to know how running and queued work behaves during maintenance.

Proposed approach. Assign an operator, rehearse worker loss and define stop, drain and rollback steps. Do not promise replay correctness without checking workflow side effects.

Example technical output

Proposed architecture decision template

```
decision: pending_workload_and_operational_evidence
option_A: tune_current_deployment
option_B: queue_mode_main_redis_workers
measure: execution_age + concurrency + DB_waits + provider_delay
check: credentials_config + binary_storage + version_constraints
rollout_gate: representative_worker_and_recovery_rehearsal
revisit_when: measured_constraint_or_operating_cost_changes
execution_result: TO_VERIFY
```

Delivery plan and ownership

Proposed action plan

Priority	Review item	Accountable role	Acceptance gate
P1	ADR-01	Engineering lead	A baseline names the limiting resource and the representative workload used.
P1	ADR-02	Platform + workflow owners	A separate worker completes the agreed workflow and its binary-data path in a rehearsal.
P2	ADR-03	Operations owner	A named operator can pause, inspect and recover the agreed execution without an unintended duplicate effect.

Acceptance checks

Run these checks before accepting the proposed changes.

Acceptance checks

Scenario	Required evidence / expected behavior	Status
Representative busy workflow set	Baseline separates queue, database and provider waits	To verify
Workflow runs on a separate worker	Credentials and agreed binary-data behavior work as intended	To verify
Worker stops during an external call	Execution state and duplicate-effect recovery are understood	To verify
Maintenance pause and rollback	Named operator follows the agreed drain/recovery procedure	To verify

Rollout and recovery

Make the decision record before changing topology. If queue mode is justified, rehearse the exact version and dependencies in an isolated environment, then begin with an agreed workflow scope. Define how new triggers pause and existing work drains. Monitor execution age and downstream completion, not only process health.

Handover and limits

The team would receive the option comparison, baseline, configuration plan and operating guide. Queue mode is a documented capability; the right configuration and its benefit depend on the customer's workload. The proposal makes no throughput or cost-saving claim before measurement.

Access and evidence handling

Area	Required control
Access	Review deployment settings and redacted execution metadata before requesting workflow payloads.
Secrets	Keep credential values and encryption keys out of the decision record; reference approved configuration names.
Evidence	Agree retention for execution samples and remove unnecessary personal or customer content.

Public documentation used

n8n: Enable queue mode - Documents main/Redis/worker flow, shared configuration and binary-data storage constraint.

Read this example online | Related service | Discuss a similar scope