ZAHLEN

Chapter 1

Introduction

What Zahlen is, the business problem it solves, and how the platform is organized

Audience

Merchants | Developers | Integration Engineers

Zahlen API User Guide v1.0

Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter purpose

This chapter gives readers the mental model they need before they create credentials or send their first API request. It explains Zahlen in plain language, shows the recurring-payment problem it addresses, and maps the commercial API to the wider platform architecture.

Chapter 1 - Introduction

Learning objectives

By the end of this chapter, you should be able to explain what Zahlen does, describe why payment recovery requires evidence-driven decisions, identify the major platform layers, and trace a payment event through the end-to-end commercial workflow.

1.1 What is Zahlen?

Zahlen is a recovery-intelligence platform for recurring and card-on-file payments. It helps a merchant decide whether a declined payment should be retried, when another authorization attempt should occur, and how the result should be recorded for future analysis.

The platform is not a payment processor and does not replace a merchant's billing system, gateway, acquirer, or issuer. Instead, Zahlen sits beside the payment stack as a decision and evidence layer. It accepts payment-event data, evaluates retry opportunities, returns explainable recommendations, records real-world outcomes, and transforms accumulated evidence into reporting and operational intelligence.

Concept	Meaning
Payment evidence	Facts supplied by the merchant or produced by the payment stack, such as a decline code, issuer BIN, amount, attempt number, processor, timestamp, and merchant reference.
Retry decision	An explicit recommendation describing whether a retry is appropriate and, when available, the recommended retry day, confidence, and reasons.
Retry outcome	The observed result of an attempted retry, including approval or final decline evidence and the identifiers that connect the outcome to the original decision.
Recovery intelligence	Patterns derived from completed evidence, including decline explanations, recovery performance, issuer behavior, and operational recommendations.
Investigation and reporting	Durable, tenant-scoped records that help merchants and operators understand larger trends, exceptions, and system performance.

A simple way to think about Zahlen

Your payment system tells Zahlen what happened. Zahlen recommends what to do next. Your payment system performs the attempt. Then it tells Zahlen what actually happened.

1.2 Who uses Zahlen?

- Merchants use Zahlen to improve the recovery of recurring revenue while reducing unnecessary authorization attempts.
- Developers integrate payment-event, retry-decision, outcome, and webhook endpoints into billing and payment services.
- Integration engineers connect Zahlen to gateways, processors, data pipelines, observability systems, and enterprise controls.
- Operators and analysts use the administrative platform to investigate issuer behavior, monitor classifications, manage watchlists, review recommendations, and supervise investigation runs.

This guide concentrates on the first three audiences. Administrative capabilities are introduced only when they affect the commercial integration or the identifiers developers must preserve.

1.3 The business problem Zahlen solves

Recurring-payment recovery looks simple from a distance: a payment fails, so the merchant tries again. In practice, a decline is evidence of a condition - not a complete explanation of that condition. The same visible decline code can have different operational meanings depending on the issuer, card brand, country, amount, processor, attempt number, billing-cycle position, and recent recovery history.

The fixed retry schedule is the answer

- Immediate retries may repeat the same failure condition before it has had time to change.
- Too many attempts can increase processor costs, create unnecessary issuer traffic, or harm the customer experience.
- Waiting too long can reduce the chance of recovery and delay merchant revenue.
- A single global schedule ignores issuer-level and decline-category differences.
- A recommendation without outcome reporting cannot be measured or improved.

The four operational questions

Concept	Meaning
Should this payment be retried?	Some declines are recoverable, some require customer action, and some should not be repeatedly attempted.
When should the retry occur?	The timing of another attempt can materially affect authorization success.
Why was this recommendation produced?	Developers and operators need reason codes, confidence, policy context, and traceable identifiers.
Did the recommendation work?	The retry outcome must be reported so recovery performance can be measured against the original decision.

The core business value

Zahlen replaces an undifferentiated "retry everything" approach with a tenant-safe, evidence-driven workflow that separates recommendation from execution and connects every recommendation to an observed outcome.

Example scenario

A subscription charge for $29.99 is declined with code 51 on the customer's first attempt. The merchant sends the payment event to Zahlen with a stable event ID, amount in minor units, issuer BIN, card brand, processor, and attempt number. Zahlen returns a decision with a recommendation, confidence, reason detail, and durable identifiers. The merchant follows the approved retry schedule, submits the next authorization through its processor, and reports the result to Zahlen. That completed chain becomes evidence for future decisions and reporting.

1	2	3	4	5
Decline observed	**Evidence sent**	**Decision returned**	**Retry executed**	**Outcome reported**
Merchant payment stack records the event.	Client posts the payment event to Zahlen.	Zahlen provides recommendation and reasons.	Merchant submits the authorized retry.	Observed result closes the learning loop.

1.4 The commercial workflow

The developer experience is organized around a five-stage flow. Each stage produces or consumes identifiers that should be stored in application logs and durable business records.

1	2	3	4	5
Payment Event	**Retry Decision**	**Retry Outcome**	**Investigation Run**	**Reporting**
Submit one or more payment-event records.	Read a generated decision or request one directly.	Report the real processor or settlement result.	Connect ingestion to durable analysis.	Review recovery, usage, and operational trends.

Stage 1 - Payment Event

The payment event is the evidence record. A merchant supplies a stable `event_id` and any available context such as decline code, response code, issuer BIN, amount, currency, attempt number, timestamps, payment token, subscription ID, processor, and metadata. The public request deliberately excludes `tenant_id`; ownership is resolved from the authenticated API key.

Stage 2 - Retry Decision

A decision tells the client what Zahlen recommends. Depending on the selected contract, the response can include the decision value, recommended retry day, confidence, confidence score, reason codes, reason detail, policy source, matched policy ID, request ID, and decision ID. Recommendation is not execution: the merchant remains responsible for submitting the payment attempt through its payment stack.

Stage 3 - Retry Outcome

After the merchant performs the attempt, the outcome endpoint records what actually happened. Reporting an outcome connects the recommendation to observed evidence, such as an approval, final decline, approval code, settled amount, currency, and outcome timestamp.

Stage 4 - Investigation Run

Payment-event ingestion can produce an upload job and an investigation run. These durable resources bridge raw evidence into Recovery Intelligence, issuer monitoring, classifications, and reports. Administrative authorization is separate from ordinary merchant API-key access.

Stage 5 - Reporting

Reporting turns individual events and outcomes into trends: event volume, accepted and rejected records, retry candidates, confidence distributions, decision distributions, processor results, recovery performance, usage, and operational health.

Traceability rule

Preserve `event_id`, `payment_event_batch_id` or `batch_id`, `upload_job_id`, `request_id`, and `decision_id` whenever they are returned. These identifiers connect the client request, Zahlen decision, downstream processor result, support investigation, and audit evidence.

1.5 Platform architecture

Zahlen is organized into layers. The layers separate the public developer contract from merchant administration, operator workflows, durable storage, and governance. This separation lets developers integrate a focused commercial API without depending on internal operator routes.

Layer	Contains	Primary responsibility
Merchant systems	Billing services, subscription platforms, payment orchestrators, gateways, processors, data pipelines, and monitoring tools.	Create evidence, execute payment attempts, and consume responses.
Commercial API	`/v1/payment-events`, retry-decision routes, retry outcomes, webhook subscriptions, health, version, and discovery.	Authenticate merchant requests, validate schemas, return durable identifiers, and enforce commercial controls.

Layer	Contains	Primary responsibility
Decision and ingestion services	Payment-event normalization, policy evaluation, idempotency, batch processing, processor-result correlation, and outcome recording.	Convert evidence into recommendations and durable processing state.
Recovery Intelligence	Recovery Truth, decline explanations, retry analysis, issuer behavior, health snapshots, radar signals, and classifications.	Transform completed evidence into explainable operational intelligence.
Administrative control plane	Operator HTML, administrative JSON APIs, investigation runs, watchlists, recommendations, incidents, queues, and supervisor dashboards.	Support human investigation, governance, and enterprise operations.
Governance and persistence	Tenant-scoped repositories, API keys, usage plans, quotas, audit trail, activity, certification, and durable databases.	Protect ownership, preserve history, and provide evidence for safe operation.

Architecture flow

1	2	3	4	5
Merchant client	**API gateway layer**	**Core services**	**Intelligence layer**	**Portal and reports**
Sends authenticated JSON over HTTPS.	Authenticates, validates, limits, and audits.	Ingest, normalize, decide, and persist.	Builds recovery and issuer evidence.	Expose tenant-scoped results and controls.

Public, administrative, and internal surfaces

Concept	Meaning
Merchant API - `/v1/*`	Used by merchant applications. Merchant-facing calls authenticate with `X-API-Key` unless a route is explicitly public, such as a deployment health check.
Operator HTML - `/admin/*`	Used by authenticated human operators. These pages normally rely on a login session, not a merchant API key.
Administrative JSON - `/v1/admin/*`	Used for governed automation, reporting, supervision, and enterprise administration. Access is separately authorized.
Background services	Workers, supervisors, repositories, and scheduled evaluation processes. These are deployment components rather than public client contracts.

Integration boundary

A merchant integration should depend only on documented public contracts. Do not call internal services or infer administrative access from the existence of a `/v1/admin/*` route.

1.6 Tenant-safe architecture

Tenant isolation is a foundational design rule, not an optional filter. The authenticated API key resolves the merchant, tenant, and actor context. Public request bodies do not control ownership. A caller therefore cannot select another tenant simply by changing JSON, a form field, or a query parameter.

Ownership model

Concept	Meaning
`tenant_id`	The ownership boundary. It determines which records, keys, quotas, audit entries, events, decisions, outcomes, and administrative resources belong together.
`merchant_id`	Business context within the tenant. It is useful for reporting and integration, but it is not a substitute for the authenticated ownership boundary.
`X-API-Key`	The merchant credential used to resolve authorized context for merchant-facing API calls.
Session authentication	The usual authorization mechanism for human-facing administrative HTML pages.

Why this matters to developers

- Do not include a tenant ID in a public request unless the documented schema explicitly defines one for a non-ownership purpose.
- Treat a cross-tenant 404 or empty result as an isolation outcome, not evidence that a filter should be bypassed.
- Keep development, staging, and production keys, base URLs, and databases separate.
- Log durable resource identifiers, but never log the complete API key.
- Expect permission checks to fail closed when tenant context is missing or invalid.

Security principle
Authentication establishes who is calling. Tenant resolution establishes what that caller owns. Authorization establishes which operation the caller may perform. All three checks are required.

1.7 What Zahlen does not do

- Zahlen does not directly move money or replace your payment processor.
- Zahlen does not guarantee that a recommended retry will be approved.
- Zahlen does not make optional evidence appear when the merchant has not supplied or generated it.

- Zahlen does not remove the merchant's responsibility for payment security, privacy, card-network rules, processor contracts, or regulatory compliance.
- Zahlen does not treat a recommendation as an observed result; the outcome must be reported separately.

1.8 Chapter summary

- Zahlen is an evidence and decision layer for recurring-payment recovery.
- The platform helps answer whether to retry, when to retry, why the decision was made, and whether it worked.
- The commercial workflow is Payment Event -> Retry Decision -> Retry Outcome -> Investigation Run -> Reporting.
- The public API is separated from administrative and internal services.
- Tenant ownership comes from authenticated context, not merchant-supplied ownership fields.
- Durable identifiers provide traceability across ingestion, decisions, outcomes, investigations, and audit records.

Next chapter

Chapter 2 - Getting Started explains how to prepare an account, obtain an API key, select an environment and usage plan, and complete the first authenticated request.

Documentation basis: Zahlen API User Guide v1.0, the confirmed 0616A request/response schema export, and the deployment architecture represented by zahlen_deploy_0616A.tar.gz.

ZAHLEN

Chapter 2

Getting Started

Create an account, generate an API key, and select a usage plan

Audience

Merchants | Developers | Integration Engineers

Zahlen API User Guide v1.0

Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter purpose

This chapter prepares a merchant team to begin integrating with Zahlen. It explains account provisioning, environment selection, secure API-key creation, usage-plan selection, and the first connectivity and authenticated checks.

Chapter 2 - Getting Started

Learning objectives

By the end of this chapter, you should be able to identify the people and information needed for onboarding, create or receive a tenant-scoped account, generate and protect an API key, select an appropriate usage plan, and verify that your integration environment is ready.

A successful Zahlen integration begins before the first line of application code. The merchant must establish a tenant, choose an environment, define who may administer credentials, select a usage plan, and decide how identifiers and secrets will be stored. These decisions create the ownership and security foundation for every payment event, retry decision, outcome, and report that follows.

Canonical retry schedule

Zahlen is built around a fixed retry schedule of Day 1, Day 2, Day 6, and Day 14. During onboarding, confirm that the merchant billing workflow can schedule, identify, and report each of these four attempts consistently.

2.1 Before you begin

Gather the following information before requesting production access. A small amount of planning here prevents key-sharing, duplicate identifiers, and environment confusion later.

Concept	Meaning
Merchant organization	Legal or operating name, primary technical contact, support contact, and the team responsible for recurring-payment recovery.
Integration owner	The person accountable for implementation decisions, testing, deployment, and production support.
Environment plan	The development, staging, and production hostnames and the systems that will call each environment.
Expected traffic	Estimated payment-event volume, decision volume, batch sizes, peak rate, and outcome-reporting frequency.
Data mapping	The merchant fields that will populate event IDs, payment tokens, billing cycles, subscriptions, decline codes, issuer details, amounts, timestamps, and processor references.
Secret storage	The secret manager, deployment system, or protected environment-variable mechanism that will store API keys.
Retry orchestration	The service that will execute the fixed Day 1, Day 2, Day 6, and Day 14

Concept	Meaning
	retry schedule and report each observed outcome.

Recommended onboarding roles

Role	Responsibility
Merchant administrator	Creates or approves the tenant account, assigns administrative access, selects the plan, and governs API keys.
Application developer	Builds requests, handles responses, stores identifiers, and implements safe retry behavior.
Integration engineer	Maps payment-system fields, environments, network access, observability, and deployment configuration.
Security reviewer	Approves secret handling, data minimization, audit requirements, and production access.
Operations owner	Monitors outcomes, failed integrations, quota pressure, and investigation-run status after launch.

2.2 Create a Zahlen account

A Zahlen account is provisioned within a tenant. The tenant is the durable ownership boundary for merchant data, credentials, quotas, audit records, and administrative resources. The exact registration screen or approval workflow may vary by deployment, so follow the invitation or onboarding process supplied by the Zahlen administrator.

Account-creation workflow

1	2	3	4	5
Request access	**Tenant provisioned**	**User invited**	**Identity verified**	**Access reviewed**
Provide merchant, contact, environment, and traffic information.	Zahlen creates or confirms the tenant ownership boundary.	An authorized administrator receives account access.	Complete password, session, and any required security setup.	Confirm environment, role, merchant context, and administrative permissions.

What to verify after sign-in

- The displayed organization or tenant is the one assigned to your merchant.
- The hostname clearly identifies development, staging, or production.
- Your user can reach only the administrative pages required for your role.
- The merchant or business context shown in the portal is correct.

- The selected plan and quota information are visible or available from your Zahlen administrator.
- API-key creation is restricted to authorized administrators.

Ownership warning

Do not work around an incorrect or missing tenant by adding a tenant identifier to a form, query string, or JSON body. Tenant ownership must come from authenticated account or API-key context.

Environment separation

Use separate credentials, hostnames, databases, and operational records for development, staging, and production. A development key should never authenticate against production, and a production key should never be copied into a developer laptop, source repository, test fixture, browser script, or chat message.

Concept	Meaning
Development	Used for local or shared integration work with fictional or approved test data. Failures are expected and should not affect production records.
Staging	Used for release-candidate testing, contract verification, load rehearsal, webhook validation, and operational readiness checks.
Production	Used only by approved services with protected secrets, monitored traffic, documented ownership, and an incident response path.

2.3 Generate an API key

Merchant-facing API calls authenticate with the `X-API-Key` header. The key resolves the authorized tenant, merchant, and actor context. It is both a credential and an ownership control, so it must be generated, stored, distributed, rotated, and revoked with the same care as a production password or service credential.

Key-generation workflow

1	2	3	4	5
Open key controls	**Choose context**	**Name the key**	**Create and capture**	**Test and audit**
Use the approved developer portal or administrative key page.	Select the correct environment and merchant scope.	Use a service-oriented label such as billing-worker-prod.	Copy the secret once into an approved secret manager.	Verify authentication, record the key ID, and review activity.

The secret value may be displayed only when the key is created. Zahlen should retain a secure hash for validation rather than storing the original secret in retrievable form. Record the non-secret key

identifier or fingerprint for support and audit purposes, but never record the full key in tickets or application logs.

Recommended key naming

Concept	Meaning
Service	Identify the calling workload, for example `billing-api`, `retry-scheduler`, or `outcome-reporter`.
Environment	Include `dev`, `stage`, or `prod` so a credential cannot be mistaken for another environment.
Owner	Associate the key with a team or system owner, not an individual developer when possible.
Purpose	Use separate keys for independent services so access and activity can be isolated and revoked safely.

Store the key securely

- Place the key in a managed secret store or protected runtime environment variable.
- Restrict read access to the service identity that needs the credential.
- Inject the key at deployment time rather than compiling it into the application.
- Do not expose the key in client-side JavaScript, a browser, or a mobile application.
- Redact the key from exception messages, HTTP tracing, screenshots, and support bundles.
- Create separate keys for separate services and environments instead of sharing one universal key.

Treat the API key as secret material

A request that contains a valid key can act within the key's authorized tenant and merchant context. Possession of the key must therefore be limited, monitored, and revocable.

Configure the client

```
export ZAHLEN_BASE_URL='https://api.example.com'
export ZAHLEN_API_KEY='zk_live_REPLACE_ME'

curl -sS "$ZAHLEN_BASE_URL/v1/version" \
  -H "X-API-Key: $ZAHLEN_API_KEY"
```

Use the base URL provided for the selected environment. Do not hard-code a placeholder hostname or production secret in source code. Some diagnostic routes may be public, but successful access to a health endpoint does not prove that the key is authorized for commercial operations.

2.4 Select a usage plan

Zahlen supports FREE, PROFESSIONAL, and ENTERPRISE plan concepts. Exact quotas, enabled features, retention, support terms, and certification requirements are deployment- or contract-specific. Select the plan based on the merchant workload and governance needs, then read the active limits from the developer portal or administrative controls instead of hard-coding assumed values.

Plan	Typical fit	Integration guidance
FREE	Evaluation, classroom-style learning, prototypes, and low-volume integration testing.	Use realistic error handling from the beginning. Expect conservative quotas and avoid treating the plan as a production guarantee.
PROFESSIONAL	Production merchant integrations with regular event, decision, and outcome traffic.	Monitor usage, use batch endpoints where appropriate, and establish key rotation, alerting, and support ownership.
ENTERPRISE	High-volume, multi-service, or governed deployments requiring custom controls.	Coordinate custom quotas, retention, administrative access, certification evidence, reporting, and operational support.

Plan-selection questions

- How many payment events will be submitted per day and during peak billing windows?
- Will the client use single-event calls, batches, direct decision calls, or all three?
- How quickly must a retry decision be returned to the merchant workflow?
- How many independent services or environments require separate API keys?
- What retention, reporting, audit, and governance evidence is required?
- Will enterprise operators need access to investigation runs or administrative reporting?
- What support response and change-management process is required for production?

Plan names are not capability guarantees

Do not assume that every tenant on the same named plan has identical features or limits. Use the active plan, capability metadata, quota configuration, and explicit API responses as the source of truth.

2.5 Prepare stable identifiers

Before sending test traffic, decide how the merchant system will create and store the identifiers that connect the complete recovery workflow. Stable identifiers make requests repeat-safe, simplify support, and preserve traceability across payment systems and Zahlen.

Concept	Meaning
`event_id`	Created by the merchant. Use one stable identifier for one payment

Concept	Meaning
	event; do not recycle it for unrelated events.
`payment_token` or `token`	A merchant-side token that identifies the payment instrument without exposing a full card number.
`subscription_id`	The merchant subscription or recurring-account reference.
`billing_cycle_id`	The billing-cycle reference used by the legacy decision contract.
`Idempotency-Key`	A stable key for one logical operation on routes that support idempotency. Reuse it only when retrying the same operation.
Returned IDs	Store batch, upload-job, request, and decision IDs returned by Zahlen for tracing and outcome correlation.

Map the fixed retry schedule

The merchant scheduler should represent the four canonical attempts explicitly. Do not collapse them into an ambiguous retry counter. Store both the attempt number and the intended day in the recovery cycle.

Attempt	Scheduled day	Integration meaning
1	**Day 1**	Initial recovery attempt or first scheduled retry in the Zahlen recovery cycle.
2	**Day 2**	Second attempt after the first-day evidence is available.
3	**Day 6**	Third attempt after a longer waiting interval.
4	**Day 14**	Final scheduled attempt in the canonical recovery sequence.

2.6 Verify connectivity

Begin with a health check against the exact environment hostname supplied by Zahlen. A health response verifies DNS, TLS, routing, and service availability. It does not verify merchant authorization.

```
curl -sS "$ZAHLEN_BASE_URL/v1/health" | python -m json.tool
```

A successful response contains service status, service name, API version, and server time. Record the environment and response during deployment verification.

Verify the deployed version

```
curl -sS "$ZAHLEN_BASE_URL/v1/version" \
  -H "X-API-Key: $ZAHLEN_API_KEY" | python -m json.tool
```

The version response can include API version, application version, rules version, and playbook version. These values are useful during support and change-management investigations.

2.7 Send the first authenticated request

Use fictional or approved test data. The following request submits one payment event. The request model requires the `events` array and each event requires a non-empty `event_id`. Unknown top-level event fields are rejected by strict schema validation.

```
curl -sS -X POST "$ZAHLEN_BASE_URL/v1/payment-events" \
 -H 'Content-Type: application/json' \
 -H "X-API-Key: $ZAHLEN_API_KEY" \
 -d '{
  "source": "getting_started",
  "events": [{
   "event_id": "evt_20260616_0001",
   "decline_code": "51",
   "issuer_bin": "411111",
   "amount_minor": 2999,
   "currency": "USD",
   "attempt_number": 1,
   "attempt_day_in_cycle": 1,
   "event_timestamp": "2026-06-16T12:00:00Z",
   "payment_token": "tok_test_001",
   "subscription_id": "sub_test_001"
  }]
}'
```

A successful ingestion response returns identifiers and counts such as `payment_event_batch_id`, `upload_job_id`, received-event count, valid and invalid row counts, status, source, and timestamps. Save these identifiers immediately.

Use only safe test data

Never send a full primary account number, CVV, password, raw bank credential, or production customer secret in a tutorial request. Use merchant-side tokens and the minimum evidence needed for decisioning.

2.8 Readiness checklist

- The correct tenant and merchant account are provisioned.
- Development, staging, and production hostnames are documented.
- An authorized administrator owns API-key creation and revocation.
- The API key is stored in a secret manager and absent from source control.
- The active usage plan and current quotas are known.
- Event, token, subscription, billing-cycle, and idempotency identifiers are defined.
- The billing workflow can execute and report Day 1, Day 2, Day 6, and Day 14 attempts.

- Health and version checks succeed in the selected environment.
- The first authenticated payment-event request succeeds with fictional test data.
- Returned batch and upload-job identifiers appear in application logs.

2.9 Common onboarding mistakes

Concept	Meaning
Using one key everywhere	Creates a large blast radius and makes service-level activity difficult to audit. Use separate keys by environment and workload.
Hard-coding plan limits	Plan configuration may differ by deployment or contract. Read current quota and capability information.
Treating health as authorization	A public or reachable health endpoint proves connectivity, not permission to use merchant routes.
Sending ownership fields	Do not try to select a tenant in the request body. Ownership comes from authenticated context.
Logging secrets	HTTP debugging and exception traces can expose `X-API-Key`. Redact authorization material.
Losing returned IDs	Without batch, upload-job, request, and decision IDs, support and outcome correlation become difficult.
Ignoring the fixed schedule	The recovery workflow must preserve the canonical Day 1, Day 2, Day 6, and Day 14 attempt sequence.

2.10 Chapter summary

- A Zahlen account is provisioned within a tenant, which is the ownership boundary for data and controls.
- Merchant-facing calls use a securely generated API key in the `X-API-Key` header.
- Keys should be separated by environment and service, stored in a secret manager, and logged only by safe identifier or fingerprint.
- Stable identifiers connect payment events, decisions, outcomes, investigation runs, and support records.
- The merchant integration must support Zahlen's fixed retry schedule: Day 1, Day 2, Day 6, and Day 14.
- Connectivity and authenticated ingestion should be verified before broader integration work begins.

Next chapter
Chapter 3 - Authentication explains the `X-API-Key` header in depth, shows authenticated request patterns, and provides security and key-rotation recommendations.

ZAHLEN

Chapter 3

Authentication

API key header, example requests, and security recommendations

Audience

Merchants | Developers | Integration Engineers

Zahlen API User Guide v1.0

Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter 3 - Authentication

Chapter purpose

This chapter explains how merchant applications authenticate to Zahlen, how the X-API-Key header establishes tenant ownership, how to send authenticated requests, and how to protect, rotate, monitor, and revoke credentials safely.

Learning objectives

By the end of this chapter, you should be able to add the correct authentication header, distinguish public and administrative access, test credentials safely, implement secure secret handling, and rotate a key without interrupting production traffic.

Every merchant-facing Zahlen request begins with authenticated context. The API key does more than prove that a caller knows a secret: it resolves the authorized merchant, tenant, and actor context used for ownership checks, quota enforcement, audit records, and response visibility. For that reason, authentication is part of the data-isolation design, not merely a login step.

Core ownership rule

Do not send tenant_id as a way to select ownership. Zahlen derives tenant ownership from the authenticated API key. A request body, query parameter, or form field must never override that boundary.

3.1 Authentication model

Surface	Typical authentication	Purpose
Merchant API - /v1/*	X-API-Key	Payment events, retry decisions, retry outcomes, webhooks, and other approved commercial operations.
Operator HTML - /admin/*	Authenticated browser session	Human administration, monitoring, investigation, and operational action.
Administrative JSON - /v1/admin/*	Administrative context	Governance, reporting, automation, and enterprise administration.
Internal workers	Service configuration	Background evaluation, persistence, and dispatch.

A merchant key should be used only on merchant-facing routes approved for that key. It should not be assumed to grant access to operator pages or administrative JSON endpoints. Authorization remains endpoint-specific even after authentication succeeds.

Authentication and authorization are different

Question	Authentication answers	Authorization answers
Who is calling?	Which API key was presented and successfully validated?	Is the resolved identity allowed to use this endpoint and resource?
Whose data is visible?	Which tenant and merchant context belong to the key?	Does the requested object belong to that tenant and fall within the key's permitted scope?
What happens on failure?	Usually HTTP 401 for missing or invalid credentials.	Usually HTTP 403 for a valid identity that lacks permission.

3.2 The X-API-Key header

Send the merchant API key in the HTTP request header named X-API-Key. Header names are case-insensitive under HTTP, but integrations should use the documented spelling consistently.

```
X-API-Key: zk_live_REPLACE_ME
```

Header	Required?	Purpose
X-API-Key	Yes for merchant-facing authenticated routes	Authenticates the caller and resolves tenant, merchant, and actor context.
Content-Type: application/json	Yes for JSON request bodies	Tells the server that the body is JSON.
Accept: application/json	Recommended	Makes the expected response representation explicit.
Idempotency-Key	Where supported	Identifies one logical write operation so a safe replay can be recognized.
User-Agent	Recommended	Identifies the calling application and version for diagnostics.

Where the key belongs

- Put the key in an HTTP header, not in the URL, query string, JSON body, or form field.
- Send the key only over HTTPS.
- Load the key from a secret manager or protected runtime environment variable.
- Keep key handling on trusted server-side infrastructure.
- Do not expose the key to browser JavaScript, mobile application bundles, public API clients, or shared screenshots.

Never place a key in a URL

URLs can be captured in browser history, reverse-proxy logs, analytics systems, support tools, and referrer headers. Authentication secrets belong in protected headers.

Key resolution sequence

1	2	3	4	5
Receive header	**Validate secret**	**Resolve context**	**Enforce access**	**Audit request**
The server reads X-API-Key from the request.	The submitted value is checked against secure key records.	The key maps to tenant, merchant, actor, status, and policy.	Endpoint, plan, quota, and ownership checks are applied.	The request is correlated with key identity and result.

3.3 Example authenticated requests

The examples in this section use fictional hosts, identifiers, and credentials. Replace them with values supplied for your Zahlen environment. Never paste a live key into documentation, source control, terminal recordings, or support tickets.

Health check without merchant authentication

```
curl -sS https://api.example.com/v1/health \
  -H 'Accept: application/json'
```

A successful health response confirms connectivity and API availability. It does not prove that a merchant API key is valid or authorized for commercial endpoints.

Authenticated payment-event request

```
curl -sS -X POST https://api.example.com/v1/payment-events \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -H 'X-API-Key: zk_live_REPLACE_ME' \
  -d '{
    "events": [{
      "event_id": "evt_20260616_0001",
      "decline_code": "51",
      "issuer_bin": "411111",
      "amount_minor": 2999,
      "currency": "USD",
      "attempt_number": 1,
      "event_timestamp": "2026-06-16T12:00:00Z"
    }]
  }'
```

Authenticated retry-decision request with idempotency

```
curl -sS -X POST https://api.example.com/v1/_next/retry-decision \
  -H 'Content-Type: application/json' \
  -H 'Accept: application/json' \
  -H 'X-API-Key: zk_live_REPLACE_ME' \
  -H 'Idempotency-Key: order-8842-attempt-2' \
  -d '{
    "attempt_number": 2,
    "decline_code": "51",
    "issuer_bin": "411111",
    "amount_minor": 2999,
    "currency": "USD"
  }'
```

3.4 Application examples

Python with requests

```
import os
import requests

base_url = os.environ['ZAHLEN_BASE_URL'].rstrip('/')
api_key = os.environ['ZAHLEN_API_KEY']

headers = {
    'X-API-Key': api_key,
    'Content-Type': 'application/json',
    'Accept': 'application/json',
    'User-Agent': 'merchant-billing/1.0',
}

payload = {
    'events': [{
        'event_id': 'evt_20260616_0001',
        'decline_code': '51',
        'amount_minor': 2999,
        'currency': 'USD',
        'attempt_number': 1,
    }]
}

response = requests.post(
    f'{base_url}/v1/payment-events',
    headers=headers,
    json=payload,
    timeout=20,
)
response.raise_for_status()
result = response.json()
print(result['upload_job_id'])
```

JavaScript on a trusted server

```
const baseUrl = process.env.ZAHLEN_BASE_URL.replace(/\/$/, '');
const apiKey = process.env.ZAHLEN_API_KEY;

const response = await fetch(`${baseUrl}/v1/_next/retry-decision`, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json',
    'Accept': 'application/json',
    'X-API-Key': apiKey,
    'Idempotency-Key': 'order-8842-attempt-2',
    'User-Agent': 'merchant-recovery/1.0'
  },
  body: JSON.stringify({
    attempt_number: 2,
    decline_code: '51',
```

```
    issuer_bin: '411111',
    amount_minor: 2999,
    currency: 'USD'
  })
});

if (!response.ok) {
  throw new Error(`Zahlen HTTP ${response.status}`);
}

const result = await response.json();
```

Server-side only

The JavaScript example is intended for Node.js or another trusted server runtime. Never put a production Zahlen API key in frontend JavaScript delivered to a browser.

3.5 Security recommendations

Treat every API key as a high-value production secret. A valid key can authorize requests, expose tenant-scoped resources, consume quota, create audit activity, and potentially influence payment-recovery operations.

Control	Recommended practice	Why it matters
Secret storage	Use a managed secret store, encrypted deployment secret, or protected environment variable.	Reduces accidental exposure and centralizes access control.
Environment separation	Use different keys for development, staging, and production.	Prevents test traffic and compromised non-production systems from reaching production.
Service separation	Issue independent keys to independent workloads when policy permits.	Limits blast radius and improves attribution.
Least privilege	Enable only approved endpoints and capabilities.	Reduces damage from misuse.
Logging	Log key ID or fingerprint, never the complete secret.	Preserves diagnostics without leaking credentials.
Transport	Require HTTPS and validate certificates.	Protects the key and request data in transit.
Rotation	Rotate on a schedule and after personnel or system changes.	Reduces long-lived exposure.
Revocation	Revoke immediately after confirmed compromise.	Stops further authenticated use.
Monitoring	Alert on authentication spikes, unusual endpoints, geography, or volume.	Detects misuse and integration defects early.

Do not log secrets

- Redact X-API-Key in application, proxy, and observability logs.
- Disable verbose HTTP tracing in production unless headers are safely filtered.

- Ensure exception reporters and APM agents do not capture authentication headers.
- Use a non-secret key ID, last-four display, or approved fingerprint for support correlation.
- Scrub copied curl commands before sharing them.

Do not send prohibited payment data

Authentication secures access to Zahlen, but it does not make every payload appropriate. Use merchant-side payment tokens and the minimum evidence needed for decisioning. Do not submit full primary account numbers, CVV values, passwords, raw bank credentials, or secrets inside metadata objects.

3.6 Key rotation without downtime

A safe rotation overlaps the old and new credentials for a controlled period. Do not revoke the old key before every production instance can authenticate with the replacement.

1	2	3	4	5
Create replacement Generate a new key in the same approved tenant, merchant, environment, and scope.	**Store securely** Place the new secret in the secret manager without removing the old one.	**Deploy gradually** Update services and instances using ordinary release controls.	**Verify traffic** Confirm successful requests, audit activity, and expected key identity.	**Revoke old key** Remove the previous key after all callers have migrated.

Phase	Old key	New key	Operator check
Before rotation	Active	Not created	Confirm current usage and owners.
Overlap	Active	Active	Watch traffic move to the new key.
Cutover complete	Active but unused	Active	Confirm no legitimate calls still use the old key.
After revocation	Revoked	Active	Alert on any attempted use of the retired key.

When to rotate immediately

- The secret appears in a public or shared repository.
- The key is pasted into a ticket, chat, email, document, screenshot, or terminal recording.
- A device or service containing the key is compromised.
- An employee or vendor with secret access leaves the role.
- Authentication activity is inconsistent with expected services, endpoints, volume, or time windows.
- The key has exceeded the organization's maximum credential age.

Compromise response

Do not wait for proof of malicious use when a production key is exposed. Create a replacement, deploy it, revoke the exposed key, review audit records, and preserve incident evidence.

3.7 Authentication and authorization failures

HTTP status	Likely meaning	Client response
401 Unauthorized	X-API-Key is missing, malformed, unknown, expired, or revoked.	Check the header, secret source, environment, and key status. Do not retry rapidly.
403 Forbidden	The key is valid but not permitted for the route, capability, plan, or resource.	Check endpoint authorization, plan, scope, and administrative boundary.
404 Not Found	The resource does not exist or is not visible in the resolved tenant context.	Verify the tenant-scoped identifier; do not weaken ownership filters.
409 Conflict	An idempotency key conflicts with a different request or resource state.	Compare the original logical operation and payload.
429 Too Many Requests	The tenant has reached a rate limit or quota.	Back off, honor Retry-After when present, and preserve idempotency for the same operation.

Diagnostic sequence

1	2	3	4	5
Check hostname Verify development, staging, or production base URL.	**Check header** Confirm X-API-Key is present once and not surrounded by unintended quotes.	**Check secret source** Confirm the deployed secret version and application configuration.	**Check status** Verify the key is active, unexpired, and associated with the expected context.	**Check authorization** Review plan, endpoint policy, audit records, and tenant ownership.

Fail closed

A missing tenant context, unresolved key, or failed ownership check should deny access. Never fall back to a default production tenant to make a request succeed.

Example error-safe client behavior

```
if response.status_code == 401:
    raise RuntimeError('Zahlen authentication failed; check key and environment')
elif response.status_code == 403:
    raise RuntimeError('Zahlen authorization denied; check route and plan')
elif response.status_code == 429:
    # Back off. Use Retry-After when provided.
    # Reuse the same Idempotency-Key for the same logical operation.
    schedule_retry_with_jitter(response.headers.get('Retry-After'))
else:
    response.raise_for_status()
```

3.8 Authentication readiness checklist

Check	Ready when
Environment	The base URL and key belong to the same intended environment.
Tenant context	The key resolves the approved tenant and merchant.
Secret storage	The full key exists only in approved protected secret storage.
Source control	No live key appears in repositories, examples, test snapshots, or documentation.
Transport	All calls use HTTPS and certificate validation remains enabled.
Client architecture	Authentication is performed by trusted server-side code.
Idempotency	Supported write operations use stable logical idempotency keys.
Logging	Authentication headers are redacted; safe key identity is logged.
Rotation	A documented overlap-and-revoke procedure exists and has been tested.
Monitoring	Alerts cover authentication failures, revoked-key use, and unusual traffic.
Incident response	Owners know how to replace and revoke a compromised key.
Schedule integrity	Recovery workflows preserve the fixed Day 1, Day 2, Day 6, and Day 14 schedule.

3.9 Chapter summary

- Merchant-facing Zahlen requests authenticate with the X-API-Key header.
- The API key resolves tenant, merchant, and actor context; tenant ownership is not selected by request JSON.
- Authentication proves identity, while authorization decides whether that identity may use a route or resource.
- Keys belong in protected server-side secret storage and must never appear in URLs, browser code, logs, or shared documents.
- Use Idempotency-Key where supported for safe replays of one logical write operation.
- Rotate keys with overlap, verify the replacement, then revoke the retired credential.
- Treat 401, 403, 404, 409, and 429 as different conditions requiring different responses.
- Authentication enables access but does not alter Zahlen's fixed Day 1, Day 2, Day 6, and Day 14 retry schedule.

Next chapter
Chapter 4 - Usage Plans explains the FREE, PROFESSIONAL, and ENTERPRISE plan concepts and how clients should interpret plan-assigned capabilities without hard-coding deployment-specific policy.

ZAHLEN

Chapter 4

Usage Plans

FREE, PROFESSIONAL, and ENTERPRISE

Audience

Merchants | Developers | Integration Engineers

Zahlen API User Guide v1.0

Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter 4 - Usage Plans

Chapter purpose

This chapter explains how Zahlen usage plans organize commercial access, capacity, support, and governance. It describes the intended fit of FREE, PROFESSIONAL, and ENTERPRISE plans without inventing universal quotas that may differ by deployment or contract.

Learning objectives

By the end of this chapter, you should be able to compare the three plan families, select an appropriate starting plan, understand which settings remain configurable, and design a client that does not hard-code assumptions about plan capabilities.

A usage plan is the commercial policy assigned to a tenant. It can influence request quotas, short-window rate limits, enabled features, retention, support expectations, reporting, certification requirements, and developer-portal visibility. The plan belongs to the authenticated tenant context; it is not selected by sending a plan name in an API request.

Important contract rule

FREE, PROFESSIONAL, and ENTERPRISE are plan concepts. Exact numeric quotas, retention periods, feature flags, and support terms are deployment-specific. Read current plan and quota information from the developer portal or approved administrative interface.

4.1 How plans fit the commercial workflow

Workflow stage	Plan-controlled concern	Client responsibility
Payment Event	Ingestion volume, batch policy, retention, reporting access	Submit valid tenant-owned evidence and store returned identifiers.
Retry Decision	Decision request capacity, enabled contract, latency expectations	Use the approved decision contract and stable idempotency semantics.
Retry Outcome	Outcome volume and retention	Report actual authorization or settlement evidence promptly.
Investigation Run	Administrative visibility and processing scale	Track upload_job_id and use approved administrative access.
Reporting	Dashboard, export, and commercial reporting access	Consume only capabilities explicitly enabled for the tenant.

Plan names do not replace capability checks

Two tenants with the same plan name may have different contractual limits or optional features. A client must respond to explicit API behavior and capability metadata rather than assuming that every tenant has identical settings.

4.2 Plan overview

Plan	Typical fit	Primary goal	Integration guidance
FREE	Evaluation, learning, proof of concept, and low-volume prototyping	Validate the workflow before production commitment	Build correct authentication, validation, 429 handling, and idempotency from the first prototype.
PROFESSIONAL	Production merchant workloads with regular payment-event and decision traffic	Operate a dependable commercial integration	Monitor usage, batch responsibly, report outcomes, and establish production support procedures.
ENTERPRISE	High-volume, multi-team, governed, or contract-specific deployments	Scale with formal controls and operational governance	Coordinate custom capacity, retention, access, reporting, certification, and support requirements.

No unlimited plan

A higher plan does not remove schema validation, tenant isolation, endpoint authorization, safe retry requirements, or operational safeguards. Every plan remains subject to technical and governance controls.

4.3 FREE plan

The FREE plan is intended for evaluation and low-volume prototyping. It lets a development team learn the API, validate payload mapping, test the end-to-end recovery workflow, and estimate future usage before moving production traffic.

Good uses for FREE

- Build and test merchant-side request models.
- Validate X-API-Key handling in a non-production environment.
- Submit representative fictional or properly tokenized payment events.
- Test the fixed Zahlen retry schedule of Day 1, Day 2, Day 6, and Day 14.
- Practice outcome reporting and identifier correlation.
- Implement 401, 403, 422, 429, and transient 5xx handling.
- Run contract and smoke tests before production onboarding.

FREE integration expectations

Expectation	Recommended approach
Traffic	Use small, representative test datasets rather than production-scale loads.
Quotas	Assume conservative limits and display quota exhaustion clearly to developers.
Availability	Treat the environment as an evaluation surface unless a separate service commitment is documented.
Data	Use fictional examples or approved tokenized test data.
Migration	Keep configuration external so the same client can move to PROFESSIONAL without code changes.

Prototype correctly

Do not postpone error handling, idempotency, secret management, or tenant-safe identifiers because traffic is small. A prototype often becomes the foundation of the production integration.

4.4 PROFESSIONAL plan

The PROFESSIONAL plan is the normal fit for production merchant workloads. It supports sustained use of the commercial workflow while retaining clear tenant-scoped controls, usage monitoring, quotas, and operational accountability.

Production responsibilities

- Use separate production keys and base URLs.
- Monitor request volume, latency, authentication failures, 429 responses, and outcome-reporting lag.
- Batch payment events only when batching improves reliability and does not delay time-sensitive processing.
- Preserve event_id, batch_id, upload_job_id, request_id, and decision_id in application logs.
- Implement bounded retries with exponential backoff and jitter.
- Rotate credentials without downtime and review key activity after cutover.
- Maintain support ownership for both merchant-side and Zahlen-side incidents.

Typical PROFESSIONAL operating pattern

1	2	3	4	5
Ingest	**Decide**	**Execute**	**Report**	**Monitor**
Send single events or controlled batches.	Request or retrieve retry decisions.	Apply the fixed Day 1, 2, 6, and 14 schedule as directed.	Send actual retry outcomes.	Review usage, errors, and unresolved processing.

Operational area	What to monitor	Escalation signal
Authentication	401 rate and revoked-key attempts	Sudden spike or use of a retired key.
Capacity	Request rate, quota utilization, and 429 rate	Sustained throttling or unexpected growth.
Decisioning	Latency, error rate, and idempotent replay	Rising 5xx rate or duplicate logical operations.
Outcomes	Reporting lag and missing correlations	Decisions without observed outcomes.
Webhooks	Delivery failures and consumer retries	Repeated callback failure or growing backlog.

4.5 ENTERPRISE plan

The ENTERPRISE plan is designed for high-volume or highly governed deployments. It may combine custom capacity with formal operational controls, enhanced reporting, environment isolation, certification evidence, support coordination, and contract-specific retention or feature enablement.

Common ENTERPRISE requirements

- Custom request and quota policy based on measured workload.
- Multiple services, business units, merchants, or environments with separate credentials.
- Formal key-rotation, revocation, and audit procedures.
- Administrative reporting and tenant-usage visibility.
- Governance certification and negative tenant-isolation tests.
- Controlled access to investigation runs or administrative integrations.
- Defined retention, export, incident-response, and support expectations.
- Capacity planning for large payment-event batches and sustained decision traffic.

Enterprise onboarding questions

Question	Why it matters
What is the peak and average request rate?	Capacity should be based on measured traffic, not only monthly totals.
How many environments and services need keys?	Separate credentials improve isolation, attribution, and rotation.
Which administrative capabilities are required?	Merchant keys do not automatically authorize /v1/admin/* routes.
What evidence and retention rules apply?	Audit, outcome, and investigation records may have contractual requirements.
What support and escalation model is needed?	Operational ownership must be clear before production launch.
Which capabilities need certification?	Enterprise governance should verify controls through evidence and tests.

Enterprise is governed, not merely larger

The defining difference is not only higher volume. ENTERPRISE is appropriate when scale, contractual controls, security review, operational governance, or administrative integration require a coordinated deployment model.

4.6 Selecting a usage plan

Choose a plan from expected behavior and operational requirements, not from the plan name alone. Start with the lowest plan that safely supports the workload, then move when measured usage or governance needs justify the change.

1	2	3	4	5
Estimate	**Classify**	**Confirm**	**Test**	**Approve**
Measure event, decision, outcome, and webhook volume.	Identify prototype, production, or governed deployment needs.	Review actual plan limits and enabled capabilities.	Run representative load and failure scenarios.	Document commercial and operational ownership.

Decision factor	FREE	PROFESSIONAL	ENTERPRISE
Lifecycle stage	Evaluation	Production	Production at scale or under formal governance
Traffic profile	Low and experimental	Regular merchant workload	High, bursty, multi-team, or custom
Support model	Self-guided or basic onboarding	Defined production support	Coordinated enterprise support and escalation
Governance	Basic tenant-safe controls	Production controls and monitoring	Formal certification, evidence, and contract policy
Configuration	Conservative defaults	Production plan settings	Custom or contract-specific settings

Questions to answer before selection

- How many payment events will be submitted per hour, day, and month?
- Will decisions be requested synchronously, read after ingestion, or both?
- How quickly will outcomes be reported?
- What peak burst is expected during billing cycles?
- How many environments and independent services need credentials?
- Are administrative investigation, reporting, or governance capabilities required?
- What retention, audit, export, and support commitments are necessary?

4.7 Plans, quotas, and rate limits

A usage plan and a quota are related but not identical. The plan is the commercial policy assignment. Quotas define longer-window usage allowances, while rate limits control short-window request pressure. Endpoint schema limits, such as maximum batch size, remain technical contract rules independent of the plan.

Control	Purpose	Example client response
Plan	Defines commercial policy and available capabilities	Read current assignment; do not infer undocumented features.
Quota	Limits usage over a billing or policy period	Track utilization and handle exhaustion without data loss.
Rate limit	Controls request volume over a short window	Back off and honor Retry-After when present.
Schema limit	Defines valid request shape or batch size	Split requests before sending; do not retry an invalid payload.
Authorization	Controls endpoint and resource access	Use only approved routes for the authenticated identity.

Schema limits remain fixed

Payment-event ingestion accepts 1 to 10,000 events per request, while the legacy batch retry-decision request accepts up to 500 events. These are request-schema limits, not promised throughput and not automatically increased by a higher plan.

Client response to plan or capability denial

```
if response.status_code == 403:
    # Authenticated, but the capability is not enabled or permitted.
    raise CapabilityNotEnabled(response.json())


if response.status_code == 429:
    # Capacity or quota enforcement. Use bounded backoff.
    retry_after = response.headers.get('Retry-After')
```

4.8 Changing plans safely

A plan change should normally be a configuration and commercial change, not an application rewrite. Keep the base URL, credentials, batching behavior, and capability flags external to code so changes can be tested and rolled out safely.

1	2	3	4	5
Review usage	**Confirm contract**	**Test behavior**	**Activate**	**Monitor**
Confirm actual volume, peaks, 429 activity, and growth.	Document new limits, features, retention, and support terms.	Validate capability and quota metadata in staging.	Apply the plan through approved administration.	Verify traffic, audit records, and quota behavior after change.

Do not assume after an upgrade

- That every endpoint is newly authorized.
- That all quotas are unlimited.
- That schema batch limits changed.
- That administrative routes accept a merchant API key.
- That retention applies retroactively.
- That existing clients can stop handling 429 or 403 responses.

4.9 Developer readiness checklist

Check	Ready when
Plan assignment	The tenant has an approved FREE, PROFESSIONAL, or ENTERPRISE assignment.
Capabilities	Required endpoints and optional features are explicitly confirmed.
Quotas	Current values and reset behavior are available to operators and developers.
Error handling	The client handles 403, 422, 429, and transient 5xx correctly.
Monitoring	Usage, throttling, authentication, latency, and outcome lag are visible.
Secrets	Environment-specific keys are stored and rotated securely.
Identifiers	Events, decisions, outcomes, and jobs can be correlated end to end.
Retry schedule	Operational systems support the fixed Day 1, Day 2, Day 6, and Day 14 schedule.
Support	Owners and escalation paths are documented before production use.

Chapter summary

FREE supports disciplined evaluation, PROFESSIONAL supports dependable production use, and ENTERPRISE supports scale plus formal governance. In every plan, the client must rely on explicit capabilities and current policy rather than hard-coded assumptions.

ZAHLEN

Chapter 5

Rate Limits & Quotas

Request limits, quota exhaustion, and HTTP 429 handling

Audience

Merchants | Developers | Integration Engineers

Zahlen API User Guide v1.0

Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter 5 - Rate Limits & Quotas

Chapter purpose

This chapter explains how Zahlen protects runtime capacity and contracted usage, how request-size limits differ from rate limits and quotas, and how a client should respond safely when the platform returns HTTP 429.

Learning objectives

By the end of this chapter, you should be able to distinguish schema limits, short-window rate limits, and longer-window quotas; identify quota-exhaustion behavior; and implement bounded, idempotent retries without creating a retry storm.

Zahlen applies controls at the authenticated tenant boundary. Rate limits protect the service during short periods of high request pressure. Quotas protect usage over longer policy or billing periods. Request-schema limits define the largest valid payload accepted by a specific endpoint. These controls solve different problems and must not be treated as interchangeable.

Tenant-scoped enforcement

A request is evaluated in the tenant context resolved from X-API-Key. A client must not send tenant_id to obtain a different limit or quota. Capacity and usage policy follow the authenticated tenant and its assigned plan.

5.1 Three different kinds of limits

Control	Time horizon	What it protects	Typical client action
Request or schema limit	One request	Payload validity and processing safety	Reduce or split the payload before sending.
Rate limit	Seconds or minutes	Short-window service capacity	Pause and retry later with bounded backoff and jitter.
Quota	Hours, days, month, or contract period	Tenant usage allowance	Stop uncontrolled retrying, preserve work, and contact the plan owner if capacity is exhausted.
Authorization or capability restriction	Until policy changes	Endpoint and feature access	Do not retry automatically; verify plan, role, and contract enablement.

Do not confuse 422 with 429

A payload that exceeds a schema limit is a client-validation problem and may return a validation response such as HTTP 422. HTTP 429 means an otherwise valid request was refused because an active rate or quota control was reached.

5.2 Request limits

Request limits are part of the endpoint contract. They describe the valid size or shape of one API call. A larger usage plan does not automatically expand these limits, because the server model still validates each request against the published schema.

Endpoint or resource	Confirmed request limit	What the client should do
POST /v1/payment-events	events must contain 1 to 10,000 items	Use one event or split larger datasets into multiple controlled requests.
POST /v1/payment-events/batch	events must contain 1 to 10,000 items	Choose a batch size that balances throughput, latency, memory, and replay safety.
POST /v1/retry-decision/batch	up to 500 legacy decision events	Split larger legacy decision workloads into groups of 500 or fewer.
GET batch resources	limit is 1 through 1,000 when supplied; offset must be 0 or greater	Paginate until has_more is false or all expected records are returned.
Webhook subscription create	1 to 20 event types; callback URL length 8 to 2,048	Validate locally before sending the subscription request.

Choosing a practical payment-event batch size

The maximum accepted batch size is not a recommended default. A smaller batch is often easier to retry, observe, and reconcile. Choose a batch size by testing representative payloads and measuring request duration, response size, failure recovery, and downstream processing time.

Batch characteristic	Smaller batches	Larger batches
Failure scope	Fewer events affected by one request failure	More events require replay or reconciliation
Network overhead	More HTTP requests	Lower per-event HTTP overhead
Latency	Events begin processing sooner	Client may wait longer to assemble and transmit
Correlation	More batch IDs to track	Fewer batch IDs, but larger failure domain
Memory and serialization	Lower client and server memory pressure	Higher memory and serialization cost

```
MAX_EVENTS_PER_REQUEST = 10_000

def chunks(items, size=1000):
    for start in range(0, len(items), size):
        yield items[start:start + size]

for event_batch in chunks(events, size=1000):
    submit_payment_events(event_batch)
```

Preserve identifiers while splitting

Each payment event should retain a stable, merchant-generated event_id. Splitting one source dataset into multiple API batches must not change the identity of the underlying events.

5.3 Rate limits

A rate limit controls how much traffic a tenant may send during a short window. The exact window and numeric allowance are deployment- and plan-specific. Clients should therefore rely on explicit responses and current portal or administrative information rather than hard-code assumed values.

Common causes of rate-limit pressure

- A billing-cycle burst sends many requests at the same second.
- Multiple application instances use the same tenant credentials without shared throttling.
- A retry loop immediately repeats requests after every failure.
- A worker backlog is released all at once after an outage.
- Health checks or polling run more frequently than required.
- A compromised key or programming error creates unexpected traffic.

Client-side traffic shaping

Technique	Purpose	Implementation note
Concurrency limit	Caps the number of requests in flight	Use a semaphore or worker pool per environment and tenant.
Token bucket or leaky bucket	Smooths bursts over time	Share state across instances when they use the same tenant quota.
Queue with bounded workers	Prevents sudden release of a large backlog	Prioritize time-sensitive decision and outcome work appropriately.
Adaptive batch size	Reduces request count when safe	Do not increase batch size beyond endpoint schema limits.
Circuit breaker	Stops repeated calls during sustained failure	Open on repeated 429 or 5xx conditions and probe cautiously.

Do not retry the payment processor schedule early

Zahlen's fixed recovery schedule is Day 1, Day 2, Day 6, and Day 14. An API 429 is a communications-capacity response; it does not authorize an extra payment retry or a change to the canonical payment-attempt schedule.

5.4 Quotas and quota exhaustion

A quota limits tenant usage over a longer policy period. Depending on the deployment, usage may be measured by request count, event volume, decision volume, outcome volume, or another contracted unit. Exact quota values belong to the tenant's assigned plan and contract.

What quota exhaustion means

When a quota is exhausted, additional eligible requests may be rejected until the quota resets or an administrator changes the allowance. The application should preserve unsubmitted work, avoid duplicate processing, and expose a clear operational alert. Repeatedly sending the same request cannot restore capacity and may increase pressure.

1	2	3	4	5
Detect	**Pause**	**Preserve**	**Assess**	**Resume**
Recognize HTTP 429 and read available response metadata.	Stop immediate retries for the affected operation or tenant.	Keep events, decisions, or outcomes in a durable merchant-side queue.	Check usage, plan assignment, traffic anomalies, and reset policy.	Drain the queue gradually after capacity is available.

Operational questions during quota exhaustion

- Is the tenant near an expected billing-cycle peak, or is this traffic abnormal?
- Is one service, API key, or endpoint responsible for most usage?
- Are duplicate requests or retries consuming the allowance?
- Are outcomes being delayed in a way that breaks the recovery learning loop?
- When does the applicable quota reset?
- Does the merchant need a plan change, a temporary adjustment, or a client-side correction?

Do not silently discard outcomes

Retry outcomes close the learning loop. If outcome submissions are temporarily throttled, store them durably and send them later with their original identifiers and actual outcome timestamps.

5.5 Understanding HTTP 429

HTTP 429 Too Many Requests indicates that the platform is enforcing a rate or quota policy. The response is not a signal to immediately repeat the call. It is a signal to reduce pressure, wait, and retry only when the operation is safe to repeat.

Response element	How to use it
HTTP status 429	Classify the failure as throttling or quota enforcement, not validation or authentication.
Retry-After header, when present	Wait at least the specified interval before retrying.
Request or correlation ID, when present	Include it in logs and support escalation.
Error code or metadata	Distinguish short-window throttling from longer-window quota exhaustion when the deployment provides that detail.
Idempotency state	Reuse the same Idempotency-Key for the same logical POST operation.

Recommended 429 response sequence

1	2	3	4	5
Stop	**Read**	**Back off**	**Reuse**	**Alert**
Do not immediately repeat the failed request.	Inspect Retry-After and structured error metadata.	Use bounded exponential backoff with randomized jitter.	Keep the original idempotency key for the same operation.	Escalate sustained throttling or quota exhaustion.

```
delay = min(max_delay, base_delay * (2 ** retry_number))
delay = delay * random.uniform(0.75, 1.25)

if retry_after_header:
    delay = max(delay, parse_retry_after(retry_after_header))
```

Bound every retry loop

Set a maximum number of attempts, a maximum total elapsed time, and a dead-letter or operator-review path. An unbounded retry loop can turn a temporary 429 into a sustained outage.

5.6 Implementation examples

Python example

```
import random
import time
import requests


def post_with_backoff(url, headers, payload, attempts=5):
    for retry_number in range(attempts):
        response = requests.post(url, headers=headers, json=payload, timeout=20)
        if response.status_code != 429:
            response.raise_for_status()
            return response.json()


        retry_after = response.headers.get('Retry-After')
        if retry_after and retry_after.isdigit():
            delay = float(retry_after)
        else:
            delay = min(60.0, 1.0 * (2 ** retry_number))
            delay *= random.uniform(0.75, 1.25)
        time.sleep(delay)


    raise RuntimeError('Zahlen request remained throttled')
```

JavaScript example

```
async function zahlenFetch(url, options, maxAttempts = 5) {
  for (let attempt = 0; attempt < maxAttempts; attempt += 1) {
    const response = await fetch(url, options);
    if (response.status !== 429) {
      if (!response.ok) throw new Error(`Zahlen HTTP ${response.status}`);
      return response.json();
    }


    const retryAfter = Number(response.headers.get('Retry-After'));
    const baseMs = Number.isFinite(retryAfter)
      ? retryAfter * 1000
      : Math.min(60000, 1000 * (2 ** attempt));
    const jitteredMs = baseMs * (0.75 + Math.random() * 0.5);
    await new Promise(resolve => setTimeout(resolve, jitteredMs));
  }
  throw new Error('Zahlen request remained throttled');
}
```

Curl diagnostic example

```
curl -i -X POST "$ZAHLEN_BASE_URL/v1/_next/retry-decision" \
  -H "Content-Type: application/json" \
  -H "X-API-Key: $ZAHLEN_API_KEY" \
  -H "Idempotency-Key: order-8842-attempt-2" \
  -d '{"attempt_number":2,"decline_code":"51"}'
```

Curl is diagnostic, not a retry engine

Use curl to inspect headers and response bodies. Production retry behavior should be implemented in application code with durable state, bounded attempts, telemetry, and idempotency.

5.7 Safe retries by operation

Operation	Retry after 429?	Required safeguard
GET event, batch, or decision resource	Yes	Use bounded backoff; GET is normally safe to repeat.
POST retry decision	Yes, carefully	Reuse the same Idempotency-Key and identical logical request.
POST retry outcome	Yes, carefully	Preserve decision_id, request_id, attempt number, outcome timestamp, and idempotency.
POST payment-event ingestion	Only with explicit replay safeguards	Use stable event_id values and understand ingestion replay behavior.
Create webhook subscription	Only after verifying prior result	Avoid creating duplicate subscriptions after an ambiguous timeout.
422 validation response	No	Correct the request before resubmitting.
401 or 403	No automatic retry	Correct authentication, authorization, plan, or capability policy first.

Idempotency and payload identity

An idempotency key represents one logical operation. A client must not reuse the same key for a materially different request. If a retry changes the payload, identifiers, or intended operation, the server may correctly treat it as a conflict rather than a replay.

Keep payment retries separate from API retries

Retrying an HTTP request is not the same as retrying a card authorization. HTTP retries preserve communication reliability. Payment retries must follow the Zahlen decision and the fixed Day 1, Day 2, Day 6, and Day 14 schedule.

5.8 Monitoring and alerting

Metric	Why it matters	Suggested alert condition
429 count and rate	Shows active throttling or quota pressure	Unexpected increase or sustained nonzero rate.
Retry-After duration	Shows how long capacity pressure persists	Increasing or unusually long delays.
Quota utilization	Provides advance warning before exhaustion	Configured percentage threshold for the tenant plan.
Queued unsubmitted events	Measures work preserved during throttling	Backlog grows faster than it drains.
Outcome-reporting lag	Detects a broken or delayed learning loop	Outcomes exceed the merchant's acceptable reporting delay.
Duplicate or replay count	Reveals client retry behavior	Unexpected rise in idempotent replays or conflicts.
Traffic by key and endpoint	Helps identify loops or compromised credentials	One key or endpoint deviates materially from baseline.

5.9 Test cases before production

- Validate that a payment-event request with 10,001 events is rejected locally before transmission.
- Validate legacy retry-decision batches are split at 500 events or fewer.
- Simulate HTTP 429 with and without Retry-After.
- Confirm exponential backoff includes jitter and has a maximum delay.
- Confirm the retry loop stops after the configured attempt or elapsed-time limit.
- Confirm POST retries preserve the same idempotency key and payload.
- Confirm throttled outcomes remain in durable storage and retain their actual timestamps.
- Confirm a 401, 403, or 422 is not automatically retried as if it were a 429.
- Confirm traffic resumes gradually after a quota reset or administrative change.
- Confirm no HTTP retry creates an extra card authorization outside Day 1, Day 2, Day 6, and Day 14.

Production readiness rule
A client is not production-ready until it can survive throttling without losing events, duplicating logical operations, creating retry storms, or changing the canonical payment-attempt schedule.

5.10 Chapter summary

- Request limits define the largest valid payload for one endpoint call.
- Payment-event ingestion accepts 1 to 10,000 events per request.
- Legacy batch retry decision accepts no more than 500 events.
- Batch-read pagination accepts limit values from 1 to 1,000 and offset values of 0 or greater.
- Rate limits protect short-window runtime capacity; quotas protect longer-window tenant usage.
- HTTP 429 requires pause, inspection, bounded backoff, jitter, idempotency, and monitoring.
- Quota exhaustion should preserve work in a durable queue rather than discard it.
- Exact numeric plan limits are deployment- and contract-specific.
- HTTP retries never authorize payment attempts outside Zahlen's fixed Day 1, Day 2, Day 6, and Day 14 schedule.

Developer checklist

Check	Ready
Client validates endpoint request-size limits before sending	[ ]
429 handling reads Retry-After when present	[ ]
Backoff uses jitter, maximum delay, and maximum attempts	[ ]
POST retries reuse stable idempotency keys	[ ]
Unsubmitted events and outcomes are stored durably	[ ]
Quota and 429 metrics are monitored by tenant, key, and endpoint	[ ]
401, 403, 422, 429, and 5xx responses have distinct handling	[ ]
API retries cannot create extra payment attempts	[ ]

ZAHLEN

Chapter 6

Payment Events API

Submit events, batch ingestion, event retrieval, and decision retrieval

Audience

Merchants | Developers | Integration Engineers

Commercial workflow
Payment Event -> Retry Decision -> Retry Outcome -> Investigation Run -> Reporting

Zahlen API User Guide v1.0
Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter 6 - Payment Events API

Learning objectives

By the end of this chapter, you should be able to submit one or many payment events, validate ingestion results, retrieve individual events, and consume generated retry decisions safely.

Payment events are the evidence layer of Zahlen. A merchant sends observed authorization or decline information; Zahlen validates and normalizes that evidence, creates durable batch and job identifiers, and makes related event, decision, summary, and processor-result resources available for retrieval.

Canonical retry schedule

Zahlen uses the fixed retry schedule Day 1, Day 2, Day 6, and Day 14. Payment-event evidence should accurately identify the attempt number and timing so the event can be interpreted within this schedule.

6.1 Endpoint overview

Method	Path	Purpose
POST	/v1/payment-events	Submit one or more events using the standard ingestion response.
POST	/v1/payment-events/batch	Submit one or more events using the batch-oriented response.
GET	/v1/payment-events/{event_id}	Retrieve normalized and raw event detail.
GET	/v1/payment-events/{event_id}/decision	Retrieve the decision generated for one event.
GET	/v1/payment-events/{event_id}/processor-result	Retrieve downstream processor execution evidence.
GET	/v1/payment-events/batches/{batch_id}	Retrieve batch state and paginated event IDs.
GET	/v1/payment-events/batches/{batch_id}/summary	Retrieve distributions and retry-candidate counts.
GET	/v1/payment-events/batches/{batch_id}/decisions	Retrieve paginated decisions for a batch.
GET	/v1/payment-events/batches/{batch_id}/processor-results	Retrieve paginated processor results.

All merchant-facing requests use the X-API-Key header. Tenant ownership is derived from the authenticated key; clients do not choose tenant ownership by including tenant_id in the JSON body.

6.2 PaymentEventInput

Each item in the events array is a PaymentEventInput. Only event_id is required by the schema, but richer evidence improves explanation, correlation, reporting, and operational diagnosis. Unknown top-level fields are rejected because additional properties are forbidden.

Required field

Field	Type	Rule	Purpose
event_id	string	Required; minimum length 1	Stable merchant-generated identifier for one payment event.

Recommended evidence groups

Group	Fields	Integration guidance
Decline evidence	decline_code, response_code, paymentech_code, decline_category	Send the processor evidence available at the time of the event.
Issuer context	issuer_bin or bin, issuer, bank, country, card_brand	Use safe issuer-level attributes; never send a full PAN.
Money	amount or amount_minor, currency	Choose units consistently. amount_minor is an integer minor-unit value.
Retry sequence	attempt_number, attempt_day_in_cycle	Align attempts with the fixed Day 1, 2, 6, and 14 schedule.
Timing	decline_timestamp, event_timestamp	Use ISO 8601 strings and preserve the actual observed time.
Merchant correlation	payment_token, customer_id, subscription_id	Use tokenized or merchant-safe references.
Processor context	processor, authorization_id, authorization_latency_ms	Useful for execution analysis and diagnostics.
Transaction context	merchant_category_code, recurring_indicator, transaction_initiator	Clarifies recurring and merchant-initiated payment behavior.
Extensions	metadata	Use only for safe, non-sensitive merchant context.

Sensitive-data rule

Do not place a full card number, CVV, password, bank credential, or other prohibited secret in any field, including metadata. Use tokens and the minimum evidence necessary.

6.3 Submit an event

POST /v1/payment-events accepts PaymentEventsIngestRequest. Although this section demonstrates one event, the request body always contains an events array. The array must contain at least 1 and no more than 10,000 events. source is optional and defaults to payment_events_api.

Minimal request

```
curl -sS -X POST "$ZAHLEN_BASE_URL/v1/payment-events" \
  -H "Content-Type: application/json" \
  -H "X-API-Key: $ZAHLEN_API_KEY" \
  -d '{
    "events": [
      {"event_id": "evt_20260616_0001"}
    ]
  }'
```

Recommended enriched request

```
{
  "source": "subscription_billing",
  "events": [{
    "event_id": "evt_20260616_0001",
    "decline_code": "51",
    "issuer_bin": "411111",
    "card_brand": "VISA",
    "amount_minor": 2999,
    "currency": "USD",
    "attempt_number": 1,
    "attempt_day_in_cycle": 1,
    "event_timestamp": "2026-06-16T12:00:00Z",
    "payment_token": "tok_customer_001",
    "customer_id": "cus_001",
    "subscription_id": "sub_001",
    "processor": "example_processor",
    "metadata": {"invoice_id": "inv_1042"}
  }]
}
```

Request validation

- events is required and must contain 1 through 10,000 items.
- Each event_id must be a non-empty string.
- amount and amount_minor cannot be negative.
- attempt_number, when supplied, must be at least 1.
- attempt_day_in_cycle and authorization_latency_ms cannot be negative.
- Unknown top-level request or event fields produce a validation failure rather than being silently ignored.

6.4 Understand the ingestion response

The standard ingestion response confirms receipt and provides identifiers required for later retrieval and support correlation. Persist these identifiers before continuing the workflow.

Field	Meaning	Client action
status	Current ingestion status	Treat as the authoritative state, not as a settlement result.
merchant_id	Authenticated merchant context	Use for diagnostics; do not use it to infer tenant ownership.
tenant_id	Resolved tenant context; nullable in schema	Never submit or override this value from client input.
payment_event_batch_id	Batch created for the request	Store with the local ingest operation.
upload_job_id	Background processing / investigation correlation	Log and retain for status and support investigations.
received_event_count	Events received by the route	Compare with the local intended count.
total_rows / valid_rows / invalid_rows	Validation and row totals	Alert on unexpected invalid or missing rows.
error_count	Recorded ingestion errors	Inspect ingestion detail when nonzero.
created_at / completed_at	Lifecycle timestamps	completed_at may be null while work continues.
ingestion	Additional ingest detail object	Treat as extensible and avoid brittle assumptions.

Acceptance is not downstream completion

A successful POST proves that the API accepted the request according to its response. It does not prove that every downstream decision, investigation, or reporting bridge has completed. Preserve upload_job_id and check the appropriate retrieval resources.

Identifier chain

Identifier	Created by	Use
event_id	Merchant	Stable correlation for one event.
payment_event_batch_id / batch_id	Zahlen	Groups an ingestion request.
upload_job_id	Zahlen	Connects ingestion to background processing and investigation.
decision_id	Zahlen	Identifies one generated retry decision.
request_id	Zahlen	Trace and support correlation.

6.5 Batch submission

POST /v1/payment-events/batch uses the same PaymentEventsIngestRequest model but returns PaymentEventsBatchIngestResponse. Use it when the integration naturally groups multiple observations and needs an explicit batch resource.

Batch request example

```
curl -sS -X POST "$ZAHLEN_BASE_URL/v1/payment-events/batch" \
  -H "Content-Type: application/json" \
  -H "X-API-Key: $ZAHLEN_API_KEY" \
  --data-binary @payment-events.json
```

```
{
  "source": "nightly_billing_export",
  "events": [
    {
      "event_id": "evt_batch_0001",
      "decline_code": "51",
      "attempt_number": 1,
      "attempt_day_in_cycle": 1
    },
    {
      "event_id": "evt_batch_0002",
      "decline_code": "91",
      "attempt_number": 2,
      "attempt_day_in_cycle": 2
    }
  ]
}
```

Batch response fields

Field	Meaning
submitted	Number presented in the request.
accepted	Number accepted for processing.
rejected	Number rejected.
batch_id	Durable batch identifier.
events_url	URL for retrieving the batch events resource.
upload_job_id	Background-processing correlation.
created_at / completed_at	Lifecycle timestamps.
ingestion	Optional extensible ingestion details.

Batch design checklist

- Validate the entire payload locally before transmission.
- Use unique event_id values unless replay behavior has been explicitly designed.
- Persist a local batch record before sending.
- Store batch_id and upload_job_id immediately after success.
- Compare submitted, accepted, and rejected counts.
- Do not split or repeat batches in a way that creates extra payment attempts outside Day 1, Day 2, Day 6, and Day 14.

6.6 Retrieve a batch

Use the batch resource to inspect state and obtain event IDs. The detail, decision, and processor-result list routes use offset-based pagination. offset must be at least 0; limit, when supplied, must be from 1 through 1,000.

```
curl -sS \
  -H "X-API-Key: $ZAHLEN_API_KEY" \
  "$ZAHLEN_BASE_URL/v1/payment-events/batches/$BATCH_ID?offset=0&limit=250"
```

Resource	Important fields
Batch detail	status, submitted, accepted, rejected, event_ids, total_events, returned, offset, limit, has_more
Batch summary	event_count, retry_candidates, top processors, decline codes, issuers, issuer BINs, card brands, confidence and decision distributions
Batch decisions	decisions, total_events, returned, offset, limit, has_more, decisions_source
Batch processor results	processor_results, total_events, returned, offset, limit, has_more, processor_results_source

Pagination loop

```
offset = 0
limit = 250
while True:
    page = get_batch_decisions(batch_id, offset=offset, limit=limit)
    process(page.get("decisions", []))
    if not page.get("has_more", False):
        break
    offset += page["returned"]
```

Pagination safety

Advance by the returned count and stop when has_more is false. Protect against a zero returned count with has_more=true so a malformed or transient response cannot create an infinite loop.

6.7 Retrieve an event

GET /v1/payment-events/{event_id} returns PaymentEventDetailResponse. The result combines durable correlation fields, commonly used normalized attributes, the normalized_event object, and the raw_event object.

```
curl -sS \
  -H "X-API-Key: $ZAHLEN_API_KEY" \
  "$ZAHLEN_BASE_URL/v1/payment-events/evt_20260616_0001"
```

How to use the response

Response area	Use
Correlation	event_id, payment_event_batch_id, upload_job_id, source, source_row_number
Merchant context	merchant_id and resolved tenant_id when returned
Normalized common fields	decline_code, response_code, issuer_bin, issuer, country, card_brand, amount_minor, currency, attempt_number, event_timestamp, processor, authorization_id
normalized_event	Preferred machine-readable normalized representation for application logic.
raw_event	Original or source representation for diagnostics and evidence review.
created_at / updated_at	Persistence lifecycle timestamps.

Normalized versus raw

Use normalized fields for routine client logic. Preserve raw_event for diagnostics, but do not build fragile production logic around processor-specific raw keys unless a separate contract guarantees them.

Not-found behavior

A 404 can mean that the event does not exist or is not visible to the authenticated tenant. Verify the environment, API key, and exact event_id. Never bypass tenant filters to make a resource appear.

6.8 Retrieve a decision

GET /v1/payment-events/{event_id}/decision returns PaymentEventDecisionResponse. This is the recommendation generated from the event evidence. A recommendation is not the observed processor outcome and must not be treated as recovered revenue.

```
curl -sS \
  -H "X-API-Key: $ZAHLEN_API_KEY" \
  "$ZAHLEN_BASE_URL/v1/payment-events/evt_20260616_0001/decision"
```

Field	Interpretation
decision	Explicit control signal. Follow this value rather than inferring action from other fields.
recommended_retry_day	Nullable recommended day. When present, it should align with the fixed Day 1, 2, 6, or 14 schedule.
confidence / confidence_score	Evidence strength, not a guarantee of payment success.
reason_codes	Machine-readable explanation categories.
reason_detail	Human-readable explanation; nullable.
policy_source / matched_policy_id	Policy provenance when available.
decision_id / request_id	Durable identifiers for logging, outcome reporting, and support.
idempotent_replay	Indicates that an earlier logical operation was safely replayed.
event	Event evidence associated with the decision.
issuer_context	Optional issuer-level context used or returned by the decision flow.

Never invent a retry day

If recommended_retry_day is null, do not select an arbitrary day. Follow the decision and reason fields. Where a retry is authorized by Zahlen, execution must remain within the fixed Day 1, Day 2, Day 6, and Day 14 schedule.

Decision and processor result are separate

The decision tells the merchant what Zahlen recommends. GET /v1/payment-events/{event_id}/processor-result reports downstream execution evidence, including result, processor reference, response code, description, and processing time. Chapter 8 explains how to report retry outcomes and close the learning loop.

6.9 End-to-end client pattern

1. Create a stable event_id and persist the local event record.
2. Validate the payload and submit it with X-API-Key.
3. Persist payment_event_batch_id or batch_id and upload_job_id from the response.
4. Retrieve the event and verify normalized evidence when needed.
5. Retrieve the generated decision and preserve decision_id and request_id.
6. Execute only the authorized payment action and only on the fixed Day 1, Day 2, Day 6, or Day 14 schedule.
7. Report the observed retry outcome through the Retry Outcome API.
8. Monitor investigation and reporting resources for broader patterns.

6.10 Error handling summary

Status	Meaning in this workflow	Response
400	Malformed or business-invalid request	Correct the payload; do not blindly retry.
401	Missing, invalid, revoked, or wrong-environment API key	Verify X-API-Key and environment.
403	Authenticated but not permitted	Check plan, capability, or endpoint authorization.
404	Event or batch absent or not tenant-visible	Verify identifier and authenticated tenant context.
409	Conflict or replay mismatch	Compare the logical operation and original request.
422	Schema validation failure	Correct field spelling, types, limits, and unknown properties.
429	Rate or quota enforcement	Back off and honor Retry-After when present.
500/503	Server or dependency failure	Retry safely with backoff; preserve stable identifiers and idempotency behavior.

6.11 Production readiness checklist

- API keys are loaded from a secret manager, not source code.
- event_id uniqueness and replay behavior are documented.
- Client models reject unknown fields before sending.
- amount and amount_minor units cannot be confused.
- All timestamps are produced and parsed consistently.
- Batch counts and invalid-row counts are monitored.
- Batch pagination is bounded and loop-safe.
- Identifiers are included in logs without exposing secrets.
- Decision handling supports nullable fields.
- Payment execution is constrained to Day 1, Day 2, Day 6, and Day 14.
- Observed outcomes are reported separately from recommendations.

Chapter takeaway

A reliable Payment Events integration preserves evidence and identifiers. Submit accurate observations, treat ingestion as an asynchronous durable workflow, retrieve normalized events and explicit decisions, and keep recommendation, execution, and outcome as separate facts.

ZAHLEN

Chapter 7

Retry Decision API

Single decisions, batch decisions, response interpretation, and confidence

Audience

Merchants | Developers | Integration Engineers

Commercial workflow
Payment Event -> Retry Decision -> Retry Outcome -> Investigation Run -> Reporting

Zahlen API User Guide v1.0
Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter 7 - Retry Decision API

Learning objectives

By the end of this chapter, you should be able to choose the correct retry-decision contract, submit single and batch requests, interpret returned actions and explanations, and use confidence without treating it as a guarantee.

The Retry Decision API evaluates payment evidence and returns an operational recommendation. It tells a merchant whether to retry now, wait for a scheduled retry day, stop retrying, or escalate for review. The API does not execute the payment and does not prove that a future retry will succeed.

Canonical retry schedule

Zahlen is built around the fixed retry schedule Day 1, Day 2, Day 6, and Day 14. A returned retry day must be interpreted within this schedule. Client code must not invent extra retry days or move an authorization attempt outside the fixed sequence.

7.1 Decision endpoints

Method	Path	Contract	Typical use
POST	/v1/retry-decision	Legacy single decision	Existing integrations requiring the full historical request and response structure.
POST	/v1/retry-decision/batch	Legacy batch decision	Evaluate up to 500 legacy-format events in one request.
POST	/v1/_next/retry-decision	Next-generation single decision	Streamlined decision contract for integrations selected during onboarding.

Do not mix contracts

The legacy and next-generation endpoints use different request and response models. Select one approved contract and map it explicitly. Do not send fields from one model to the other or build client logic that silently combines both.

7.2 Authentication and idempotency

Merchant-facing decision requests use the X-API-Key header. Tenant and merchant ownership are resolved from the authenticated key. Decision routes support Idempotency-Key where documented by the implementation; use one stable key for one logical decision operation.

```
X-API-Key: zk_live_REPLACE_ME
Idempotency-Key: order-8842-attempt-2
Content-Type: application/json
```

Why idempotency matters

- A network timeout can occur after Zahlen has created the decision but before the client receives the response.
- Retrying with the same idempotency key allows the server to recognize the same logical operation.
- A retry for the same operation must preserve the same request body and key.
- A genuinely new attempt should use a new idempotency key.
- Persist the returned request_id, decision_id, and idempotent replay indicator.

API retry versus payment retry

Retrying an HTTP request is not the same as retrying the customer payment. HTTP retries must be safe and idempotent. Payment retries must occur only on the fixed Day 1, Day 2, Day 6, and Day 14 schedule.

Recommended idempotency pattern

```
decision:{merchant_order_id}:{billing_cycle_id}:attempt:{attempt_number}

Example:
decision:order-8842:cycle-2026-06:attempt:2
```

7.3 Legacy single decision

POST /v1/retry-decision uses RetryDecisionRequest. Extra fields are forbidden. Five fields are required; the remaining fields provide issuer, processor, transaction, and policy context.

Required request fields

Field	Type	Purpose
payment_token	string	Merchant-safe token representing the payment method.
billing_cycle_id	string	Stable identifier for the subscription or billing cycle.
event_ts_iso	string	Timestamp of the observed authorization or decline event.
cycle_start_ts_iso	string	Start timestamp for the billing cycle.
paymentech_code	string	Processor response or decline code expected by the legacy contract.

Legacy request example

```
curl -sS -X POST "$ZAHLEN_BASE_URL/v1/retry-decision" \
  -H "Content-Type: application/json" \
  -H "X-API-Key: $ZAHLEN_API_KEY" \
  -H "Idempotency-Key: cycle-2026-06-attempt-2" \
  -d '{
    "payment_token": "tok_001",
    "billing_cycle_id": "cycle_2026_06",
    "event_ts_iso": "2026-06-16T12:00:00Z",
    "cycle_start_ts_iso": "2026-06-01T00:00:00Z",
    "paymentech_code": "51",
    "card_network": "VISA",
    "amount": 29.99,
    "currency": "USD",
    "attempt_day_in_cycle": 2,
    "subscription_id": "sub_001"
  }'
```

Optional legacy fields include country and issuer context, BIN values, processor, recommended action evidence, outcome flags, order/invoice/subscription references, transaction kind, spike-alert controls, and deterministic or AI external-change mode. Send only fields your integration can populate accurately.

7.4 Interpret the legacy response

RetryDecisionResponse is organized into decision, reasoning, signals, explanations, metadata, and an optional policy block. This structure separates the control instruction from the evidence and version information used to produce it.

Decision block

Field	Allowed values / type	Interpretation
action	RETRY_NOW, WAIT, STOP, ESCALATE	Primary operational instruction.
state	RETRY_SCHEDULED, RETRY_NOW_ELIGIBLE, DO_NOT_RETRY, REQUIRES_REVIEW	More specific decision state.
recommended_next_attempt_day_in_cycle	integer or null	Next permitted retry day when one is scheduled.
recommended_next_attempt_at	string or null	Timestamp representation when provided.

Reasoning and signals

Section	Important fields	Use
reasoning	reason_code, reason, confidence, guidance_source	Explain why the action was selected.
signals	seen_before, truth_confidence_band, external_status, analysis_mode	Describe supporting evidence and prior observation.
explanations	array	Present additional human-readable explanation when available.
meta	request_id, versions, processed_at, idempotency_key	Support traceability and reproducibility.
policy	applied, merchant_id, rule_id, source	Show whether a merchant policy affected the result.

Action is the control signal

Base automated workflow on the explicit action and state. Reason text is designed for explanation and support, not for fragile string matching.

7.5 Legacy batch decision

POST /v1/retry-decision/batch accepts BatchRetryDecisionRequest. The events array can contain up to 500 legacy decision requests. This is a schema ceiling, not a guaranteed throughput target.

```
{
  "events": [
    {
      "payment_token": "tok_001",
      "billing_cycle_id": "cycle_2026_06_a",
      "event_ts_iso": "2026-06-16T12:00:00Z",
      "cycle_start_ts_iso": "2026-06-01T00:00:00Z",
      "paymentech_code": "51",
      "attempt_day_in_cycle": 1
    },
    {
      "payment_token": "tok_002",
      "billing_cycle_id": "cycle_2026_06_b",
      "event_ts_iso": "2026-06-16T12:01:00Z",
      "cycle_start_ts_iso": "2026-06-01T00:00:00Z",
      "paymentech_code": "91",
      "attempt_day_in_cycle": 2
    }
  ]
}
```

Batch response

Field	Meaning	Client guidance
results	One evaluated item per returned decision	Correlate each result with the original event identity; do not rely only on array position unless contractually guaranteed.
meta	Batch request/version metadata	Log request ID, processing time, and version fields.
count	Number of returned results	Compare with the intended and submitted event counts.

- Split larger workloads into bounded batches below the 500-item maximum.
- Do not retry the whole batch blindly after a partial or uncertain result.
- Use stable event and billing-cycle identifiers so individual items can be reconciled.
- Monitor batch latency and 429 activity; batching does not bypass rate limits or quotas.

7.6 Next-generation single decision

POST /v1/_next/retry-decision uses NextRetryDecisionPayload. Only attempt_number is required, and it must be at least 1. The streamlined contract can carry decline, issuer, amount, timing, authorization, and recurring-payment context. Extra fields are forbidden.

Core request fields

Field	Type	Required	Notes
attempt_number	integer	Yes	Minimum 1; identify the attempt within the merchant workflow.
token	string or null	No	Merchant-safe payment token.
decline_code	string or null	No	Observed decline or processor response code.
issuer_bin	string or null	No	Issuer identification context; never send a full PAN.
issuer_name / issuer_country / card_brand	string or null	No	Issuer and card context.
amount_minor	integer or null	No	Nonnegative integer in minor currency units.
currency	string or null	No	Currency code paired with amount_minor.
decline_category	string or null	No	Merchant or processor classification when available.
event_timestamp	string or null	No	Observed event time.
authorization_id / authorization_latency_ms	string / number or null	No	Processor correlation and latency.
merchant_category_code / recurring_indicator / transaction_initiator	string or null	No	Transaction context.

Request example

```
curl -sS -X POST "$ZAHLEN_BASE_URL/v1/_next/retry-decision" \
  -H "Content-Type: application/json" \
  -H "X-API-Key: $ZAHLEN_API_KEY" \
  -H "Idempotency-Key: order-8842-attempt-2" \
  -d '{
    "token": "tok_001",
    "attempt_number": 2,
    "decline_code": "51",
    "issuer_bin": "411111",
    "card_brand": "VISA",
    "amount_minor": 2999,
    "currency": "USD",
    "event_timestamp": "2026-06-16T12:00:00Z"
  }'
```

7.7 Interpret the next-generation response

NextRetryDecisionResponse provides a flat decision envelope with traceability, explanation, policy, and issuer context. Nullable fields must be represented explicitly in typed clients.

Field	Type	Interpretation
request_id	string	Request correlation for logs and support.
decision_id	string	Durable identifier for this decision.
merchant_id	string	Resolved merchant context.
token	string or null	Returned merchant token correlation.
attempt_number	integer	Attempt evaluated.
decision	string	Primary decision value; use as the control signal.
retry_day	integer or null	Scheduled retry day, when applicable.
reason_code	string	Machine-readable explanation category.
reason_detail	string or null	Human-readable explanation.
policy_source / matched_policy_id	string or null	Policy provenance.
confidence	number or null	Evidence-strength score, not a guarantee.
created_at	string	Decision creation timestamp.
idempotent_replay	boolean	Whether the result was served as an idempotent replay.
issuer_context	object or null	Extensible issuer evidence.
explainability_sections	array	Additional structured explanations.
decision_trace	object	Machine-readable trace details.

Nullable retry day

A null retry_day can be correct. It may accompany a stop, escalation, or other decision that intentionally does not schedule another payment attempt. Do not substitute the next calendar day.

Schedule validation

When a retry is authorized, client validation should confirm that the returned retry day is one of the canonical schedule points: 1, 2, 6, or 14. An unexpected value should be logged and escalated rather than silently executed.

```
ALLOWED_RETRY_DAYS = {1, 2, 6, 14}

if response["retry_day"] is not None:
    if response["retry_day"] not in ALLOWED_RETRY_DAYS:
        raise ValueError("Unexpected retry day; do not execute payment")
```

7.8 Confidence scores

Confidence describes the strength and consistency of the evidence behind a decision. It does not measure the dollar importance of the account and does not promise authorization success. The legacy response uses LOW, MED, or HIGH; the next-generation response exposes a nullable numeric confidence value.

Confidence concept	Correct interpretation	Incorrect interpretation
LOW / lower numeric confidence	Limited, sparse, new, or conflicting evidence; preserve decision but increase review and monitoring.	The decision is automatically wrong.
MED / middle-range confidence	Useful evidence with some uncertainty; follow the decision and retain explanation.	There is a fixed universal success percentage.
HIGH / higher numeric confidence	Strong, consistent evidence for the selected action.	The payment is guaranteed to recover.

Operational use

- Always honor the explicit decision before interpreting confidence.
- Expose reason and confidence to operators who investigate exceptions.
- Use confidence to select review, monitoring, or escalation intensity.
- Do not create undocumented confidence thresholds unless they are part of an approved merchant policy.
- Record confidence with the decision ID, request ID, rules or policy source, and timestamp.

High confidence does not replace outcome reporting

A high-confidence retry recommendation still requires a real processor attempt and a reported retry outcome. Decision quality is learned and measured only when the merchant closes the loop.

7.9 Decision-handling workflow

1. Build a schema-valid request using the approved legacy or next-generation contract.
2. Send X-API-Key and a stable Idempotency-Key for the logical operation.
3. Persist the complete response before scheduling or executing a payment action.
4. Read the explicit decision or action and state.
5. Validate any retry day against Day 1, Day 2, Day 6, and Day 14.
6. Use reason, confidence, policy, and issuer context for explanation and review.
7. Execute the permitted payment attempt in the merchant payment stack.
8. Report the observed result through POST /v1/retry-outcome.

Example control logic

```
decision = response["decision"]
retry_day = response.get("retry_day")

if decision in {"STOP", "DO_NOT_RETRY"}:
    cancel_future_payment_attempts()
elif decision in {"ESCALATE", "REQUIRES_REVIEW"}:
    open_manual_review(response)
elif retry_day in {1, 2, 6, 14}:
    schedule_payment_attempt(day=retry_day)
else:
    hold_and_alert("Decision cannot be safely executed")
```

Recommendation is not settlement

The Retry Decision API recommends operational behavior. The processor result and settlement status remain separate facts and must be reported through the outcome workflow.

7.10 Error handling and safe retries

Status	Meaning	Client response
400	Malformed or business-invalid request	Correct the request; do not blindly retry.
401	Missing or invalid API key	Check secret injection, key status, and environment.
403	Authenticated but not authorized	Check plan, endpoint access, or contract.
409	Conflict or idempotency mismatch	Compare the original body and idempotency key.
422	Schema validation failed	Fix required, typed, constrained, or unknown fields.
429	Rate or quota enforcement	Honor Retry-After when present and use backoff with jitter.
500/503	Server or dependency failure	Retry with the same idempotency key and bounded backoff.

Production checklist

- The client implements exactly one approved decision contract per integration path.
- Unknown request fields are rejected locally before transmission.
- API keys and idempotency keys are centralized and never logged as secrets.
- Every response is logged using request_id and decision_id.
- Nullable retry day, confidence, policy, and issuer context are handled safely.
- Only Day 1, Day 2, Day 6, and Day 14 can create payment attempts.
- Batch reconciliation does not rely on fragile assumptions.
- 429 and transient 5xx retries are bounded, jittered, and idempotent.
- Every executed recommendation is followed by outcome reporting.

Chapter summary

The Retry Decision API converts payment evidence into an explainable action. Reliable integrations keep legacy and next-generation contracts separate, use idempotency for safe HTTP retries, treat confidence as evidence strength, obey the fixed Day 1/2/6/14 schedule, and report the actual outcome.

ZAHLEN

Chapter 8
Retry Outcome API

Outcome reporting • Recovery learning loop

For merchants, developers, and integration engineers

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Core outcome principle

A retry decision is a recommendation. A retry outcome is the observed result. Zahlen becomes more useful only when merchants report what actually happened after the scheduled attempt.

Canonical Zahlen retry schedule

Day 1 → Day 2 → Day 6 → Day 14

Chapter 8 - Retry Outcome API

Learning objectives

By the end of this chapter, you should be able to submit accurate retry outcomes, correlate outcomes to decisions, preserve idempotency, and explain how outcome data closes Zahlen's recovery learning loop.

8.1 Why outcome reporting matters

The Retry Outcome API records the result of an authorization attempt after Zahlen has produced a retry decision or after the merchant has executed a scheduled retry. The endpoint is:

```
POST /v1/retry-outcome
```

Without outcome reporting, the platform can recommend an action but cannot verify whether the action recovered revenue, produced another decline, or resulted in a neutral operational result. Outcome data changes Zahlen from a one-way recommendation service into a measurable recovery system.

Stage	Question answered	Durable evidence
Payment Event	What happened at the original authorization?	Event ID, decline evidence, issuer and processor context
Retry Decision	Should another attempt occur, and on which fixed schedule day?	Request ID, decision ID, retry day, reason and confidence
Retry Outcome	What actually happened when the attempt was executed?	Outcome ID, outcome, approval or final decline evidence
Recovery Intelligence	Did the schedule and policy recover revenue over time?	Observed recovery truth, trends, issuer behavior and reporting

Do not infer success from scheduling

A scheduled retry is not a successful payment. Report success only from observed authorization or settlement evidence.

8.2 The outcome-reporting workflow

1. Preserve the request_id and decision_id returned by the Retry Decision API.
2. Execute the payment attempt in the merchant payment stack on the applicable fixed retry day: Day 1, Day 2, Day 6, or Day 14.
3. Capture the processor result, timestamp, approval code or final decline code, and settled amount when available.
4. Map the processor result into the documented RetryOutcomePayload.

5. Submit the outcome using the merchant X-API-Key and safe replay controls.
6. Persist the returned outcome_id and matched_by value with the merchant authorization audit trail.
7. Monitor missing or delayed outcomes so the recovery learning loop remains complete.

Two different kinds of retry

An HTTP retry repeats an API request after a network or server failure. A payment retry performs another authorization attempt. HTTP retries must never create additional payment attempts or alter Zahlen's fixed Day 1, Day 2, Day 6, and Day 14 schedule.

8.3 Request contract: RetryOutcomePayload

The request model forbids unknown top-level properties. Only attempt_number and outcome are required by the schema, but correlation and processor evidence fields are operationally important. Tenant ownership is resolved from the authenticated API key, not trusted from the request body.

Field	Type	Required	Meaning and guidance
merchant_id	string or null	No	Optional business identifier; authenticated tenant context remains authoritative.
attempt_number	integer	Yes	Minimum 0. Use the attempt number that actually executed.
outcome	string	Yes	Observed result. Use the deployment-approved outcome vocabulary.
request_id	string or null	No	Correlates the outcome to the originating decision request.
decision_id	string or null	No	Correlates the outcome to the specific Zahlen decision.
token	string or null	No	Merchant payment token or safe payment reference; never a raw PAN.
approval_code	string or null	No	Processor approval evidence when the attempt succeeds.
final_decline_code	string or null	No	Final processor decline code when the attempt fails.
settled_amount_minor	integer or null	No	Non-negative amount in minor units, such as cents.
currency	string or null	No	Currency associated with settled_amount_minor.

Field	Type	Required	Meaning and guidance
outcome_timestamp	string or null	No	Actual event time, preferably ISO 8601 with timezone.

Outcome vocabulary is deployment-specific

The schema requires a string but does not define a universal enumeration. Use only outcome values documented by the active Zahlen deployment or onboarding contract. Do not invent client-only values without agreement.

8.4 Submit a successful outcome

The following example reports that attempt 2, executed on Day 2 of the fixed schedule, was approved and settled for $29.99 USD.

```
curl -sS -X POST https://api.example.com/v1/retry-outcome \
 -H 'Content-Type: application/json' \
 -H 'X-API-Key: zk_live_REPLACE_ME' \
 -H 'Idempotency-Key: outcome-dec_01-attempt-2' \
 -d '{
  "attempt_number": 2,
  "outcome": "RECOVERED",
  "request_id": "req_01",
  "decision_id": "dec_01",
  "token": "tok_customer_001",
  "approval_code": "APPR42",
  "settled_amount_minor": 2999,
  "currency": "USD",
  "outcome_timestamp": "2026-06-17T15:42:18Z"
 }'
```

Example values are illustrative

Confirm the accepted outcome vocabulary and whether Idempotency-Key is enabled for the active deployment before production use.

8.5 Submit a declined outcome

When a scheduled attempt is declined, report the observed decline rather than omitting the outcome. Failed attempts are essential evidence for evaluating later retry days and issuer patterns.

```
{
 "attempt_number": 3,
 "outcome": "DECLINED",
 "request_id": "req_02",
 "decision_id": "dec_02",
 "token": "tok_customer_002",
```

```
 "final_decline_code": "51",
 "currency": "USD",
 "outcome_timestamp": "2026-06-21T10:03:00Z"
}
```


In this example, attempt 3 corresponds to the Day 6 position in Zahlen's fixed schedule. A later attempt must still follow the next permitted schedule position rather than being triggered by an API transport retry.

8.6 Response contract: RetryOutcomeResponse

A successful response confirms the durable outcome record and returns the matching information Zahlen used to associate it with prior evidence.

Field	Type	How to use it
outcome_id	string	Durable identifier for the recorded outcome.
merchant_id	string	Merchant resolved by the authenticated context.
request_id	string or null	Returned correlation to the decision request.
decision_id	string or null	Returned correlation to the decision.
token	string or null	Returned safe payment reference.
attempt_number	integer	Recorded attempt number.
outcome	string	Recorded outcome value.
approval_code	string or null	Recorded approval evidence.
final_decline_code	string or null	Recorded decline evidence.
settled_amount_minor	integer or null	Recorded settled amount in minor units.
currency	string or null	Recorded currency.
outcome_timestamp	string or null	Time the merchant says the outcome occurred.
created_at	string	Time Zahlen created the durable record.
matched_by	string	How Zahlen correlated the outcome to existing evidence.


```
{
 "outcome_id": "out_01",
 "merchant_id": "merchant_demo",
 "request_id": "req_01",
 "decision_id": "dec_01",
 "token": "tok_customer_001",
```

```
 "attempt_number": 2,
 "outcome": "RECOVERED",
 "approval_code": "APPR42",
 "final_decline_code": null,
 "settled_amount_minor": 2999,
 "currency": "USD",
 "outcome_timestamp": "2026-06-17T15:42:18Z",
 "created_at": "2026-06-17T15:42:20Z",
 "matched_by": "decision_id"
}
```


8.7 Correlation and matching

Strong correlation allows Zahlen to connect a real processor result to the exact recommendation and payment context that produced it. Supply the strongest available identifiers rather than relying on a token alone.

Correlation field	Strength	Guidance
decision_id	Strongest	Preferred direct link to one Zahlen decision.
request_id	Strong	Links the outcome to the decision request and support trace.
token + attempt_number	Fallback	Useful when decision identifiers are unavailable; requires careful token hygiene.
merchant_id	Context only	Business context; not a substitute for tenant authentication or decision correlation.

Store matched_by

The response field matched_by explains how the server linked the outcome. Persist it for diagnostics and alert when production traffic unexpectedly falls back to a weaker matching method.

8.8 The recovery learning loop

Outcome records are the feedback signal that lets Zahlen measure the effectiveness of the fixed retry schedule and distinguish recommendations from observed recovery. The loop operates at several levels:

Level	Outcome evidence enables
Individual payment	Confirm whether the specific retry recovered or remained declined.
Retry-day performance	Compare observed results at Day 1, Day 2, Day 6, and Day

Level	Outcome evidence enables
	14.
Decline category	Measure which categories respond to later attempts and which remain unrecoverable.
Issuer and BIN	Identify issuer-level recovery shifts, degradation, and timing patterns.
Merchant program	Measure recovery rate, reporting completeness, and revenue recovered.
Operational intelligence	Populate Recovery Truth, trends, monitoring events, issuer health, and investigation reporting.

A complete loop does not mean the decision engine changes the fixed retry days. It means the platform can measure and explain performance within the canonical Day 1, Day 2, Day 6, and Day 14 framework and surface operational patterns that require attention.

8.9 Data-quality rules

- Report the actual authorization or settlement result, not a prediction or scheduled state.
- Use the actual outcome timestamp, not the time a reconciliation report was generated.
- Keep attempt_number aligned with the executed attempt and fixed retry position.
- Use settled_amount_minor only with the corresponding currency and non-negative integer units.
- Send approval_code for observed approvals and final_decline_code for observed declines when available.
- Use payment tokens or safe references; never submit a full PAN, CVV, password, or raw banking credential.
- Preserve request_id, decision_id, outcome_id, and matched_by in logs and audit records.
- Do not silently transform unknown processor results into RECOVERED or DECLINED; use an approved mapping table.

Reporting completeness is a production metric

Track the percentage of executed attempts that receive a durable outcome. Missing outcomes create biased recovery metrics and weaken issuer and retry-day analysis.

8.10 Safe API retries and duplicate prevention

Outcome submission can fail after the server has accepted a request but before the client receives the response. Design the client so repeating the same logical report does not create conflicting evidence.

8. Create one stable idempotency key for one logical outcome report when the deployment supports the header.
9. Reuse the same request body and idempotency key after a timeout or transient 5xx response.
10. Never change outcome, amount, or identifiers while reusing an idempotency key.
11. Apply bounded exponential backoff with jitter for 429 and transient server failures.

12. Do not retry validation failures until the payload is corrected.
13. If the final submission status is unknown, reconcile by identifiers or contact the approved support path before sending a contradictory outcome.

```
# Conceptual retry delay
base_delay = 1.0
max_delay = 30.0
delay = min(max_delay, base_delay * (2 ** retry_number))
delay *= random.uniform(0.75, 1.25)
```

8.11 Error handling

HTTP status	Meaning	Client action
200 / 201	Outcome accepted	Persist outcome_id, matched_by, and correlation fields.
400	Malformed or business-invalid outcome	Correct mapping or required business data; do not blindly retry.
401	Missing or invalid API key	Check secret, environment, and X-API-Key header.
403	Authenticated but not permitted	Check plan, capability, or endpoint authorization.
409	Conflict or idempotency mismatch	Compare the original payload and idempotency key.
422	Schema validation failure	Fix field names and types; extra fields are forbidden.
429	Rate or quota enforcement	Honor Retry-After when present and back off with jitter.
500 / 503	Transient server or dependency failure	Retry safely using stable identifiers and idempotency; alert if sustained.

8.12 Python example

```
import os
import requests

base_url = os.environ["ZAHLEN_BASE_URL"]
api_key = os.environ["ZAHLEN_API_KEY"]

payload = {
  "attempt_number": 2,
  "outcome": "RECOVERED",
  "request_id": "req_01",
  "decision_id": "dec_01",
  "token": "tok_customer_001",
  "approval_code": "APPR42",
  "settled_amount_minor": 2999,
  "currency": "USD",
```

```
  "outcome_timestamp": "2026-06-17T15:42:18Z",
}

response = requests.post(
  f"{base_url}/v1/retry-outcome",
  headers={
    "Content-Type": "application/json",
    "X-API-Key": api_key,
    "Idempotency-Key": "outcome-dec_01-attempt-2",
  },
  json=payload,
  timeout=20,
)
response.raise_for_status()
result = response.json()
print(result["outcome_id"], result["matched_by"])
```

8.13 JavaScript example

```
const payload = {
 attempt_number: 2,
 outcome: "RECOVERED",
 request_id: "req_01",
 decision_id: "dec_01",
 token: "tok_customer_001",
 approval_code: "APPR42",
 settled_amount_minor: 2999,
 currency: "USD",
 outcome_timestamp: "2026-06-17T15:42:18Z"
};

const response = await fetch(
 `${process.env.ZAHLEN_BASE_URL}/v1/retry-outcome`,
 {
  method: "POST",
  headers: {
   "Content-Type": "application/json",
   "X-API-Key": process.env.ZAHLEN_API_KEY,
   "Idempotency-Key": "outcome-dec_01-attempt-2"
  },
  body: JSON.stringify(payload)
 }
);

if (!response.ok) {
 throw new Error(`Zahlen HTTP ${response.status}`);
}
const result = await response.json();
console.log(result.outcome_id, result.matched_by);
```

8.14 Production readiness checklist

- The processor-to-Zahlen outcome mapping is documented and contract-tested.
- Every executed fixed-schedule attempt can be correlated to request_id and decision_id when available.

- Outcome timestamps preserve timezone and represent the actual authorization or settlement event.
- Approval and decline evidence is captured without prohibited cardholder data.
- HTTP retries reuse stable identifiers and idempotency semantics.
- Outcome reporting lag and reporting-completeness percentages are monitored.
- Conflicting duplicate outcomes create alerts rather than silent overwrites.
- Logs include outcome_id, decision_id, request_id, attempt_number, and matched_by.
- Day 1, Day 2, Day 6, and Day 14 are the only payment retry positions used by the integration.
- Production smoke tests verify successful, declined, validation-error, 429, timeout, and replay scenarios.

8.15 Chapter summary

The Retry Outcome API closes Zahlen's commercial recovery loop by recording the result that actually occurred after a retry decision or scheduled attempt. Clients submit POST /v1/retry-outcome with attempt_number and outcome, enrich the report with decision, token, processor, amount, and timestamp evidence, and persist the returned outcome_id and matched_by fields. Reliable outcome reporting provides the observed Recovery Truth needed to measure performance across the fixed Day 1, Day 2, Day 6, and Day 14 schedule.

Next chapter

Chapter 9 explains how investigation runs connect ingested evidence and outcome history to larger operational analysis and reporting.

Source basis: Zahlen 0616A request/response schema export and Zahlen API User Guide v1.0. Example outcome values are illustrative unless defined by the active deployment contract.

ZAHLEN

API USER GUIDE

Chapter 9

Investigation Runs API

Listing runs • Run status • Run detail retrieval

For merchants, developers, and integration engineers

Source baseline: zahlen_deploy_0616A.tar.gz | Version 1.0 | June 2026

Administrative visibility • Tenant-safe processing • Durable operational traceability

Commercial workflow context

Payment Event → Retry Decision → Retry Outcome → Investigation Run → Reporting

Chapter 9 — Investigation Runs API

Learning objectives

By the end of this chapter, you should be able to list tenant-scoped investigation runs, read processing status, retrieve run details, and connect each run to the wider Zahlen evidence pipeline.

Investigation runs are durable administrative records that track how uploaded or API-ingested payment evidence moves through validation, normalization, analysis, and downstream population. They provide the operational bridge between a merchant submission and the reporting, Recovery Intelligence, issuer monitoring, and governance views that follow.

9.1 Where investigation runs fit

Stage	Primary identifier	What it tells you
Payment-event ingestion	payment_event_batch_id / batch_id	Which merchant evidence was accepted for processing.
Background processing	upload_job_id / job_id	Which durable processing job owns the work.
Investigation run	run or job resource	Whether processing completed and what downstream evidence was produced.
Reporting and monitoring	durable record IDs	How the completed evidence appears in Recovery Truth, issuer health, timelines, and reports.

Administrative boundary

Investigation-run routes are under /v1/admin/investigation-runs. Do not assume that a merchant X-API-Key grants access. These routes require an approved administrative context, such as an authenticated operator session or enterprise administrative authorization.

The merchant-facing API returns an upload_job_id during payment-event ingestion. Store that identifier even when your application does not have direct access to the administrative routes. It gives support and operations teams a stable correlation point.

9.2 Confirmed route family

Method	Path	Purpose
GET	/v1/admin/investigation-runs	List investigation runs visible to the authenticated tenant context.
GET	/v1/admin/investigation-runs/{job_id}	Retrieve detailed information for one run.
GET	/v1/admin/investigation-runs/{job_id}/status	Retrieve the current processing status for one run.
GET	/v1/admin/investigation-runs/readiness	Evaluate whether the investigation-run subsystem is ready and properly connected.

9.3 Authentication model

Administrative API access is governed separately from the merchant API. The exact credential or session mechanism depends on the deployment and enterprise contract. The important rule is that tenant ownership must come from authenticated context, not from a tenant_id supplied in the URL, query string, or request body.

- Use the administrative base URL and authentication method supplied by the Zahlen administrator.
- Never add tenant_id to a request merely to make an empty result return data.
- Treat a 401 response as an authentication problem and a 403 response as an authorization or role problem.
- Treat an empty list as a possible valid tenant-scoped result until runtime and population health are checked.

Fail closed

If the platform cannot resolve the authenticated tenant or administrative identity, access should be denied. A production system must not fall back to a default tenant.

9.4 Listing investigation runs

Use the list route to discover recent runs available to the current administrative tenant. A list response commonly acts as an operational index: it helps operators find a job by date, source, status, or upload identifier before opening the detail resource.

```
curl -sS 'https://api.example.com/v1/admin/investigation-runs' \
  -H 'Accept: application/json' \
  -H 'Authorization: Bearer ADMIN_TOKEN_REPLACE_ME' | python -m json.tool
```

The header above is illustrative. Use the administrative authentication contract supplied by your deployment; do not substitute X-API-Key unless that route is explicitly configured to accept it.

9.4.1 What to capture from a list response

Field category	Examples	Client use
Identity	job_id, upload_job_id, run identifier	Open status and detail resources; correlate with ingestion responses.
Ownership	tenant or merchant context	Confirm the record belongs to the authenticated scope.
Lifecycle	status, created_at, started_at, completed_at	Sort recent work and identify stale or unfinished runs.
Volume	total rows, valid rows, invalid rows, error count	Determine whether the run processed the expected evidence.
Source	upload, API ingestion, source label	Trace the run back to the originating integration path.

9.4.2 List-processing pattern

1. Request the list using the approved administrative context.
2. Filter or sort locally only after confirming the server already enforced tenant scope.
3. Select the run using a durable job identifier, not only a human-readable timestamp.
4. Open the status resource for active work and the detail resource for completed or failed work.

9.5 Reading run status

The status route answers a narrow operational question: what state is this job in now? It is appropriate for polling while a run is still processing and for detecting terminal completion or failure.

```
curl -sS \
  'https://api.example.com/v1/admin/investigation-runs/ZN-2026-06-16-0001/status' \
  -H 'Accept: application/json' \
  -H 'Authorization: Bearer ADMIN_TOKEN_REPLACE_ME' | python -m json.tool
```

9.5.1 Status categories

Category	Meaning	Recommended behavior
Queued or pending	Accepted but not actively processing.	Poll at a controlled interval; watch queue age.
Running or processing	Work is active.	Continue polling with increasing intervals; do not submit a duplicate job.
Completed	Primary run processing reached a terminal success state.	Retrieve detail and confirm downstream population.
Failed	The run ended without successful completion.	Read error detail, preserve identifiers, and alert operations.
Unknown or unavailable	The resource is not visible, missing, or status cannot be resolved.	Check tenant context, identifier, authorization, and runtime health.

Completed is not the final diagnostic step

A run can report completion while a downstream bridge or composition layer remains empty. After completion, verify Recovery Truth, radar, issuer health, monitoring events, timelines, cohort memory, and classification persistence when those features are in scope.

9.5.2 Polling guidance

- Start with a moderate interval rather than polling continuously.
- Increase the interval for long-running jobs.
- Stop polling when a terminal state is reached.
- Apply a maximum polling duration and alert when it is exceeded.
- Log job_id, request correlation, timestamps, and the final status.

9.6 Retrieving run details

The detail route provides the richer record needed for troubleshooting, reporting correlation, and governance review. Use it after a run completes, fails, or appears inconsistent with the downstream dashboards.

```
curl -sS \
  'https://api.example.com/v1/admin/investigation-runs/ZN-2026-06-16-0001' \
  -H 'Accept: application/json' \
  -H 'Authorization: Bearer ADMIN_TOKEN_REPLACE_ME' | python -m json.tool
```

9.6.1 Detail categories to inspect

Category	Questions to answer
Identity	Does the job ID match the upload_job_id captured during ingestion?
Tenant and merchant scope	Is the record visible only within the authenticated ownership boundary?
Source evidence	Was the run created from the expected API batch or uploaded evidence?
Row accounting	Do total, valid, invalid, and error counts reconcile?
Lifecycle timing	When was the run created, started, and completed? Is the duration plausible?
Errors and warnings	Are failures actionable, repeatable, and tied to specific evidence?
Downstream outputs	Which durable stores and monitoring layers were populated?

9.6.2 Reconciliation rule

Do not evaluate one count in isolation. Reconcile the evidence chain: submitted rows → accepted rows → valid rows → invalid rows → persisted records → downstream monitoring artifacts. A discrepancy may be valid, but it should be explainable.

Preserve identifiers

Store upload_job_id from the merchant ingestion response and the administrative job identifier returned by the run APIs. These IDs are the most reliable way to connect a customer support case to durable processing evidence.

9.7 End-to-end investigation-run workflow

Step	Developer or operator action	Expected evidence
1	Submit payment events through /v1/payment-events or /v1/payment-events/batch.	batch ID and upload_job_id
2	Store the returned identifiers with merchant-side event records.	durable client correlation
3	List administrative investigation runs when authorized.	tenant-scoped run index
4	Poll the selected run status at a controlled interval.	current lifecycle state
5	Retrieve run detail after terminal completion or failure.	row accounting, timestamps, errors, outputs
6	Verify downstream population and reporting.	Recovery Truth, monitoring, classification, reports
7	Escalate anomalies using IDs, timestamps, and evidence.	auditable incident or support record

9.8 Relationship to the fixed retry schedule

Investigation runs analyze evidence generated by the merchant payment process. They do not change Zahlen's canonical retry schedule. Payment attempts remain governed by the fixed sequence: Day 1, Day 2, Day 6, and Day 14. Administrative polling or rerunning a report must never create an additional payment attempt.

Operation	May repeat automatically?	Payment effect
GET run list	Yes, with reasonable polling limits.	None
GET run status	Yes, with controlled intervals.	None
GET run detail	Yes.	None
Resubmit payment evidence	Only with documented replay safeguards.	May create duplicate processing evidence if IDs are not stable.
Execute payment retry	Only on the fixed Day 1, Day 2, Day 6, Day 14 schedule.	Creates a real authorization attempt.

9.9 Error handling

HTTP status	Likely meaning	Recommended response
200	Request succeeded.	Parse the resource and persist identifiers or status.
401	Administrative authentication is missing or invalid.	Verify the approved credential and environment.
403	The identity is authenticated but not allowed to access the route.	Check role, enterprise entitlement, or endpoint policy.
404	The run is absent or not visible in the authenticated tenant scope.	Verify job ID and tenant context; do not bypass ownership filters.
422	A supplied parameter failed validation.	Correct the request before retrying.
429	Administrative rate limit or quota enforcement.	Back off and honor Retry-After when present.
500/503	Runtime or dependency failure.	Retry GET operations with bounded backoff; alert if sustained.

9.9.1 Empty list troubleshooting

5. Confirm the administrative identity and current environment.
6. Confirm the expected ingestion request returned an upload_job_id.
7. Check the investigation-run readiness route.
8. Check worker, supervisor, queue, and last-cycle health.
9. Verify tenant resolution before considering any data repair.
10. Use backfill only as a controlled remediation after the missing bridge is identified.

Do not bypass tenant isolation

An empty list can be the correct response for the authenticated tenant. Never remove tenant filters or substitute a default production tenant merely to make data appear.

9.10 Example client polling pattern

The following Python example illustrates a bounded status poll. Replace the illustrative bearer token with the administrative authentication contract for your deployment.

```
import os
import time
import requests

BASE_URL = os.environ["ZAHLEN_BASE_URL"]
ADMIN_TOKEN = os.environ["ZAHLEN_ADMIN_TOKEN"]
JOB_ID = "ZN-2026-06-16-0001"

headers = {
    "Accept": "application/json",
    "Authorization": f"Bearer {ADMIN_TOKEN}",
}

interval_seconds = 5
max_interval_seconds = 60
deadline = time.monotonic() + 15 * 60

while time.monotonic() < deadline:
    response = requests.get(
        f"{BASE_URL}/v1/admin/investigation-runs/{JOB_ID}/status",
        headers=headers,
        timeout=20,
    )
    response.raise_for_status()
    payload = response.json()
    status = str(payload.get("status", "")).upper()

    if status in {"COMPLETED", "FAILED"}:
        print(payload)
        break

    time.sleep(interval_seconds)
    interval_seconds = min(max_interval_seconds, interval_seconds * 2)
else:
    raise TimeoutError(f"Investigation run {JOB_ID} did not finish in time")
```

9.10.1 Production improvements

- Add jitter so multiple clients do not poll in lockstep.
- Handle 429 using Retry-After when present.
- Capture request IDs and response timestamps in logs.
- Use a circuit breaker for sustained 5xx failures.
- Retrieve the detail resource after COMPLETED or FAILED.

9.11 Production readiness checklist

☐ Administrative access is explicitly approved and separated from merchant API-key access.

☐ Environment base URLs and credentials are not shared across development, staging, and production.

☐ upload_job_id is stored for every accepted payment-event ingestion request.

☐ Run listing is tenant-scoped and an empty list is handled as a valid possible result.

☐ Status polling is bounded, uses increasing intervals, and stops on terminal states.

☐ Completed runs are followed by downstream population verification when dashboards are expected to contain data.

☐ Failed runs preserve error evidence, identifiers, and timestamps for support or incident review.

☐ No administrative poll, retry, or remediation creates a payment attempt outside Day 1, Day 2, Day 6, and Day 14.

☐ Backfill is used only after identifying the missing population bridge.

☐ Logs do not expose secrets or prohibited cardholder data.

Chapter summary

Investigation runs make background processing observable. Use the list route to find tenant-scoped work, the status route to monitor lifecycle state, and the detail route to reconcile evidence, errors, and downstream outputs. Preserve upload_job_id, respect the administrative authorization boundary, and treat completion as the start of downstream verification—not as proof that every reporting layer is populated.

Key terms

Term	Meaning
upload_job_id	Identifier returned during ingestion that correlates merchant evidence with background processing.
Investigation run	Durable administrative record of processing and downstream population.
Terminal state	A lifecycle state such as COMPLETED or FAILED that ends active polling.
Readiness	Operational evaluation of whether investigation-run services and dependencies are available.
Population bridge	The service path that converts completed evidence into Recovery Truth and monitoring artifacts.

ZAHLEN

API User Guide

Chapter 10 - Webhooks

Outcome contract | Verification | Retry handling

Audience
Merchants, developers, and integration engineers who build or operate webhook consumers for Zahlen events.

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Commercial developer experience | Tenant-safe operations | Explainable retry intelligence

Chapter 10 - Webhooks

Learning objectives
By the end of this chapter, you should be able to create and manage webhook subscriptions, interpret the outcome-delivery contract, verify deliveries using the active deployment policy, and implement retry-safe, duplicate-safe processing.

Webhooks allow Zahlen to send event notifications to a merchant-controlled HTTPS endpoint. Instead of repeatedly polling for a change, the merchant registers a callback URL and one or more event types. When a subscribed event occurs, Zahlen can deliver a request to that callback according to the active deployment contract.

The confirmed merchant-facing subscription surface contains three operations:

Method	Path	Purpose
POST	/v1/webhook-subscriptions	Create a tenant-scoped subscription.
GET	/v1/webhook-subscriptions	List subscriptions visible to the authenticated merchant.
DELETE	/v1/webhook-subscriptions/{subscription_id}	Delete or deactivate a subscription.

Contract boundary
The uploaded schema confirms subscription management fields, but it does not define one universal delivery payload, signature algorithm, signing header, or retry timetable. Production clients must obtain the active webhook outcome contract and verification policy from their Zahlen deployment.

10.1 Why webhooks matter

In the Zahlen commercial workflow, a decision is not the same as an observed result. Webhooks can notify downstream systems that durable outcome or operational evidence is available, reducing polling and helping merchants keep their own records synchronized.

- Use webhooks for notifications, not as the only system of record.
- Persist durable Zahlen identifiers such as decision ID, request ID, outcome ID, event ID, and subscription ID.
- Design for delayed, duplicated, and out-of-order delivery.
- Keep payment scheduling separate from HTTP delivery retries. Webhook retries must never create extra payment attempts outside Zahlen's fixed Day 1, Day 2, Day 6, and Day 14 schedule.

10.2 Subscription contract

All merchant-facing subscription calls use the X-API-Key header. Tenant and merchant ownership are derived from the authenticated key rather than trusted from the JSON body.

Create a subscription

```
curl -sS -X POST "$ZAHLEN_BASE_URL/v1/webhook-subscriptions" \
 -H "Content-Type: application/json" \
 -H "X-API-Key: $ZAHLEN_API_KEY" \
 -d '{
  "callback_url": "https://merchant.example.com/webhooks/zahlen",
  "events": ["REPLACE_WITH_ACTIVE_EVENT_TYPE"]
 }'
```

Illustrative event value

REPLACE_WITH_ACTIVE_EVENT_TYPE is intentionally not a real contract promise. Obtain the enabled event-type catalog from the deployed Zahlen outcome contract before creating a production subscription.

Field	Type	Required	Constraints and meaning
callback_url	string	Yes	Minimum length 8; maximum length 2,048. Use an HTTPS endpoint in production.
events	array[string]	Yes	At least 1 and at most 20 event-type strings.

Unknown top-level properties are rejected because the create model forbids extra fields. This protects integrations from silently sending misspelled or unsupported configuration.

Create response

```
{
 "subscription_id": "whsub_01J...",
 "merchant_id": "merchant_example",
 "tenant_id": "tenant_example",
 "callback_url": "https://merchant.example.com/webhooks/zahlen",
 "events": ["REPLACE_WITH_ACTIVE_EVENT_TYPE"],
 "status": "ACTIVE",
 "created_at": "2026-06-16T14:00:00Z",
 "updated_at": "2026-06-16T14:00:00Z",
 "deleted_at": null
}
```

Example values

Identifier formats and status values above are illustrative. Parse the documented fields, but do not hard-code invented prefixes or status catalogs unless your deployment contract defines them.

10.3 Response fields and lifecycle

Field	Type	Required	Client use
subscription_id	string	Yes	Stable identifier for later deletion, logs, support, and audit.
merchant_id	string	Yes	Merchant context resolved by Zahlen.
tenant_id	string	Yes	Tenant ownership resolved from authentication.
callback_url	string	Yes	Registered destination.
events	array[string]	Yes	Subscribed event types.
status	string	Yes	Current subscription state.
created_at	string	Yes	Creation timestamp.
updated_at	string	Yes	Most recent update timestamp.
deleted_at	string or null	No	Deletion/deactivation timestamp when applicable.

List subscriptions

```
curl -sS "$ZAHLEN_BASE_URL/v1/webhook-subscriptions" \
 -H "X-API-Key: $ZAHLEN_API_KEY"
```

The list response contains merchant_id, tenant_id, count, and subscriptions. Treat an empty list as a valid tenant-scoped result; do not bypass tenant filters to find subscriptions belonging to another account.

Delete or deactivate a subscription

```
curl -sS -X DELETE \
 "$ZAHLEN_BASE_URL/v1/webhook-subscriptions/whsub_01J..." \
 -H "X-API-Key: $ZAHLEN_API_KEY"
```

Delete response field	Type	Meaning
subscription_id	string	Subscription acted upon.
merchant_id	string	Authenticated merchant context.
tenant_id	string	Authenticated tenant context.
deleted	boolean	Whether deletion/deactivation occurred.
status	string	Resulting subscription status.
deleted_at	string or null	Deletion time when supplied.

Operational rule
Store subscription_id locally. Do not discover it by matching callback URLs during an outage or deployment change.

10.4 Outcome contract

The outcome contract describes what Zahlen sends, when it sends it, how the receiver identifies the event, and how authenticity is verified. It is distinct from the subscription-management schema.

Contract area	Questions the production contract must answer
Event catalog	Which event-type strings may be subscribed to? Which versions are active?
HTTP request	Which method, content type, timeout, and headers are used?
Envelope	Where are event ID, delivery ID, event type, version, and creation time located?
Payload	Which outcome, decision, payment-event, and merchant correlation fields are included?
Verification	Which signature algorithm, secret, header names, timestamp rules, and canonical byte sequence apply?
Acknowledgment	Which HTTP response codes count as successful delivery?
Retry policy	Which failures are retried, how often, and for how long?
Replay	How can an authorized operator replay a failed delivery?

Recommended consumer envelope

A robust client should be able to process an envelope with stable delivery metadata and a versioned payload. The following structure is conceptual only and must not replace the active deployment contract:

```
{
 "event_id": "provider-defined stable event identifier",
 "delivery_id": "provider-defined delivery attempt identifier",
 "event_type": "contract-defined event type",
 "schema_version": "contract-defined version",
 "created_at": "ISO-8601 timestamp",
 "data": { "contract-defined payload": true }
}
```

Do not guess
Do not infer field names, event names, or signature headers from this conceptual envelope. Generate production parsing and verification from the actual Zahlen outcome contract.

Correlation with retry outcomes

Where the active event carries retry-outcome evidence, correlate it using durable identifiers instead of customer-readable labels. The retry outcome API can expose outcome_id, request_id, decision_id, token, attempt_number, outcome, processor evidence, timestamp, and matched_by. A webhook consumer should store whichever of these fields are included by the active contract and avoid declaring recovery based only on a scheduled attempt.

10.5 Verification

Verification proves that an inbound request was created by the expected Zahlen deployment and was not modified in transit. Because the 0616A subscription schema does not define a universal signing mechanism, the steps below describe the required security pattern without inventing algorithm-specific details.

1. Read the exact raw request body before JSON parsing or reformatting.
2. Read the contract-defined signature, timestamp, key identifier, and version headers.
3. Reject missing or unsupported verification metadata.
4. Check that the delivery timestamp falls within the allowed replay window.
5. Compute the expected signature using the contract-defined algorithm and canonical input.
6. Compare signatures using a constant-time comparison function.
7. Deduplicate the stable event or delivery identifier before applying business effects.
8. Only then parse and process the payload.

Raw bytes matter
Many signing schemes authenticate the exact HTTP body bytes. Parsing JSON and serializing it again can change whitespace or property ordering and cause valid signatures to fail.

Illustrative verification pseudocode

```
raw_body = request.read_raw_bytes()
metadata = read_contract_headers(request.headers)

if metadata.missing_or_unsupported():
  return HTTP_401

if timestamp_outside_allowed_window(metadata.timestamp):
  return HTTP_401

expected = contract_sign(secret, metadata, raw_body)
if not constant_time_equal(expected, metadata.signature):
  return HTTP_401

envelope = parse_json(raw_body)
if already_processed(envelope.stable_event_id):
  return HTTP_200

enqueue_for_processing(envelope)
return HTTP_200
```

Secret management

- Store verification secrets in a secret manager or protected environment variable.
- Do not log signature secrets, API keys, or complete authorization headers.
- Support secret rotation with an overlap window or key identifier when the contract provides one.
- Use separate verification material for development, staging, and production.
- Restrict callback endpoints to HTTPS and maintain valid TLS configuration.

10.6 Reliable consumer architecture

The callback handler should do as little synchronous work as possible. Long-running business logic increases timeout risk and can cause Zahlen to retry a delivery that actually reached your system.

Stage	Responsibility	Failure behavior
1. Receive	Accept HTTPS request and retain raw bytes.	Reject malformed transport safely.
2. Verify	Authenticate signature and replay timestamp.	Return the contract-defined authentication failure.
3. Deduplicate	Reserve stable event ID in durable storage.	Previously completed event returns success without reapplying effects.
4. Persist	Store envelope, headers needed for audit, and receive time.	Do not acknowledge if durable acceptance failed.
5. Enqueue	Place work on an internal queue.	Use transactional outbox/inbox patterns where practical.
6. Acknowledge	Return success quickly.	Use only success codes recognized by the active contract.
7. Process	Apply idempotent business logic asynchronously.	Retry internally without requiring another external delivery.

Separate receipt from business completion
A successful webhook response should ordinarily mean the event was verified and durably accepted, not that every downstream workflow has finished.

Deduplication record

Stored value	Purpose
stable_event_id	Prevents duplicate business effects across repeated deliveries.
delivery_id	Supports per-attempt diagnostics when the contract supplies it.
event_type and schema_version	Selects the correct parser and handler.
received_at and processed_at	Measures delivery and processing lag.
payload hash	Supports integrity and duplicate diagnostics without storing excess sensitive data.
processing status and error	Supports internal retry and operations.

10.7 Retry handling

Webhook retry handling has two sides: Zahlen may retry delivery when the callback is unavailable, and the merchant may retry internal processing after the callback has been durably accepted. Both sides must be duplicate-safe.

Provider delivery retries

- Expect the same logical event more than once.
- Do not use arrival count as a business quantity.
- Return the contract-defined success response for an already-processed event.
- Do not deliberately fail duplicates to force another delivery.
- Use administrative operations for authorized replay rather than editing delivery records directly.

Merchant internal retries

- Retry downstream database, queue, notification, or reporting work using the stored event ID.
- Make every side effect idempotent: use unique keys, compare-and-set transitions, or transactional records.
- Apply bounded exponential backoff with jitter to transient internal failures.
- Move repeatedly failing work to a dead-letter or manual-review queue without losing the original envelope.

```
delay = min(max_delay, base_delay * (2 ** retry_number))
delay = delay * random.uniform(0.75, 1.25)
```

HTTP response guidance

Condition	Typical receiver behavior	Why
Valid and durably accepted	Return an allowed success code quickly.	Prevents unnecessary provider retries.
Valid duplicate already processed	Return an allowed success code.	Duplicate delivery is normal network behavior.
Invalid signature or stale replay	Return the contract-defined authentication failure.	Do not process unauthenticated content.
Malformed payload	Return the contract-defined client failure and quarantine evidence.	Blind retries usually cannot repair invalid content.
Temporary inability to persist	Return a retryable failure only if the contract defines it.	Provider retry may recover a transient outage.
Downstream business system unavailable after durable acceptance	Acknowledge and retry internally.	Avoid coupling callback latency to downstream health.

Payment retry boundary
A duplicate or delayed webhook must not trigger an additional card authorization. Payment attempts remain governed by the fixed Day 1, Day 2, Day 6, and Day 14 schedule and the explicit Zahlen decision/outcome workflow.

10.8 Ordering, versions, and schema evolution

Network delivery does not guarantee that related events arrive in the order they were created. Consumers should use timestamps, versions, and durable state transitions instead of assuming arrival order is business order.

- Ignore or quarantine unsupported event types rather than treating them as a known event.
- Route each schema_version to a compatible parser when version metadata exists.
- Allow additive optional fields without breaking parsing, while still validating required fields.
- Do not overwrite newer local state with an older event solely because it arrived later.
- Keep contract tests for current and prior supported payload versions.

10.9 Test plan

Test	Expected result
Valid signed delivery	Verified, stored, queued, and acknowledged once.
Duplicate valid delivery	No duplicate business effect; successful acknowledgment.
Invalid signature	Rejected before payload processing.
Missing verification metadata	Rejected according to the active contract.
Old replay timestamp	Rejected outside allowed replay window.
Unknown event type	Safely ignored or quarantined with an operational signal.
Out-of-order related events	State remains correct and does not regress.
Temporary database outage before persistence	Retryable response according to contract.
Downstream outage after persistence	Callback succeeds; internal work retries.
Consumer timeout	No partial untracked business effect.
Secret rotation overlap	Deliveries signed with permitted current/previous keys verify correctly.
Deleted subscription	No new deliveries expected after contract-defined deactivation behavior.

10.10 Implementation examples

Python callback skeleton

```
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.post("/webhooks/zahlen")
def zahlen_webhook():
    raw_body = request.get_data(cache=True)
    metadata = read_contract_headers(request.headers)

    if not verify_with_active_contract(raw_body, metadata):
        return jsonify({"accepted": False}), 401

    envelope = parse_and_validate_contract_payload(raw_body)
    event_id = stable_event_id(envelope)

    if inbox_already_completed(event_id):
        return jsonify({"accepted": True, "duplicate": True}), 200

    persist_and_enqueue_atomically(event_id, envelope, metadata)
    return jsonify({"accepted": True}), 200
```

JavaScript callback skeleton

```
app.post("/webhooks/zahlen", rawBodyMiddleware, async (req, res) => {
  const rawBody = req.body;
  const metadata = readContractHeaders(req.headers);

  if (!verifyWithActiveContract(rawBody, metadata)) {
    return res.status(401).json({ accepted: false });
  }

  const envelope = parseAndValidateContractPayload(rawBody);
  const eventId = stableEventId(envelope);

  if (await inboxAlreadyCompleted(eventId)) {
    return res.status(200).json({ accepted: true, duplicate: true });
  }

  await persistAndEnqueueAtomically(eventId, envelope, metadata);
  return res.status(200).json({ accepted: true });
});
```

Framework warning

Some web frameworks parse JSON before your handler runs. Configure raw-body capture for the webhook route if the active verification algorithm signs the exact request bytes.

10.11 Operations and troubleshooting

Symptom	Likely checks
No deliveries	Subscription status, enabled event types, callback URL, tenant/key context, and whether the triggering event occurred.
Repeated deliveries	Receiver timeout, non-success response, persistence failure, or expected at-least-once behavior.
Every signature fails	Wrong environment secret, body mutation, incorrect canonical input, clock skew, or unsupported contract version.
Some signatures fail	Secret rotation, multiple key IDs, proxy body transformation, or timestamp parsing.
High processing lag	Synchronous callback work, queue backlog, downstream dependency failure, or insufficient workers.
Duplicate business actions	Missing durable inbox key or non-idempotent downstream operation.
Events appear out of order	Normal network behavior; use event time/version and monotonic state transitions.
Deleted endpoint still receives requests	In-flight delivery, deactivation semantics, cache/propagation delay, or another active subscription.

Production readiness checklist

- Callback URL uses HTTPS and has valid TLS.
- Production event types come from the active Zahlen outcome contract.
- Raw request body is retained long enough for verification.
- Verification rejects missing, invalid, and stale authentication metadata.
- Secrets are stored securely and rotation is tested.
- Durable deduplication is performed before business effects.
- Callback persistence and acknowledgment complete quickly.
- Downstream work is asynchronous and independently retryable.
- Unknown event types and schema versions are quarantined safely.
- Duplicate, delayed, out-of-order, and replayed deliveries are tested.
- Alerts cover signature failures, delivery failure rate, processing lag, and dead-letter growth.
- Webhook handling cannot create payment attempts outside Day 1, Day 2, Day 6, and Day 14.

Chapter summary
Create, list, and delete subscriptions with X-API-Key. Treat the deployed outcome contract as authoritative for event names, payloads, verification, acknowledgments, and retry policy. Verify raw deliveries, persist before acknowledging, deduplicate with durable identifiers, process asynchronously, and keep webhook retries completely separate from payment retries.

Source basis: Zahlen 0616A request/response schema export and Zahlen API User Guide v1.0. Deployment-specific delivery and verification details must be obtained from the active Zahlen webhook outcome contract.

ZAHLEN

API User Guide

Chapter 11 - Error Handling

HTTP status codes | Validation errors | Authentication failures

Audience
Merchants, developers, and integration engineers who build resilient client applications against the Zahlen API.

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Commercial developer experience | Tenant-safe operations | Explainable retry intelligence

Chapter 11 - Error Handling

Learning objectives

By the end of this chapter, you should be able to distinguish transport, authentication, authorization, validation, throttling, conflict, and server errors; decide whether a request is safe to retry; and preserve the identifiers needed for support and audit.

A dependable API client does not treat every non-success response the same way. A malformed JSON request needs a code change. A revoked API key needs credential remediation. A rate-limit response needs controlled backoff. A transient server failure may be safe to retry only when the operation has stable idempotency semantics.

The first rule is simple: classify the failure before deciding what to do next. Blind retries can create duplicate work, trigger retry storms, consume quotas, or obscure the original cause.

Payment schedule boundary

HTTP request retries are not payment retries. Client error recovery must never create authorization attempts outside Zahlen's fixed Day 1, Day 2, Day 6, and Day 14 payment schedule.

11.1 Error-handling goals

- Give developers enough detail to correct a request without exposing secrets or internal implementation data.
- Preserve request IDs, event IDs, batch IDs, decision IDs, outcome IDs, and upload job IDs for traceability.
- Retry only transient failures and only with bounded backoff, jitter, and stable idempotency.
- Fail closed when authentication or tenant ownership cannot be resolved.
- Make errors observable through logs, metrics, alerts, and audit records.

11.2 HTTP status codes

The table below describes the status classes a Zahlen client should be prepared to handle. Exact response bodies may vary by route, but the client behavior should remain consistent.

Status	Meaning	Typical cause	Recommended client response
200 / 201	Request succeeded	Resource read, created, or accepted.	Parse the body and persist returned identifiers.
400	Malformed or business-invalid request	Invalid JSON, incompatible values, or a business rule failure.	Correct the request. Do not blindly retry.
401	Missing or invalid authentication	Missing X-API-Key, malformed key, revoked key, or wrong environment.	Stop the request flow and repair credentials.
403	Authenticated but not permitted	Plan, role, capability, or endpoint policy denies access.	Check authorization and contract settings.
404	Resource not visible or not found	Wrong identifier, wrong tenant, deleted resource, or wrong environment.	Verify tenant-scoped identifiers and hostname.
409	Conflict or idempotency mismatch	Same idempotency key used with a different logical request or state conflict.	Compare the original request and key; do not generate another payment attempt.
422	Schema validation failure	Missing required field, wrong type, invalid bounds, or forbidden extra field.	Read field-level errors and fix serialization.
429	Rate or quota enforcement	Short-window rate limit or longer-window quota exhausted.	Honor Retry-After when present; back off with jitter.
500	Unexpected server failure	Unhandled server condition.	Retry cautiously with idempotency; alert if repeated.
503	Service or dependency unavailable	Maintenance, dependency outage, worker issue, or overload.	Use bounded backoff; stop after a configured limit.

Do not assume every 4xx is permanent

Some 401, 403, or 404 responses result from using the wrong environment or tenant-scoped identifier. Diagnose the context before changing application logic.

11.3 Reading an error response

The deployed API includes an ApiErrorResponse model with top-level error and meta objects. Clients should preserve the entire safe response for diagnostics, while avoiding logs that expose credentials or prohibited payment data.

```
{
 "error": {
  "code": "EXAMPLE_CODE",
  "message": "Human-readable explanation",
  "details": {"field": "example"}
 },
 "meta": {
  "request_id": "req_example",
  "time": "2026-06-16T15:00:00Z"
 }
}
```

Illustrative structure
The sample above demonstrates a practical parsing pattern. Treat the actual response contract returned by the route as the source of truth and tolerate additional documented metadata.

What to capture

Value	Why it matters
HTTP method and path	Identifies the failing operation without logging secrets.
Status code	Primary classification for client behavior.
Safe error code and message	Supports remediation and alert grouping.
Server request ID	Connects merchant logs to Zahlen audit and support records.
Client correlation IDs	Links event, batch, decision, outcome, and job flows.
Attempt count and elapsed time	Shows retry behavior and retry-storm risk.
Environment and service version	Helps identify wrong-host and contract-version problems.

What not to log

- The complete X-API-Key or webhook verification secret.
- Full primary account numbers, CVV values, passwords, or raw bank credentials.
- Unredacted Authorization headers or secret-manager values.
- Arbitrary metadata without a reviewed allowlist.

11.4 Validation errors

Key Zahlen request models use strict validation and forbid unknown top-level properties. This prevents misspelled or unsupported fields from being silently ignored. A validation failure is a client defect or contract mismatch, not a transient network event.

Common causes

Cause	Example	Correction
Missing required field	Payment event without event_id.	Add the required field before sending.
Wrong data type	attempt_number sent as an object.	Serialize the documented integer type.
Out-of-range value	attempt_number below its minimum.	Validate bounds in the client model.
Empty collection	Payment-events request with no events.	Send at least one event.
Oversized collection	More than 10,000 payment events or more than 500 legacy batch decisions.	Split into valid batches and respect quotas.
Forbidden extra field	Misspelled property such as eventid.	Correct the field name; do not expect it to be ignored.
Invalid URL or string length	Webhook callback URL outside schema constraints.	Validate before submission.

Example: invalid payment event

```
{
 "events": [{
  "eventid": "evt_0001",
  "attempt_number": 0
 }]
}
```

This request has two problems: eventid is not the required event_id field, and attempt_number violates the payment-event minimum of 1. Because unknown fields are forbidden, the misspelled field is not silently accepted.

Client-side validation pattern

1. Build request objects from strict typed models.
2. Validate required fields, types, bounds, and collection sizes before network transmission.
3. Serialize once and contract-test the exact JSON shape.
4. On HTTP 422, map returned field errors to developer-visible diagnostics.
5. Do not retry until the payload is corrected.

11.5 Authentication failures

Merchant-facing routes authenticate with the X-API-Key header. Zahlen derives merchant, tenant, and actor context from the key. A client must not attempt to replace missing authentication by supplying tenant_id in JSON, query strings, or forms.

```
X-API-Key: zk_live_REPLACE_ME
```

Diagnostic sequence for HTTP 401

6. Confirm the X-API-Key header is present and spelled exactly.
7. Confirm the secret was loaded from the intended secret manager or environment variable.
8. Confirm the base URL belongs to the same environment as the key.
9. Confirm the key is active and has not been revoked or expired.
10. Check whether whitespace, quotes, line breaks, or proxy configuration altered the header.
11. Use the key identifier or safe fingerprint to review activity without exposing the secret.
12. Rotate immediately if compromise is suspected.

Fail closed
When a key cannot be resolved to a valid tenant context, the request must be denied. Never fall back to a default production tenant.

401 versus 403

Response	Interpretation	Example response action
401 Unauthorized	The caller is not successfully authenticated.	Repair or rotate the credential.
403 Forbidden	The caller is authenticated but lacks permission for the route or capability.	Check plan, role, endpoint authorization, and contract.

Security response to suspected compromise

- Revoke the affected key and create a replacement through an approved administrative workflow.
- Deploy the replacement to all intended services and verify authenticated traffic.
- Review audit and activity records for unexpected endpoints, tenants, IPs, or volumes.
- Do not paste the key into support messages, tickets, screenshots, or command history.

11.6 Resource, conflict, and ownership errors

HTTP 404

A resource can be absent, deleted, located in another environment, or intentionally invisible because it belongs to another tenant. A tenant-scoped 404 should not cause a client to remove ownership filters or try identifiers from another merchant.

- Verify the exact event_id, batch_id, subscription_id, decision_id, outcome_id, or job_id.
- Verify the base URL and environment.
- Verify that the same authenticated tenant created or owns the resource.
- Check whether the resource is eventually available after asynchronous processing, but poll at a controlled interval.

HTTP 409

A conflict commonly indicates an idempotency mismatch or incompatible resource state. The safe response is to compare the request with the original operation, not to generate a fresh key automatically.

```
Idempotency-Key: merchant-order-8842-attempt-2
```

Idempotency rule
Use one stable idempotency key for one logical operation. Reusing the key with changed request data can correctly produce a conflict. Generating a new key to bypass that conflict can create duplicate decisions or duplicate outcome records.

Conflict checklist

13. Locate the original request body and idempotency key.
14. Compare the current body byte-for-byte or field-for-field with the original logical operation.
15. Read the existing resource if the API exposes it.
16. Continue from the known durable result rather than creating a duplicate.
17. Escalate if the original result cannot be reconciled safely.

11.7 Safe retry strategy

Automatic retry behavior should be explicit for each operation. Retry budgets must be bounded by maximum attempts, maximum elapsed time, and circuit-breaker rules.

Operation / response	Automatic retry?	Required safeguards
GET resource + transient 5xx/503	Usually	Exponential backoff, jitter, maximum attempts.
POST retry decision + transient failure	Yes, carefully	Reuse the same Idempotency-Key and identical logical request.
POST retry outcome + transient failure	Yes, carefully	Reuse stable identifiers and idempotency where supported.
POST payment-event batch + uncertain result	Only with safeguards	Use stable event IDs; first attempt to read the resulting resource or reconcile ingestion.
HTTP 400 or 422	No	Fix the request before resubmission.
HTTP 401	No	Repair authentication or rotate the key.
HTTP 403	No	Resolve authorization or plan restrictions.
HTTP 404	Usually no	Verify identifier, ownership, environment, and asynchronous timing.
HTTP 409	No blind retry	Reconcile the original idempotent operation.
HTTP 429	Yes, later	Honor Retry-After; use bounded backoff and jitter.

Exponential backoff with jitter

```
delay = min(max_delay, base_delay * (2 ** retry_number))
delay = delay * random.uniform(0.75, 1.25)
```

- Use a maximum number of attempts and maximum total elapsed time.
- Share throttling state across workers when many instances use the same tenant quota.
- Stop retrying when a circuit breaker detects a sustained outage.
- Preserve the same idempotency key for the same logical POST operation.
- Never let network retry loops schedule additional payment attempts.

11.8 Implementation examples

Python error mapper

```
import time
import random
import requests

RETRYABLE = {429, 500, 503}

def request_with_policy(method, url, *, headers, json=None, max_attempts=4):
  for attempt in range(max_attempts):
    response = requests.request(
      method, url, headers=headers, json=json, timeout=20
    )
    if response.ok:
      return response

    request_id = response.headers.get("X-Request-ID")
    if response.status_code not in RETRYABLE:
      raise RuntimeError(
        f"Zahlen HTTP {response.status_code}; request_id={request_id}; "
        f"body={response.text[:1000]}"
      )

    if attempt == max_attempts - 1:
      response.raise_for_status()

    retry_after = response.headers.get("Retry-After")
    if retry_after and retry_after.isdigit():
      delay = float(retry_after)
    else:
      delay = min(30.0, 0.5 * (2 ** attempt))
      delay *= random.uniform(0.75, 1.25)
    time.sleep(delay)

  raise RuntimeError("unreachable")
```

Production note

Use structured logs and redact response content according to your data policy. The example truncates the body but does not replace a proper allowlist-based logging design.

JavaScript status handling

```
const response = await fetch(url, options);
const requestId = response.headers.get("x-request-id");

if (response.ok) {
 return await response.json();
}

const body = await response.text();

switch (response.status) {
 case 401:
  throw new Error(`Authentication failed; request_id=${requestId}`);
 case 403:
  throw new Error(`Not permitted; request_id=${requestId}`);
 case 422:
  throw new Error(`Validation failed; request_id=${requestId}; ${body}`);
 case 429:
  throw new Error(`Throttled; retry-after=${response.headers.get("retry-after")}`);
 default:
  throw new Error(`Zahlen HTTP ${response.status}; request_id=${requestId}`);
}
```

11.9 Monitoring and alerting

Error handling is incomplete until failures are observable. Aggregate by tenant, route, status class, error code, key identifier, and deployment version while protecting sensitive information.

Signal	What it may indicate	Suggested response
401 rate	Revoked, expired, missing, or misconfigured keys.	Check recent deployments and key activity.
403 rate	Plan or authorization mismatch.	Review capability and endpoint policy.
422 rate	Client release or schema drift.	Inspect field errors and contract tests.
429 rate	Capacity pressure, quota exhaustion, or request loop.	Throttle clients and review usage.
5xx / 503 rate	Runtime or dependency degradation.	Open incident and use circuit breakers.
Idempotent replay rate	Expected retry behavior or unstable client network.	Verify that replays are returning the same durable result.
Outcome-reporting lag	Broken recovery learning loop.	Check retry-outcome clients and queues.

Alert design

- Alert on sustained rates or error-budget impact, not every individual validation error.
- Use separate alerts for authentication spikes, throttling, server failures, and outcome-reporting lag.
- Include safe correlation identifiers and links to internal dashboards, never complete secrets.
- Suppress duplicate alerts during a known incident while preserving metrics and logs.

Support package
When escalating to Zahlen support, provide environment, UTC time window, method, path, status, safe error code, request ID, and relevant durable identifiers. Do not provide the full API key or prohibited payment data.

11.10 Troubleshooting playbooks

Validation failures after a client deployment

18. Compare the new serialized JSON with the previous known-good payload.
19. Check renamed, missing, nullable, and forbidden fields.
20. Validate collection sizes and numeric bounds.
21. Run contract tests against the current discovery schema.
22. Roll back or correct the client; do not add blind retry logic.

Authentication failures across all routes

23. Check base URL and environment.
24. Check secret injection and header construction.
25. Check key status, rotation timing, and revocation records.
26. Verify the system clock and proxy/header forwarding behavior where relevant.
27. Rotate if compromise or accidental disclosure is possible.

Repeated 429 responses

28. Stop immediate retries and honor Retry-After when provided.
29. Measure traffic by service, route, and key identifier.
30. Look for a retry loop or duplicated worker deployment.
31. Check plan assignment, quota configuration, and current usage.
32. Increase capacity or quota only after explaining the traffic pattern.

Repeated 5xx or 503 responses

33. Enable bounded backoff and a circuit breaker.
34. Preserve idempotency keys and request correlation.
35. Check Zahlen health and version endpoints when reachable.
36. Stop before exceeding the retry budget.
37. Escalate with a safe support package and UTC timestamps.

Final rule
A successful recovery client is conservative: it validates before sending, authenticates securely, retries only when safe, preserves durable identifiers, and never converts transport uncertainty into an extra payment attempt.

11.11 Production readiness checklist

- Every API call has a documented status-handling policy.
- Validation is performed locally with strict typed request models.
- Unknown fields fail tests before reaching production.
- API keys are never logged and 401 handling stops automatic retries.
- 409 handling reconciles the original idempotent operation.
- 429 handling honors Retry-After and uses bounded jittered backoff.
- POST retries reuse the same stable idempotency key where supported.

ZAHLEN

API User Guide

Chapter 12 - Audit & Compliance

Audit trail | Tenant isolation | Governance controls

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Commercial developer experience | Tenant-safe operations | Explainable retry intelligence

Chapter 12 - Audit & Compliance

Learning objectives

By the end of this chapter, you should be able to explain how Zahlen records API activity, enforces tenant isolation, and uses governance controls to support secure, traceable commercial operation.

Audit and compliance are not separate from API design. They are built into how requests are authenticated, how ownership is resolved, how actions are recorded, and how operators verify that controls continue to work. For merchants and integration teams, these capabilities make it possible to answer practical questions such as: Who called the API? Which key was used? Which tenant owned the resource? What decision was returned? Was a quota enforced? What changed after an administrative action?

Zahlen provides the technical evidence needed for operational review and governance. It does not replace a merchant's own legal, privacy, security, or payment-compliance obligations. Each customer remains responsible for deciding what data may be transmitted, how long local records are retained, and which staff may access them.

Core ownership rule

tenant_id is the ownership boundary. merchant_id is business data. Client applications must never attempt to select a tenant by placing tenant_id in a request body, query string, or form. Zahlen resolves ownership from authenticated context.

12.1 Why auditability matters

- Incident response: correlate a reported problem with a request, API key, endpoint, decision, outcome, batch, or investigation run.
- Security review: identify unusual key activity, failed authentication, access denials, and post-revocation attempts.
- Operational accountability: show who changed plans, quotas, policies, keys, or workflow state and when the change occurred.
- Customer support: trace a merchant event through ingestion, decisioning, retry outcome, investigation, and reporting.
- Governance certification: prove that tenant isolation, rate limits, quotas, audit logging, and portal controls are active and tested.

Question	Evidence commonly used
Who made the call?	Actor identity, API key ID or safe fingerprint, authenticated user or service identity
What operation ran?	HTTP method, endpoint, action name, resource type, request ID
Which tenant owned it?	Resolved tenant_id and merchant context from

Question	Evidence commonly used
	authentication
What happened?	Status code, result, decision ID, outcome ID, before-and-after state
When did it happen?	Server timestamp, created_at, updated_at, processing and completion times
How long did it take?	Latency, queue timing, background-job timing, downstream status

12.2 Audit trail

The Zahlen commercial governance layer can record API activity and privileged administrative actions as durable audit evidence. A useful audit record connects the authenticated caller to the operation and its result without storing the full secret API key or unnecessary payment data.

Audit element	Purpose	Client responsibility
request_id	Correlates one API exchange across services and support	Capture it in logs and support tickets
api_key_id / fingerprint	Identifies the credential without exposing the secret	Never log the full X-API-Key value
tenant_id	Confirms ownership scope	Do not supply or override it from merchant JSON
endpoint and method	Shows what operation was attempted	Log route and HTTP method
status_code	Separates success, validation, authorization, throttling, and server failure	Classify before retrying
latency	Supports runtime and performance analysis	Monitor percentiles and sustained change
resource identifiers	Connects event, batch, decision, outcome, and run records	Persist durable IDs returned by Zahlen
before/after state	Explains privileged changes	Record administrative changes through approved routes

12.3 Correlation identifiers

A complete audit story depends on preserving the identifier chain created across the commercial workflow.

Identifier	Created by	Use in audit and support
event_id	Merchant	Stable correlation for one payment event
payment_event_batch_id / batch_id	Zahlen	Groups one ingestion request
upload_job_id	Zahlen	Links ingestion to background processing and investigation runs
request_id	Zahlen	Correlates an API request and support investigation
decision_id	Zahlen	Identifies the retry decision that controlled merchant behavior
outcome_id	Zahlen	Identifies the recorded real-world result
subscription_id	Zahlen	Identifies a webhook subscription
investigation run ID	Zahlen	Links completed evidence to downstream analysis

```
{
  "event_id": "evt_20260616_0001",
  "request_id": "req_example_001",
  "decision_id": "dec_example_001",
  "outcome_id": "out_example_001",
  "upload_job_id": "job_example_001"
}
```

Logging rule
Log durable identifiers, timestamps, status, and safe credential identifiers. Do not log full API keys, full card numbers, CVV, passwords, or raw bank credentials.

12.4 Tenant isolation

Tenant isolation prevents one customer from reading, changing, or inferring another customer's data. Zahlen derives tenant ownership from the authenticated API key for merchant routes and from authenticated session or administrative context for operator routes. Repository queries and service calls must continue carrying that tenant context all the way to persistence and rendering.

Layer	Tenant-isolation expectation
Request authentication	Resolve the tenant from X-API-Key or approved administrative identity
Request model	Do not expose tenant_id as a merchant-controlled ownership selector

Layer	Tenant-isolation expectation
Service layer	Pass tenant context explicitly to downstream services
Repository layer	Filter reads and writes by tenant_id plus the resource identifier
HTML and JSON rendering	Return only records visible to the authenticated tenant
Exports and reports	Preserve the same tenant filters used by interactive views
Background workers	Load tenant scope from durable job ownership, not a global default
Errors	Fail closed when tenant context is missing, unresolved, or inconsistent

12.5 Understanding tenant-safe 404 responses

A resource owned by another tenant should normally appear unavailable. Returning 404 instead of revealing that the resource exists reduces cross-tenant information leakage. A client must not interpret every 404 as proof that an identifier was never created; it can also mean that the authenticated tenant is not allowed to see it.

1. Confirm that the client is using the expected environment and hostname.
2. Confirm that the correct API key is loaded.
3. Confirm the event, batch, decision, outcome, or run identifier.
4. Check whether the resource was created under another merchant account or environment.
5. Use request IDs and audit records for support; do not weaken repository filters to make the record visible.

12.6 Governance controls

Governance controls turn security and operational expectations into enforceable, reviewable behavior. They cover the entire commercial API lifecycle rather than only authentication.

Control family	What it protects	Evidence to review
API key management	Credential creation, hashing, rotation, and revocation	Key lifecycle records, audit events, revoked-key tests
Usage plans	Contracted capabilities and service levels	Plan assignment, capability checks, negative tests
Rate limits and quotas	Runtime capacity and contracted usage	Enforcement records, 429 behavior, quota checks
Audit trail	Traceability and accountability	Request correlation, status, latency, actor, tenant
Tenant isolation	Customer ownership boundary	Cross-tenant tests, repository filters, fail-closed behavior

Control family	What it protects	Evidence to review
Developer portal controls	Authorized visibility and self-service actions	Role checks, tenant-scoped pages and APIs
Runtime health	Continued operation of workers and services	Health status, queue state, last cycle, failures
Governance certification	Evidence that required controls are present and tested	Certification results, runbooks, restart durability

12.7 Governance certification

Governance certification is an evidence-based review of whether the commercial API is safe to operate. It is not a marketing label and it should not be treated as permanent. Certification must be repeated after material changes to authentication, tenant ownership, persistence, quotas, portal authorization, or deployment topology.

6. Confirm that every merchant route derives ownership from the authenticated key.
7. Run negative cross-tenant tests for reads, writes, exports, and administrative actions.
8. Verify key generation, secure hashing, rotation overlap, revocation, and post-revocation denial.
9. Verify plan restrictions, rate-limit enforcement, quota exhaustion, and HTTP 429 behavior.
10. Confirm that audit records contain sufficient correlation without exposing credentials or prohibited data.
11. Verify restart durability for jobs, outcomes, monitoring state, and governance records.
12. Confirm that missing tenant context fails closed and never falls back to a default production tenant.

Fail closed
A missing tenant context, unresolved key, or failed ownership check must deny access. Production code must not fall back to a development tenant, global tenant, or first available record.

12.8 Data minimization and payment data

Auditability does not require copying every request field into logs. Zahlen request models include payment tokens, merchant references, issuer evidence, and optional metadata. These fields should be used carefully. A generic metadata object is not permission to store prohibited cardholder data.

Prefer	Avoid
Merchant-generated payment tokens	Full primary account numbers
Stable customer, subscription, order, and invoice references	CVV or security codes
Issuer BIN and normalized decline evidence when contractually approved	Passwords, authentication secrets, or raw bank credentials
Minor-unit monetary values and currency	Unstructured dumps of processor payloads containing sensitive data

Prefer	Avoid
Safe API key IDs or fingerprints	Complete X-API-Key values
Request and resource identifiers	Screenshots as the only evidence

12.9 Merchant and integration-team responsibilities

- Classify the data your application sends and confirm that it is allowed by contract and policy.
- Restrict production API keys to server-side applications and approved operators.
- Maintain local logs that can correlate with Zahlen request IDs without storing secrets.
- Define retention and deletion policies for local payment-event, decision, and outcome records.
- Monitor authentication failures, 403/404 anomalies, 409 conflicts, 429 enforcement, and sustained 5xx responses.
- Rotate keys through a controlled overlap procedure and preserve evidence of the change.
- Report observed retry outcomes accurately so governance and performance reporting are based on real evidence.
- Ensure HTTP or webhook retries never create payment attempts outside Zahlen's fixed Day 1, Day 2, Day 6, and Day 14 schedule.

12.10 Practical audit logging example

The following Python example logs safe correlation data while excluding the API key and request body. Production applications should use structured logging and a centralized log platform.

```
import logging
import os
import requests

log = logging.getLogger("zahlen.client")
base_url = os.environ["ZAHLEN_BASE_URL"]
headers = {
    "X-API-Key": os.environ["ZAHLEN_API_KEY"],
    "Content-Type": "application/json",
    "Idempotency-Key": "order-8842-attempt-2",
}

response = requests.post(
    f"{base_url}/v1/_next/retry-decision",
    headers=headers,
    json={"attempt_number": 2, "decline_code": "51"},
    timeout=20,
)

request_id = response.headers.get("X-Request-ID")
body = response.json() if response.content else {}
log.info(
    "zahlen_api_call",
    extra={
        "method": "POST",
        "path": "/v1/_next/retry-decision",
        "status": response.status_code,
        "request_id": request_id or body.get("request_id"),
        "decision_id": body.get("decision_id"),
        "idempotency_key": "order-8842-attempt-2",
    },
```

```
)
response.raise_for_status()
```

Do not copy this literally into every environment

Review whether idempotency keys, merchant references, or other identifiers are considered sensitive in your organization. Apply masking and retention rules appropriate to your contract.

12.11 Audit and compliance review checklist

Review area	Ready when
Authentication	All merchant calls use X-API-Key over HTTPS; secrets are not logged
Ownership	tenant_id comes only from authenticated context
Correlation	Request, event, batch, decision, outcome, and job IDs are retained
Validation	Unknown fields and invalid values are rejected and monitored
Key lifecycle	Creation, rotation, revocation, and unusual use are auditable
Quotas	Plan, usage, quota checks, and 429 activity are reviewed together
Tenant tests	Cross-tenant reads, writes, exports, and actions fail
Sensitive data	Logs and metadata exclude prohibited cardholder and credential data
Runtime	Audit persistence and governance controls survive service restart
Change management	Material control changes trigger contract and governance tests

12.12 Summary

Zahlen audit and compliance controls make the commercial workflow traceable without weakening tenant isolation. The audit trail records who did what, when, through which credential and endpoint, and with what result. Tenant ownership is derived from authenticated context and preserved through services, repositories, pages, APIs, exports, and background work. Governance controls verify that keys, plans, quotas, audit logging, portal authorization, and fail-closed behavior continue to operate as designed.

For integration teams, the practical rule is straightforward: preserve safe identifiers, protect secrets, minimize sensitive data, treat tenant boundaries as absolute, and use approved API operations so that every important action remains explainable and auditable.

Source note

This chapter is based on the Zahlen API User Guide v1.0, the Zahlen Platform Administration Guide v1.0, and the 0616A request/response schema export. Configurable retention periods, plan entitlements, verification policies, and certification procedures remain deployment- or contract-specific.

ZAHLEN

API User Guide

Chapter 13 - SDK Examples

Python | curl | JavaScript

Audience
Merchants, developers, and integration engineers implementing Zahlen in server-side applications, automation scripts, and integration services.

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter 13 - SDK Examples

Learning objectives

By the end of this chapter, you should be able to call the core Zahlen endpoints from curl, Python, and JavaScript; centralize authentication; preserve idempotency and correlation identifiers; and implement safe timeout and error behavior.

The examples in this chapter are intentionally small enough to understand and strong enough to become the foundation of a production client. They use environment variables for secrets, JSON request bodies, explicit timeouts, and stable identifiers.

Canonical payment schedule

These code examples transport events, decisions, and outcomes. They must never be used to create payment attempts outside Zahlen's fixed Day 1, Day 2, Day 6, and Day 14 schedule.

13.1 Common environment variables

```
export ZAHLEN_BASE_URL="https://api.example.com"
export ZAHLEN_API_KEY="zk_live_REPLACE_ME"
```

- Use a secret manager or protected runtime environment for ZAHLEN_API_KEY.
- Use separate base URLs and keys for development, staging, and production.
- Never expose the key in browser code, a mobile binary, source control, screenshots, or support tickets.

13.2 curl examples

curl is useful for connectivity tests, smoke tests, and reproducing an integration issue. Use -sS for clean output and --fail-with-body when available so error bodies remain visible.

Health and version

```
curl -sS "$ZAHLEN_BASE_URL/v1/health" | python -m json.tool

curl -sS "$ZAHLEN_BASE_URL/v1/version" | python -m json.tool
```

Submit a payment event

```
curl --fail-with-body -sS -X POST \
 "$ZAHLEN_BASE_URL/v1/payment-events" \
 -H 'Content-Type: application/json' \
 -H "X-API-Key: $ZAHLEN_API_KEY" \
 -d '{
   "source": "billing_platform",
   "events": [{
     "event_id": "evt_20260616_0001",
     "decline_code": "51",
     "issuer_bin": "411111",
     "amount_minor": 2999,
     "currency": "USD",
     "attempt_number": 1,
     "event_timestamp": "2026-06-16T12:00:00Z"
   }]
 }' | python -m json.tool
```

Request a next-generation retry decision

```
curl --fail-with-body -sS -X POST \
 "$ZAHLEN_BASE_URL/v1/_next/retry-decision" \
 -H 'Content-Type: application/json' \
 -H "X-API-Key: $ZAHLEN_API_KEY" \
 -H 'Idempotency-Key: order-8842-attempt-2' \
 -d '{
   "attempt_number": 2,
   "decline_code": "51",
   "issuer_bin": "411111",
   "amount_minor": 2999,
   "currency": "USD"
 }' | python -m json.tool
```

13.3 More curl workflow examples

Retrieve an event and its decision

```
EVENT_ID="evt_20260616_0001"

curl --fail-with-body -sS \
 -H "X-API-Key: $ZAHLEN_API_KEY" \
 "$ZAHLEN_BASE_URL/v1/payment-events/$EVENT_ID" | python -m json.tool

curl --fail-with-body -sS \
 -H "X-API-Key: $ZAHLEN_API_KEY" \
 "$ZAHLEN_BASE_URL/v1/payment-events/$EVENT_ID/decision" | python -m json.tool
```

Report a retry outcome

```
curl --fail-with-body -sS -X POST \
 "$ZAHLEN_BASE_URL/v1/retry-outcome" \
 -H 'Content-Type: application/json' \
 -H "X-API-Key: $ZAHLEN_API_KEY" \
 -H 'Idempotency-Key: outcome-order-8842-attempt-2' \
 -d '{
   "attempt_number": 2,
   "outcome": "APPROVED",
   "request_id": "req_example",
   "decision_id": "dec_example",
   "token": "tok_customer_001",
   "approval_code": "A12345",
   "settled_amount_minor": 2999,
   "currency": "USD",
   "outcome_timestamp": "2026-06-17T12:05:00Z"
 }' | python -m json.tool
```

Shell history warning

Inline secrets and sensitive payloads may remain in shell history. Prefer environment variables, protected files, or approved secret-injection tooling.

13.4 Python client foundation

```
from __future__ import annotations

import os
from typing import Any
import requests

BASE_URL = os.environ["ZAHLEN_BASE_URL"].rstrip("/")
API_KEY = os.environ["ZAHLEN_API_KEY"]

class ZahlenClient:
    def __init__(self, base_url: str = BASE_URL, api_key: str = API_KEY) -> None:
        self.base_url = base_url.rstrip("/")
        self.session = requests.Session()
        self.session.headers.update({
            "X-API-Key": api_key,
            "Content-Type": "application/json",
        })

    def request(self, method: str, path: str, *, json: dict[str, Any] | None = None,
                idempotency_key: str | None = None) -> dict[str, Any]:
        headers = {}
        if idempotency_key:
            headers["Idempotency-Key"] = idempotency_key
        response = self.session.request(
            method,
            f"{self.base_url}{path}",
            json=json,
            headers=headers,
            timeout=(5, 25),
        )
        response.raise_for_status()
        return response.json()
```

The connect/read timeout tuple prevents a process from waiting indefinitely. Production clients should also map HTTP errors into application-specific exceptions and capture safe request correlation metadata.

13.5 Python core workflow

```
client = ZahlenClient()

ingest = client.request("POST", "/v1/payment-events", json={
   "source": "billing_platform",
   "events": [{
      "event_id": "evt_20260616_0001",
      "decline_code": "51",
      "issuer_bin": "411111",
      "amount_minor": 2999,
      "currency": "USD",
      "attempt_number": 1,
   }],
})
print("upload_job_id:", ingest["upload_job_id"])

decision = client.request(
   "POST",
   "/v1/_next/retry-decision",
   json={
      "attempt_number": 2,
      "decline_code": "51",
      "issuer_bin": "411111",
      "amount_minor": 2999,
      "currency": "USD",
   },
   idempotency_key="order-8842-attempt-2",
)
print("decision:", decision.get("decision"))

outcome = client.request(
   "POST",
   "/v1/retry-outcome",
   json={
      "attempt_number": 2,
      "outcome": "APPROVED",
      "request_id": decision.get("request_id"),
      "decision_id": decision.get("decision_id"),
      "settled_amount_minor": 2999,
      "currency": "USD",
   },
   idempotency_key="outcome-order-8842-attempt-2",
)
print("outcome_id:", outcome.get("outcome_id"))
```

Preserve the identifier chain

Store event_id, batch_id or payment_event_batch_id, upload_job_id, request_id, decision_id, and outcome_id. These values connect merchant logs, Zahlen audit records, and investigation workflows.

13.6 Python error handling

```
import time
import random
import requests

def call_with_backoff(call, max_attempts: int = 5):
    for attempt in range(max_attempts):
        try:
            return call()
        except requests.HTTPError as exc:
            status = exc.response.status_code
            if status in {400, 401, 403, 404, 409, 422}:
                raise
            if status not in {429, 500, 503} or attempt == max_attempts - 1:
                raise
            retry_after = exc.response.headers.get("Retry-After")
            delay = float(retry_after) if retry_after else min(30, 2 ** attempt)
            time.sleep(delay * random.uniform(0.75, 1.25))
        except (requests.Timeout, requests.ConnectionError):
            if attempt == max_attempts - 1:
                raise
            time.sleep(min(30, 2 ** attempt) * random.uniform(0.75, 1.25))
```

Condition	Python behavior
400 / 422	Raise immediately; correct the payload.
401 / 403	Stop and repair credentials or authorization.
409	Reconcile the original idempotent operation.
429	Honor Retry-After when present and apply jitter.
500 / 503	Retry only with a bounded budget and stable idempotency.
Timeout / connection failure	Treat the result as uncertain and reconcile before repeating a write.

13.7 JavaScript client foundation

```
const BASE_URL = process.env.ZAHLEN_BASE_URL.replace(/\/$/, "");
const API_KEY = process.env.ZAHLEN_API_KEY;

async function zahlenRequest(path, options = {}) {
 const controller = new AbortController();
 const timer = setTimeout(() => controller.abort(), 25000);
 try {
  const response = await fetch(`${BASE_URL}${path}`, {
   ...options,
   signal: controller.signal,
   headers: {
    "Content-Type": "application/json",
    "X-API-Key": API_KEY,
    ...(options.headers || {})
   }
  });
  const body = await response.json().catch(() => ({}));
  if (!response.ok) {
   const error = new Error(`Zahlen HTTP ${response.status}`);
   error.status = response.status;
   error.body = body;
   throw error;
  }
  return body;
 } finally {
  clearTimeout(timer);
 }
}
```

Server-side only

Do not place a merchant API key in browser JavaScript. Use Node.js or another trusted server environment and expose only your own appropriately authorized application endpoints to browsers.

13.8 JavaScript core workflow

```
const ingest = await zahlenRequest("/v1/payment-events", {
  method: "POST",
  body: JSON.stringify({
    source: "billing_platform",
    events: [{
      event_id: "evt_20260616_0001",
      decline_code: "51",
      issuer_bin: "411111",
      amount_minor: 2999,
      currency: "USD",
      attempt_number: 1
    }]
  })
});
console.log(ingest.upload_job_id);

const decision = await zahlenRequest("/v1/_next/retry-decision", {
  method: "POST",
  headers: {"Idempotency-Key": "order-8842-attempt-2"},
  body: JSON.stringify({
    attempt_number: 2,
    decline_code: "51",
    issuer_bin: "411111",
    amount_minor: 2999,
    currency: "USD"
  })
});
console.log(decision);

const outcome = await zahlenRequest("/v1/retry-outcome", {
  method: "POST",
  headers: {"Idempotency-Key": "outcome-order-8842-attempt-2"},
  body: JSON.stringify({
    attempt_number: 2,
    outcome: "APPROVED",
    request_id: decision.request_id,
    decision_id: decision.decision_id,
    settled_amount_minor: 2999,
    currency: "USD"
  })
});
console.log(outcome.outcome_id);
```

13.9 Typed-client checklist

- Reject unknown request fields before transmission, matching Zahlen strict models.
- Represent nullable response fields explicitly rather than assuming every enrichment exists.
- Keep decimal amount fields separate from integer amount_minor and settled_amount_minor fields.
- Centralize authentication, timeouts, status mapping, retry budgets, and structured logging.
- Use unique event IDs and stable idempotency keys for one logical operation.
- Capture server request IDs and durable object identifiers in logs without exposing secrets.
- Test 401, 403, 404, 409, 422, 429, 500, 503, timeout, and uncertain-write scenarios.

13.10 Production readiness checklist

Check	Ready when
Secrets	Keys are injected at runtime and never committed or exposed to clients.
Timeouts	Every network call has connect and read limits.
Idempotency	Retry decision and outcome writes reuse stable keys.
Backoff	429 and transient 5xx responses use bounded exponential backoff with jitter.
Validation	Client models enforce required fields, bounds, and collection limits.
Observability	Logs include safe request and resource identifiers.
Schedule safety	No transport retry can initiate an off-schedule payment attempt.
Contract tests	Examples are tested against the target environment before release.

Chapter summary

curl is ideal for direct inspection, Python provides a concise service-client pattern, and server-side JavaScript fits event-driven integrations. In every language, production quality depends on secure key handling, strict serialization, durable identifiers, explicit timeouts, bounded retries, and respect for Zahlen's fixed Day 1, Day 2, Day 6, and Day 14 payment schedule.

ZAHLEN

API User Guide

Chapter 14 - Production Best Practices

Key rotation | Retry strategies | Monitoring

Audience
Merchants, developers, and integration engineers responsible for deploying, operating, and supporting a production Zahlen integration.

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Chapter 14 - Production Best Practices

Learning objectives

By the end of this chapter, you should be able to rotate API keys without downtime, implement safe network and application retries, monitor the complete Zahlen workflow, and define a practical production-readiness process.

A correct API integration is not automatically a production-ready integration. Production systems must assume credentials will be rotated, networks will fail, requests may be repeated, dependencies may slow down, and operators will need enough evidence to understand what happened.

Canonical payment schedule

Zahlen payment retries follow the fixed Day 1, Day 2, Day 6, and Day 14 schedule. HTTP retries, queue retries, webhook retries, and worker retries are technical recovery mechanisms; they must never create extra payment authorization attempts outside that schedule.

14.1 Production operating principles

- Fail closed when authentication or tenant context cannot be resolved.
- Use stable identifiers and idempotency keys for every repeatable write operation.
- Separate payment retry scheduling from API transport retry behavior.
- Collect enough telemetry to correlate merchant activity with Zahlen request, decision, batch, job, and outcome identifiers.
- Test key rotation, throttling, partial failures, and dependency outages before launch.

14.2 Key rotation

API keys are production credentials. Rotation should be planned as a normal operational process, not reserved only for emergencies. A safe rotation uses a controlled overlap period so the replacement key can be deployed and verified before the previous key is revoked.

Recommended rotation procedure

1. Create a replacement key for the correct tenant, merchant, environment, and service.
2. Store the new key in the approved secret manager. Do not place it in source control, a ticket, or a chat message.
3. Deploy the new key to one canary instance or a small percentage of traffic.
4. Verify authenticated health, payment-event, decision, and outcome requests with the new key.
5. Roll the new key out to all application instances, workers, and scheduled jobs.
6. Confirm that traffic using the old key has stopped.
7. Revoke the old key and monitor for rejected use attempts.
8. Record the rotation event, key identifier, operator, time, and validation evidence in the audit trail.

Do not rotate by overwrite alone
Replacing a secret everywhere at once without overlap can create an avoidable outage. Use dual-key overlap when the administrative policy permits it, then revoke the retired credential after verification.

14.3 Key storage and exposure prevention

Control	Production expectation
Secret storage	Use a managed secret store or protected runtime injection mechanism.
Application logs	Never log the complete key; log only a safe key ID or fingerprint.
Client location	Keep production keys in server-side systems, not browsers or mobile applications.
Environment separation	Use different keys for development, staging, production, and independent services.
Access scope	Grant secret access only to the workloads and operators that require it.
Revocation	Revoke immediately after confirmed compromise or unauthorized disclosure.
Audit	Retain creation, rotation, revocation, and failed-use evidence.

Compromise response

- Identify the affected key without redistributing the secret.
- Revoke or disable the key according to incident policy.
- Issue and deploy a replacement credential through the controlled rotation process.
- Review API activity by tenant, endpoint, time, source, and status code.
- Investigate unexpected outcome reporting, event ingestion, or decision traffic.

14.4 Retry strategy: classify before retrying

The client must determine whether a failure is safe to repeat. A retry decision should consider the HTTP method, status code, idempotency behavior, whether the server may have completed the operation, and the maximum retry budget.

Condition	Automatic retry?	Recommended behavior
GET with timeout or transient 5xx	Usually	Retry with bounded exponential backoff and jitter.
POST with stable Idempotency-Key	Carefully	Repeat the same logical request with the same key.
POST without idempotency guarantee	No automatic blind retry	Reconcile using stable identifiers before repeating.
400 or 422	No	Correct the payload; do not retry unchanged.
401 or 403	No	Repair authentication or authorization.
404	Usually no	Verify tenant-scoped identifier and endpoint.
409	Reconcile	Compare the original request and idempotency key.
429	Yes, later	Honor Retry-After when present and apply jitter.
500 or 503	Carefully	Retry within a bounded budget using idempotency.

Uncertain result

A timeout does not prove that the server did nothing. For a write request, treat the result as uncertain until you reconcile it using the idempotency key or durable business identifier.

14.5 Exponential backoff with jitter

Backoff reduces pressure on a recovering service. Jitter prevents many clients from retrying at the same instant. Always cap both the delay and the total number of attempts.

```
import random
import time

BASE_DELAY = 1.0
MAX_DELAY = 30.0
MAX_ATTEMPTS = 5

for retry_number in range(MAX_ATTEMPTS):
    delay = min(MAX_DELAY, BASE_DELAY * (2 ** retry_number))
    delay *= random.uniform(0.75, 1.25)
    time.sleep(delay)
```

Retry budget guidance

- Use a small maximum attempt count for synchronous user-facing requests.
- Use longer, durable retry queues for asynchronous processing when appropriate.
- Do not retry forever; move exhausted work to an alert or review queue.
- Preserve the same idempotency key for the same logical operation.
- Use a new idempotency key only for a genuinely new operation.

14.6 Technical retries versus payment retries

A production integration has several different retry layers. They must remain separate so a network recovery does not accidentally become another charge attempt.

Retry layer	Purpose	May create a new payment attempt?
HTTP request retry	Recover from network or server failure	No, not by itself
Queue delivery retry	Redeliver internal work	No, not by itself
Webhook delivery retry	Redeliver notification payload	No
Worker retry	Re-run a failed technical task	No, unless the task is explicitly authorized and idempotent
Zahlen payment retry schedule	Execute the business-authorized authorization attempt	Yes, only on Day 1, Day 2, Day 6, or Day 14

Critical safeguard

Store a durable payment-attempt record keyed by subscription or billing cycle and attempt number. Before sending an authorization to the processor, verify that the planned attempt matches the authorized Zahlen schedule and has not already been executed.

14.7 Monitoring the end-to-end workflow

Monitor the complete commercial workflow, not only API uptime. A healthy health endpoint does not prove that decisioning, outcomes, investigation runs, or webhooks are functioning correctly.

Signal	Why it matters	Example alert condition
Authentication failure rate	Detects revoked, expired, or misconfigured keys	Sustained increase in 401 responses
Authorization failure rate	Detects plan, role, or route policy problems	Unexpected increase in 403 responses
429 rate	Shows rate-limit or quota pressure	Repeated throttling above normal baseline
Request latency	Detects slow dependencies or policy evaluation	High p95 or p99 latency
5xx / 503 rate	Shows service or dependency failure	Error ratio exceeds agreed threshold
Decision completion	Confirms event-to-decision flow	Events remain without decisions beyond expected time
Outcome reporting lag	Shows learning-loop interruption	Decision executed but outcome not reported
Webhook failure rate	Detects unavailable callbacks or consumer errors	Delivery failures or retry backlog
Investigation-run backlog	Shows ingestion or bridge pressure	Runs remain non-terminal or unpopulated

14.8 Correlation and structured logging

Production logs should make it possible to trace one payment event across merchant systems and Zahlen without exposing credentials or prohibited payment data.

Identifier	Use in logs
event_id	Merchant-created correlation for one payment event
payment_event_batch_id / batch_id	Groups events submitted together
upload_job_id	Connects ingestion with background processing and investigation
request_id	Correlates an API request with support and audit records
decision_id	Identifies the retry recommendation
outcome_id	Identifies the reported execution result
subscription_id / billing_cycle_id	Connects activity to the merchant billing workflow
safe key ID or fingerprint	Identifies credential use without exposing the secret

```
{
 "event": "zahlen_api_request",
 "endpoint": "/v1/_next/retry-decision",
 "method": "POST",
 "status_code": 200,
 "latency_ms": 142,
 "event_id": "evt_20260616_0001",
 "request_id": "req_example",
 "decision_id": "dec_example",
 "idempotent_replay": false
}
```

Sensitive-data rule
Do not log API keys, full card numbers, CVV values, passwords, raw bank credentials, or unrestricted request bodies. Prefer allow-listed structured fields and merchant-side tokens.

14.9 Health checks and synthetic tests

Use layered checks. A connectivity probe verifies that the service responds; an authenticated synthetic test verifies that the commercial integration works. Synthetic tests must use approved test tenants, test credentials, and non-production payment evidence.

Check	What it proves
GET /v1/health	Service is reachable and returning health metadata
GET /v1/version	Deployed API and application version can be identified
Authenticated read	Key resolution and tenant-scoped authorization work
Synthetic event ingestion	Schema validation and ingestion path work
Synthetic decision request	Decision contract and policy evaluation work
Synthetic outcome report	Learning-loop write path works
Webhook test delivery	Callback, verification, deduplication, and processing work

- Run smoke tests after deployment, configuration change, key rotation, and dependency maintenance.
- Use unique test identifiers so synthetic activity can be filtered from business reporting.
- Alert when a test fails repeatedly, but avoid aggressive probes that create rate-limit pressure.

14.10 Deployment and change management

9. Validate request models against the current /v1 contract or discovery metadata.
10. Run unit, integration, contract, and tenant-isolation tests.
11. Deploy to staging with staging-only keys and data stores.
12. Run synthetic event, decision, outcome, and webhook tests.
13. Deploy gradually to production and observe error rate, latency, and throughput.
14. Verify the fixed Day 1, Day 2, Day 6, and Day 14 payment schedule in production configuration.
15. Record the release, configuration version, and verification evidence.

Rollback readiness

- Keep the previous application version and configuration available for rollback.
- Do not roll back databases or durable events without a tested migration strategy.
- Preserve idempotency and identifiers across deployment boundaries.
- After rollback, reconcile uncertain writes and verify outcome-reporting continuity.

14.11 Production-readiness checklist

Area	Ready when
Credentials	Keys are secret-managed, environment-specific, rotatable, and auditable.
Tenant safety	No client-supplied tenant ID is trusted as the ownership boundary.
Idempotency	Repeatable POST operations use stable logical keys and reconciliation.
Retry policy	400/401/403/404/422 are not blindly retried; 429/5xx use bounded backoff.
Payment schedule	Processor attempts are restricted to Day 1, Day 2, Day 6, and Day 14.
Timeouts	Every external request has connect and read timeouts.
Monitoring	Authentication, 429, 5xx, latency, decisions, outcomes, webhooks, and runs are observed.
Logging	Correlation IDs are captured without logging secrets or prohibited card data.
Testing	Key rotation, throttling, timeouts, duplicate delivery, and partial failure have been tested.
Operations	Runbooks, owners, escalation paths, and rollback procedures are documented.

Go-live rule
Do not launch solely because a happy-path request succeeded. Go live only after failure behavior, security controls, observability, and operator response have been demonstrated in a production-like environment.

14.12 Practical incident playbooks

Authentication failures spike

16. Check recent deployments and secret-manager changes.
17. Identify affected key IDs and environments without exposing secret material.
18. Compare 401 failures by endpoint, tenant, and application instance.
19. Rotate or replace misconfigured or compromised keys.
20. Review audit activity and confirm recovery after deployment.

429 responses increase

21. Confirm whether the limit is short-window rate limiting or longer-window quota exhaustion.
22. Check for loops, duplicate jobs, or unexpected traffic growth.
23. Verify that clients honor Retry-After and use jitter.
24. Reduce concurrency or batch safely where appropriate.
25. Review plan and quota settings only after abnormal traffic has been ruled out.

Outcome reporting falls behind

26. Compare completed processor attempts with reported outcome IDs.
27. Check queue depth, worker health, and failed outcome requests.
28. Replay only with stable identifiers and idempotency.
29. Reconcile missing records without creating another payment attempt.
30. Confirm the learning loop has resumed and backlog is decreasing.

14.13 Chapter summary

- Rotate keys with overlap, verification, revocation, and audit evidence.
- Classify failures before retrying and use stable idempotency for uncertain writes.
- Use bounded exponential backoff with jitter for 429 and transient server failures.
- Keep technical retries separate from the fixed Day 1, Day 2, Day 6, and Day 14 payment schedule.
- Monitor authentication, throttling, latency, decisions, outcomes, webhooks, and investigation runs.
- Log durable correlation identifiers while minimizing sensitive data.
- Treat failure testing, runbooks, and rollback readiness as launch requirements.

Next step
Use Appendix A for the OpenAPI specification, Appendix B for complete JSON examples, and Appendix C for troubleshooting guidance.

ZAHLEN

API User Guide

Appendix A - OpenAPI Specification

Discovery document | Endpoint catalog | Client-generation guidance

Audience
Merchants, developers, and integration engineers who need machine-readable API discovery, contract validation, testing, or client-generation guidance.

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Appendix A - OpenAPI Specification

Purpose

This appendix explains how to retrieve, read, validate, and safely use the Zahlen OpenAPI discovery document exposed at GET /v1/openapi.json.

OpenAPI is a machine-readable description format for HTTP APIs. It helps developers discover routes, authentication requirements, supported methods, and response categories. Tools can use an OpenAPI document to render interactive documentation, generate starter clients, create test cases, and compare API contracts between releases.

Important scope note

The deployed Zahlen /v1/openapi.json document is a static discovery document. It identifies selected routes, methods, basic response categories, and the X-API-Key security scheme. It is not the complete source of every request and response field. The detailed schema export and this guide remain necessary for full contract implementation.

A.1 Discovery endpoint

```
curl -sS https://api.example.com/v1/openapi.json | python -m json.tool
```

The discovery endpoint itself does not require the ApiKeyAuth security requirement in the static document. Other listed routes are marked with the ApiKeyAuth scheme. Environment policy may still place network or gateway controls in front of discovery.

A.2 Top-level document structure

Property	Type	Meaning
openapi	string	OpenAPI specification version used by the document. Zahlen returns 3.0.3.
info	object	Human-readable API title, version, and description.
paths	object	Route templates and supported HTTP methods.
components	object	Reusable definitions, including the API-key security scheme.

```
{
 "openapi": "3.0.3",
 "info": {
  "title": "Zahlen Payment Events API",
  "version": "v1",
  "description": "Payment Events, Batch Processing, Webhooks, Observability, and Governance APIs."
 },
 "paths": { "...": {} },
 "components": { "securitySchemes": { "...": {} } }
}
```

A.3 Security scheme

```
"ApiKeyAuth": {
 "type": "apiKey",
 "in": "header",
 "name": "X-API-Key"
}
```

For authenticated routes, send the merchant API key in the X-API-Key request header. Do not put the key in the URL, query string, JSON body, browser code, or logs.

A.4 Commercial route catalog

Method	Path	Purpose
POST	/v1/payment-events	Submit one or more payment events.
POST	/v1/payment-events/batch	Submit a batch and receive batch-oriented response data.
GET	/v1/payment-events/{event_id}	Retrieve one stored payment event.
GET	/v1/payment-events/{event_id}/decision	Retrieve the decision associated with an event.
GET	/v1/payment-events/{event_id}/ processor-result	Retrieve downstream processor execution evidence.
GET	/v1/payment-events/batches/{batch_id}	Retrieve batch detail and paginated event identifiers.
GET	/v1/payment-events/batches/ {batch_id}/summary	Retrieve batch-level summary and distributions.
GET	/v1/payment-events/batches/ {batch_id}/decisions	Retrieve paginated decisions for a batch.
GET	/v1/payment-events/batches/ {batch_id}/processor-results	Retrieve paginated processor results for a batch.
POST	/v1/webhook-subscriptions	Create a webhook subscription.
GET	/v1/webhook-subscriptions	List visible webhook subscriptions.
DELETE	/v1/webhook-subscriptions/ {subscription_id}	Delete or deactivate a subscription.

Additional commercial endpoints

The actual deployed API also contains retry-decision, retry-outcome, health, and version contracts documented elsewhere in this guide. Do not assume that the static discovery document lists every deployed merchant-facing route.

A.5 Governance and observability routes in discovery

Method	Path	Category
POST / GET	/v1/admin/api-keys	API key lifecycle
DELETE	/v1/admin/api-keys/{key_id}	API key revocation
GET	/v1/admin/api/metrics	API metrics
GET	/v1/admin/api/activity	API activity
GET	/v1/admin/api/webhook-operations	Webhook operations
GET	/v1/admin/api/health	Administrative API health
GET	/v1/admin/api/tenant-usage	Tenant usage reporting
GET	/v1/admin/api/audit-trail	API audit records
GET	/v1/admin/api/documentation	Structured documentation and examples
GET	/v1/admin/developer-portal	Developer portal data
GET	/v1/openapi.json	OpenAPI discovery

Authorization boundary

Routes under /v1/admin/* are administrative resources. A merchant X-API-Key must not be assumed to grant access. Use only the authorization method and enterprise contract approved for the deployment.

A.6 Path objects and operations

Each path contains one or more HTTP operations. The static document provides a generated summary, common response descriptions, and security metadata.

```
"/v1/payment-events": {
 "post": {
  "summary": "POST payment events",
  "responses": {
   "200": {"description": "Successful response"},
   "401": {"description": "Missing or invalid API key"},
   "429": {"description": "Plan quota exceeded"}
  },
  "security": [{"ApiKeyAuth": []}]
 }
}
```

Element	How to use it
summary	Display label only; do not build application logic from its wording.
responses.200	Indicates a successful response category, not the full payload schema.
responses.401	Signals missing or invalid authentication.
responses.429	Signals rate-limit or quota enforcement.
security	Shows whether ApiKeyAuth applies to the operation.

Actual routes may also return validation, authorization, not-found, conflict, or server-error responses. Chapter 11 defines the broader error-handling model.

A.7 Downloading and storing the specification

```
curl -sS https://api.example.com/v1/openapi.json   -o zahlen-openapi-v1.json

python -m json.tool zahlen-openapi-v1.json > /dev/null
```

- Store the downloaded document as a build or test artifact, not as a secret.
- Record the environment, download time, and deployed application version.
- Validate that the openapi property is 3.0.3 and the info.version value is expected.
- Do not silently replace a pinned contract during a production build.
- Review differences before adopting a newly downloaded document.

A.8 Inspecting with jq

```
# List all paths
jq -r '.paths | keys[]' zahlen-openapi-v1.json

# List method and path pairs
jq -r '.paths | to_entries[] as $p |
  $p.value | keys[] | "\(. | ascii_upcase) \($p.key)"' zahlen-openapi-v1.json

# Inspect the API-key security scheme
jq '.components.securitySchemes.ApiKeyAuth' zahlen-openapi-v1.json
```

A.9 Contract validation in Python

```
import json
from pathlib import Path

spec = json.loads(Path("zahlen-openapi-v1.json").read_text())

assert spec["openapi"] == "3.0.3"
assert spec["info"]["version"] == "v1"
assert "/v1/payment-events" in spec["paths"]
assert "post" in spec["paths"]["/v1/payment-events"]

scheme = spec["components"]["securitySchemes"]["ApiKeyAuth"]
assert scheme == {
    "type": "apiKey",
    "in": "header",
    "name": "X-API-Key",
}

print("Zahlen discovery contract validated")
```

Contract checks should fail clearly when a required route, method, version, or security definition changes. Keep the checks small and focused so a failure tells the developer what changed.

A.10 Client generation guidance

Many tools can generate API clients from OpenAPI. For Zahlen, use generated output as a starting point rather than unquestioned production code because the static discovery document does not contain the complete detailed schemas for every operation.

Generated artifact	Recommended treatment
HTTP method and path constants	Generally safe after review.
Authentication middleware	Confirm it sends X-API-Key only from secure server-side configuration.
Request models	Generate from the detailed schema export, not only the static discovery document.
Response models	Compare with documented Pydantic-derived schemas and nullable fields.
Retry behavior	Implement manually; generators should not decide payment or transport retry policy.
Error mapping	Extend to cover 400, 403, 404, 409, 422, 429, 500, and 503.
Timeouts and logging	Add explicitly; do not rely on generator defaults.

Payment schedule safeguard
Generated clients must not schedule authorization attempts. Zahlen payment retries remain fixed at Day 1, Day 2, Day 6, and Day 14. OpenAPI tooling describes HTTP operations; it does not replace the business retry policy.

A.11 Comparing specifications between releases

1. Download the specification from the current environment and the candidate environment.
2. Normalize JSON formatting so differences are readable.
3. Compare paths, methods, security requirements, versions, and response categories.
4. Review removed or renamed operations as potentially breaking changes.
5. Run request and response contract tests against the candidate deployment.
6. Approve the new contract before updating generated code or production artifacts.

```
python -m json.tool openapi-current.json > current.pretty.json
python -m json.tool openapi-candidate.json > candidate.pretty.json
diff -u current.pretty.json candidate.pretty.json
```

Change	Risk
New optional route	Usually additive.
New method on an existing path	Usually additive, but review authorization.
Removed path or method	Breaking for clients that use it.
Changed security requirement	High risk; review before deployment.
Changed version or response category	Requires contract and error-handling review.
Detailed schema change not visible in static discovery	Must be detected using schema export and contract tests.

A.12 OpenAPI versus the detailed schema reference

Source	Best use
/v1/openapi.json	Route discovery, method discovery, security-scheme discovery, basic tooling.
Detailed JSON schema export	Exact fields, required properties, nullability, constraints, and nested models.
This API User Guide	Developer workflows, examples, safety rules, retry behavior, and interpretation.
Runtime responses	Final evidence of deployed behavior; capture identifiers and errors for diagnostics.

Source-of-truth rule
For payload construction, use the detailed request and response schema definitions. For discovery, use /v1/openapi.json. When sources appear inconsistent, stop integration work and confirm the deployed contract with the Zahlen onboarding or platform team.

A.13 Appendix checklist

- The discovery document downloads successfully from the correct environment.
- The OpenAPI version is 3.0.3 and the API version is v1.
- The required route and method pairs are present.
- ApiKeyAuth uses the X-API-Key header.
- Administrative routes are treated as separately governed.
- Detailed request and response schemas are validated outside the static discovery document.
- Generated clients include explicit timeouts, error mapping, logging, and idempotency.
- Contract differences are reviewed before release.
- No generated behavior can create payment attempts outside Day 1, Day 2, Day 6, and Day 14.

ZAHLEN

Appendix B - JSON Examples

Zahlen API User Guide

For merchants, developers, and integration engineers

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Appendix B - JSON Examples

Purpose

This appendix provides copy-ready JSON examples for the confirmed Zahlen commercial API contracts. Replace fictional identifiers, timestamps, and URLs before testing. Field names and limits follow the 0616A schema export.

Examples are intentionally concise. Optional fields are included only when they help explain correlation, payment evidence, or the fixed retry schedule of Day 1, Day 2, Day 6, and Day 14.

B.1 Conventions

- All examples use fictional identifiers and tokenized payment references.
- Merchant-facing requests authenticate with the X-API-Key header; the key is not shown inside JSON bodies.
- Unknown top-level properties are rejected by strict request models where documented.
- Monetary values ending in _minor are integer minor units, such as cents.
- Timestamps are ISO 8601 strings in UTC.
- Response examples illustrate the documented shape; actual identifiers, timestamps, statuses, and optional fields vary.

Security reminder

Never place full card numbers, CVV values, passwords, API keys, or raw bank credentials inside metadata or example payloads.

B.2 Payment Event Submission

B.2.1 Single-event request

POST /v1/payment-events

```
{
  "source": "billing_platform",
  "events": [
    {
      "event_id": "evt_20260616_0001",
      "decline_code": "51",
      "issuer_bin": "411111",
      "issuer": "Example Bank",
      "country": "US",
      "card_brand": "VISA",
      "amount_minor": 2999,
      "currency": "USD",
      "attempt_number": 1,
      "attempt_day_in_cycle": 1,
      "event_timestamp": "2026-06-16T12:00:00Z",
      "payment_token": "tok_customer_001",
      "customer_id": "cus_001",
      "subscription_id": "sub_001",
      "processor": "example_processor",
      "metadata": {
        "invoice_id": "inv_1042",
        "channel": "subscription_renewal"
      }
    }
  ]
}
```

B.2.2 Example ingestion response

```
{
  "status": "COMPLETED",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "payment_event_batch_id": "peb_01JABC123",
  "upload_job_id": "ZN-2026-06-16-0001",
  "source": "billing_platform",
  "received_event_count": 1,
  "total_rows": 1,
  "valid_rows": 1,
  "invalid_rows": 0,
  "error_count": 0,
  "created_at": "2026-06-16T12:00:01Z",
  "completed_at": "2026-06-16T12:00:02Z",
  "ingestion": {
    "mode": "api",
    "normalized": 1
  }
}
```

Response values are illustrative

The schema defines the response fields and types, but exact status vocabulary and ingestion metadata are runtime values. Client code should not invent undocumented status transitions.

B.3 Batch Payment Event Submission

Both single and batch ingestion use PaymentEventsIngestRequest. The events array accepts 1 through 10,000 items.

POST /v1/payment-events/batch

```
{
  "source": "nightly_renewal_export",
  "events": [
    {
      "event_id": "evt_0001",
      "decline_code": "51",
      "issuer_bin": "411111",
      "amount_minor": 2999,
      "currency": "USD",
      "attempt_number": 1,
      "attempt_day_in_cycle": 1
    },
    {
      "event_id": "evt_0002",
      "decline_code": "91",
      "issuer_bin": "550000",
      "amount_minor": 1499,
      "currency": "USD",
      "attempt_number": 2,
      "attempt_day_in_cycle": 2
    },
    {
      "event_id": "evt_0003",
      "decline_code": "05",
      "issuer_bin": "340000",
      "amount_minor": 4999,
      "currency": "USD",
      "attempt_number": 3,
      "attempt_day_in_cycle": 6
    }
  ]
}
```

Example batch response

```
{
  "status": "ACCEPTED",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "submitted": 3,
  "accepted": 3,
  "rejected": 0,
  "batch_id": "peb_01JABC456",
  "events_url": "/v1/payment-events/batches/peb_01JABC456",
  "upload_job_id": "ZN-2026-06-16-0002",
  "source": "nightly_renewal_export",
  "created_at": "2026-06-16T12:05:00Z",
  "completed_at": null,
  "ingestion": {
    "queued": true
  }
}
```

B.3.1 Batch-detail response

GET /v1/payment-events/batches/{batch_id}?limit=100&offset=0

```
{
  "batch_id": "peb_01JABC456",
  "status": "COMPLETED",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "submitted": 3,
  "accepted": 3,
  "rejected": 0,
  "created_at": "2026-06-16T12:05:00Z",
  "completed_at": "2026-06-16T12:05:06Z",
  "upload_job_id": "ZN-2026-06-16-0002",
  "source": "nightly_renewal_export",
  "event_ids": [
    "evt_0001",
    "evt_0002",
    "evt_0003"
  ],
  "total_events": 3,
  "returned": 3,
  "offset": 0,
  "limit": 100,
  "has_more": false,
  "events_url": "/v1/payment-events/batches/peb_01JABC456",
  "ingestion": {
    "normalized": 3
  }
}
```

B.4 Batch Summary and Decisions

GET /v1/payment-events/batches/{batch_id}/summary


```
{
  "batch_id": "peb_01JABC456",
  "status": "COMPLETED",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "submitted": 3,
  "accepted": 3,
  "rejected": 0,
  "created_at": "2026-06-16T12:05:00Z",
  "completed_at": "2026-06-16T12:05:06Z",
  "upload_job_id": "ZN-2026-06-16-0002",
  "source": "nightly_renewal_export",
  "event_count": 3,
  "retry_candidates": 2,
  "top_processors": [
    {
      "name": "example_processor",
      "count": 3
    }
  ],
  "top_decline_codes": [
    {
      "code": "51",
      "count": 1
    },
    {
      "code": "91",
      "count": 1
    },
    {
      "code": "05",
      "count": 1
    }
  ],
  "top_issuers": [
    {
      "issuer": "Example Bank",
      "count": 2
    }
  ],
  "top_issuer_bins": [
    {
      "issuer_bin": "411111",
      "count": 1
    }
  ],
  "top_card_brands": [
    {
      "card_brand": "VISA",
      "count": 2
    }
  ],
  "confidence_distribution": {
    "HIGH": 1,
    "MEDIUM": 1,
    "LOW": 1
  },
  "decision_distribution": {
    "RETRY": 2,
    "DO_NOT_RETRY": 1
  },
```

```
  "events_url": "/v1/payment-events/batches/peb_01JABC456",
  "summary_source": "decision_pipeline",
  "ingestion": {
    "normalized": 3
  }
}
```


GET /v1/payment-events/batches/{batch_id}/decisions?limit=100&offset=0


```
{
  "batch_id": "peb_01JABC456",
  "status": "COMPLETED",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "submitted": 3,
  "accepted": 3,
  "rejected": 0,
  "created_at": "2026-06-16T12:05:00Z",
  "completed_at": "2026-06-16T12:05:06Z",
  "upload_job_id": "ZN-2026-06-16-0002",
  "source": "nightly_renewal_export",
  "event_count": 3,
  "total_events": 3,
  "returned": 1,
  "offset": 0,
  "limit": 100,
  "has_more": false,
  "decisions": [
    {
      "event_id": "evt_0001",
      "merchant_id": "merchant_example",
      "tenant_id": "tenant_example",
      "payment_event_batch_id": "peb_01JABC456",
      "upload_job_id": "ZN-2026-06-16-0002",
      "decision_id": "dec_01JABC001",
      "request_id": "req_01JABC001",
      "decision": "RETRY",
      "recommended_retry_day": 2,
      "confidence": "HIGH",
      "confidence_score": 0.92,
      "reason_codes": [
        "INSUFFICIENT_FUNDS_PATTERN"
      ],
      "reason_detail": "Retry on the next approved schedule day.",
      "policy_source": "zahlen_fixed_schedule",
      "matched_policy_id": "policy_default",
      "idempotent_replay": false,
      "created_at": "2026-06-16T12:05:04Z",
      "source": "payment_events",
      "event": {
        "event_id": "evt_0001"
      },
      "issuer_context": {
        "issuer_bin": "411111"
      }
    }
  ],
  "events_url": "/v1/payment-events/batches/peb_01JABC456",
  "decisions_source": "decision_pipeline",
  "ingestion": {
    "normalized": 3
  }
}
```

B.5 Event, Decision, and Processor Result Retrieval

GET /v1/payment-events/{event_id}


```
{
  "event_id": "evt_0001",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "payment_event_batch_id": "peb_01JABC456",
  "upload_job_id": "ZN-2026-06-16-0002",
  "source": "nightly_renewal_export",
  "source_row_number": 1,
  "payment_token": "tok_customer_001",
  "decline_code": "51",
  "response_code": "DECLINED",
  "issuer_bin": "411111",
  "issuer": "Example Bank",
  "country": "US",
  "card_brand": "VISA",
  "amount_minor": 2999,
  "currency": "USD",
  "attempt_number": 1,
  "event_timestamp": "2026-06-16T12:00:00Z",
  "processor": "example_processor",
  "authorization_id": "auth_001",
  "normalized_event": {
    "decline_category": "INSUFFICIENT_FUNDS"
  },
  "raw_event": {
    "decline_code": "51",
    "processor": "example_processor"
  },
  "created_at": "2026-06-16T12:05:01Z",
  "updated_at": "2026-06-16T12:05:02Z"
}
```


GET /v1/payment-events/{event_id}/decision


```
{
  "event_id": "evt_0001",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "payment_event_batch_id": "peb_01JABC456",
  "upload_job_id": "ZN-2026-06-16-0002",
  "decision_id": "dec_01JABC001",
  "request_id": "req_01JABC001",
  "decision": "RETRY",
  "recommended_retry_day": 2,
  "confidence": "HIGH",
  "confidence_score": 0.92,
  "reason_codes": [
    "INSUFFICIENT_FUNDS_PATTERN"
  ],
  "reason_detail": "Retry on Day 2, the next approved fixed-schedule day.",
  "policy_source": "zahlen_fixed_schedule",
  "matched_policy_id": "policy_default",
  "idempotent_replay": false,
  "created_at": "2026-06-16T12:05:04Z",
  "source": "payment_events",
  "event": {
    "event_id": "evt_0001",
    "attempt_number": 1
  },
  "issuer_context": {
    "issuer_bin": "411111",
    "issuer": "Example Bank"
```

```
  }
}
```

GET /v1/payment-events/{event_id}/processor-result

```
{
  "event_id": "evt_0001",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "payment_event_batch_id": "peb_01JABC456",
  "upload_job_id": "ZN-2026-06-16-0002",
  "processor": "example_processor",
  "processor_reference": "proc_ref_001",
  "result": "DECLINED",
  "response_code": "51",
  "response_description": "Insufficient funds",
  "processed_at": "2026-06-16T12:00:00Z",
  "source": "processor_callback",
  "event": {
    "event_id": "evt_0001"
  }
}
```

B.6 Legacy Retry Decision

POST /v1/retry-decision

```
{
  "payment_token": "tok_001",
  "billing_cycle_id": "cycle_2026_06",
  "event_ts_iso": "2026-06-16T12:00:00Z",
  "cycle_start_ts_iso": "2026-06-01T00:00:00Z",
  "country_code": "US",
  "card_network": "VISA",
  "paymentech_code": "51",
  "attempt_day_in_cycle": 1,
  "days_until_suspension": 20,
  "bank": "Example Bank",
  "issuer_id": "issuer_001",
  "bin": "411111",
  "amount": 29.99,
  "currency": "USD",
  "processor": "paymentech",
  "transaction_kind": "RECURRING",
  "subscription_id": "sub_001",
  "merchant_reference_id": "inv_1042",
  "ai_mode": false,
  "enable_spike_alerts": true,
  "external_change_mode": "DETERMINISTIC"
}
```

The legacy response contains nested decision, reasoning, signals, explanations, and meta blocks. Because those nested blocks are deployment-defined structures, clients should parse documented keys while preserving forward-compatible handling for optional content.

Illustrative legacy response shape

```
{
  "decision": {
    "action": "RETRY",
    "recommended_retry_day": 2
  },
  "reasoning": {
    "summary": "Evidence supports the next fixed-schedule attempt."
  },
  "signals": {
    "decline_code": "51",
    "attempt_day_in_cycle": 1
  },
  "explanations": [
    "The next approved retry day is Day 2."
  ],
  "meta": {
    "request_id": "req_legacy_001",
    "api_version": "v1"
  },
  "policy": {
    "source": "zahlen_fixed_schedule"
  }
}
```

Do not combine contracts

The legacy RetryDecisionRequest and the next-generation NextRetryDecisionPayload are separate contracts. Do not mix fields from both models in one request.

B.7 Legacy Batch Retry Decision

BatchRetryDecisionRequest accepts an events array with no more than 500 legacy decision requests.

POST /v1/retry-decision/batch

```
{
  "events": [
    {
      "payment_token": "tok_001",
      "billing_cycle_id": "cycle_2026_06_a",
      "event_ts_iso": "2026-06-16T12:00:00Z",
      "cycle_start_ts_iso": "2026-06-01T00:00:00Z",
      "paymentech_code": "51",
      "attempt_day_in_cycle": 1
    },
    {
      "payment_token": "tok_002",
      "billing_cycle_id": "cycle_2026_06_b",
      "event_ts_iso": "2026-06-16T12:01:00Z",
      "cycle_start_ts_iso": "2026-06-01T00:00:00Z",
      "paymentech_code": "91",
      "attempt_day_in_cycle": 2
    }
  ]
}
```

Illustrative batch response shape

```
{
  "results": [
    {
      "decision": {
        "action": "RETRY",
        "recommended_retry_day": 2
      },
      "reasoning": {
        "summary": "Proceed to Day 2."
      },
      "signals": {},
      "explanations": [],
      "meta": {
        "request_id": "req_001"
      }
    },
    {
      "decision": {
        "action": "RETRY",
        "recommended_retry_day": 6
      },
      "reasoning": {
        "summary": "Proceed to Day 6."
      },
      "signals": {},
      "explanations": [],
      "meta": {
        "request_id": "req_002"
      }
    }
  ],
  "meta": {
    "batch_request_id": "batch_req_001"
  },
  "count": 2
}
```

B.8 Next-Generation Retry Decision

POST /v1/_next/retry-decision

```
{
  "merchant_id": "merchant_example",
  "token": "tok_001",
  "attempt_number": 2,
  "decline_code": "51",
  "issuer_bin": "411111",
  "issuer_name": "Example Bank",
  "card_brand": "VISA",
  "issuer_country": "US",
  "amount_minor": 2999,
  "currency": "USD",
  "decline_category": "INSUFFICIENT_FUNDS",
  "subscription_id": "sub_001",
  "invoice_id": "inv_1042",
  "order_id": "ord_1042",
  "processor": "example_processor"
}
```

The only universally required request field in the next-generation contract is attempt_number, which must be at least 1. Include optional evidence when it is trustworthy and available.

Example response using confirmed top-level fields

```
{
  "request_id": "req_next_001",
  "decision_id": "dec_next_001",
  "merchant_id": "merchant_example",
  "token": "tok_001",
  "attempt_number": 2,
  "decision": "RETRY",
  "retry_day": 6,
  "reason_code": "FIXED_SCHEDULE_NEXT_DAY",
  "reason_detail": "Attempt 2 advances to Day 6.",
  "policy_source": "zahlen_fixed_schedule",
  "matched_policy_id": "policy_default",
  "confidence": "HIGH",
  "created_at": "2026-06-16T12:10:00Z",
  "idempotent_replay": false,
  "issuer_context": {
    "issuer_bin": "411111"
  },
  "explainability_sections": [],
  "decision_trace": {}
}
```

Schedule interpretation

A technical resend of the same API call does not advance the payment schedule. Payment attempts remain limited to Day 1, Day 2, Day 6, and Day 14.

B.9 Retry Outcome Reporting

POST /v1/retry-outcome - recovered example

```
{
  "merchant_id": "merchant_example",
  "attempt_number": 2,
  "outcome": "RECOVERED",
  "request_id": "req_next_001",
  "decision_id": "dec_next_001",
  "token": "tok_001",
  "approval_code": "APPROVED123",
  "final_decline_code": null,
  "settled_amount_minor": 2999,
  "currency": "USD",
  "outcome_timestamp": "2026-06-17T09:15:00Z"
}
```

POST /v1/retry-outcome - declined example

```
{
  "merchant_id": "merchant_example",
  "attempt_number": 3,
  "outcome": "DECLINED",
  "request_id": "req_next_002",
  "decision_id": "dec_next_002",
  "token": "tok_002",
  "approval_code": null,
  "final_decline_code": "51",
  "settled_amount_minor": null,
  "currency": "USD",
  "outcome_timestamp": "2026-06-21T09:15:00Z"
}
```

Example outcome response

```
{
  "outcome_id": "out_01JABC001",
  "merchant_id": "merchant_example",
  "request_id": "req_next_001",
  "decision_id": "dec_next_001",
  "token": "tok_001",
  "attempt_number": 2,
  "outcome": "RECOVERED",
  "approval_code": "APPROVED123",
  "final_decline_code": null,
  "settled_amount_minor": 2999,
  "currency": "USD",
  "outcome_timestamp": "2026-06-17T09:15:00Z",
  "created_at": "2026-06-17T09:15:01Z",
  "matched_by": "decision_id"
}
```

Report the observed processor or settlement result. A scheduled attempt is not a recovered payment until the authorization or settlement evidence confirms success.

B.10 Webhook Subscriptions

POST /v1/webhook-subscriptions

```
{
  "callback_url": "https://merchant.example.com/webhooks/zahlen",
  "events": [
    "retry.outcome.recorded",
    "payment_event.decision.ready"
  ]
}
```

Illustrative subscription response

```
{
  "subscription_id": "whsub_01JABC001",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "callback_url": "https://merchant.example.com/webhooks/zahlen",
  "events": [
    "retry.outcome.recorded",
    "payment_event.decision.ready"
  ],
  "status": "ACTIVE",
  "created_at": "2026-06-16T12:20:00Z",
  "updated_at": "2026-06-16T12:20:00Z"
}
```

Illustrative list response

```
{
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "count": 1,
  "subscriptions": [
    {
      "subscription_id": "whsub_01JABC001",
      "merchant_id": "merchant_example",
      "tenant_id": "tenant_example",
      "callback_url": "https://merchant.example.com/webhooks/zahlen",
      "events": [
        "retry.outcome.recorded"
      ],
      "status": "ACTIVE",
      "created_at": "2026-06-16T12:20:00Z",
      "updated_at": "2026-06-16T12:20:00Z",
      "deleted_at": null
    }
  ]
}
```

Illustrative delete response

```
{
  "subscription_id": "whsub_01JABC001",
  "merchant_id": "merchant_example",
  "tenant_id": "tenant_example",
  "deleted": true,
  "status": "DELETED",
  "deleted_at": "2026-06-16T12:30:00Z"
}
```

Event names and verification are deployment-specific

The confirmed schema defines subscription fields, not a universal event catalog, signing header, or signature algorithm. Use the active Zahlen webhook outcome contract for production.

B.11 Health, Version, and Errors

GET /v1/health

```
{
  "status": "ok",
  "service": "zahlen",
  "api_version": "v1",
  "time": "2026-06-16T12:30:00Z"
}
```

GET /v1/version

```
{
  "api_version": "v1",
  "app_version": "0616A",
  "rules_version": "rules-2026-06",
  "playbook_version": "playbook-2026-06"
}
```

B.11.1 Validation error

Illustrative HTTP 422 body

```
{
  "error": {
    "code": "VALIDATION_ERROR",
    "message": "Request validation failed",
    "details": [
      {
        "field": "events.0.event_id",
        "message": "Field required"
      }
    ]
  },
  "meta": {
    "request_id": "req_error_001",
    "status_code": 422
  }
}
```

B.11.2 Authentication error

Illustrative HTTP 401 body

```
{
  "error": {
    "code": "INVALID_API_KEY",
    "message": "Authentication failed"
  },
  "meta": {
    "request_id": "req_error_002",
    "status_code": 401
  }
}
```

B.11.3 Rate-limit response

Illustrative HTTP 429 body

```
{
  "error": {
    "code": "RATE_LIMITED",
    "message": "Request limit exceeded"
  },
```

```
  "meta": {
    "request_id": "req_error_003",
    "status_code": 429,
    "retry_after_seconds": 30
  }
}
```

Error-body details can vary by route and middleware. Always branch first on the HTTP status, then parse documented fields defensively.

B.12 Copy-and-Test Checklist

Step	Check	Why it matters
1	Replace the base URL	Use the correct development, staging, or production hostname.
2	Set X-API-Key outside JSON	Store the credential in a secret manager or environment variable.
3	Replace identifiers	Use unique event IDs, billing-cycle IDs, and idempotency keys.
4	Keep strict field names	Unknown top-level properties may produce HTTP 422.
5	Validate monetary units	Do not mix decimal amount with integer amount_minor.
6	Preserve correlation	Log request_id, decision_id, event_id, batch_id, and upload_job_id.
7	Test negative paths	Exercise 401, 403, 404, 409, 422, 429, and transient 5xx handling.
8	Protect the schedule	Never let API retries create payment attempts outside Day 1, 2, 6, and 14.

Publication note

These examples are teaching aids grounded in the confirmed schema. When an example includes runtime vocabulary or nested flexible objects, treat it as illustrative rather than a universal enumeration.

ZAHLEN

Appendix C - Troubleshooting

Zahlen API User Guide

For merchants, developers, and integration engineers

Version 1.0 | Source baseline: zahlen_deploy_0616A.tar.gz | June 2026

Commercial developer experience | Tenant-safe operations | Explainable retry intelligence

Appendix C - Troubleshooting

Purpose

This appendix provides a repeatable diagnostic method for common Zahlen API integration failures. Begin with evidence, preserve identifiers, and change only one variable at a time. Do not use troubleshooting as a reason to bypass authentication, tenant isolation, idempotency, quotas, or the fixed payment retry schedule.

The fastest path to resolution is usually not a larger retry loop. It is a precise answer to four questions: Which operation failed? What did the server return? Which durable identifiers were created? Did the failure occur before or after Zahlen accepted the logical operation?

C.1 Before You Troubleshoot

- Record the environment hostname and the exact route, method, and UTC timestamp.
- Capture the HTTP status, response body, response headers, and any request or correlation identifier.
- Identify the API key by safe key ID or fingerprint; never paste the secret into tickets or chat.
- Preserve merchant-side event IDs, idempotency keys, batch IDs, upload job IDs, decision IDs, and outcome IDs.
- Confirm whether the client retried and whether each retry reused the same logical identifiers.
- Redact payment tokens, customer identifiers, and sensitive metadata before sharing evidence.

Payment schedule boundary

HTTP retries, polling, webhook redelivery, and administrative remediation must never create extra payment authorizations. Zahlen payment attempts follow the fixed schedule: Day 1, Day 2, Day 6, and Day 14.

C.2 Five-Minute Triage

Step	Question	Evidence
1	Is the API reachable?	GET /v1/health, DNS, TLS, network path
2	Is authentication working?	X-API-Key presence, environment, key status
3	Is the request contract valid?	Content-Type, JSON syntax, required fields, types
4	Was the operation accepted?	HTTP status and returned durable identifiers
5	Is downstream processing complete?	Batch status, upload_job_id, decision/outcome resources

C.3 Quick Symptom Matrix

Symptom	Most likely causes	First action
No response / timeout	DNS, TLS, firewall, proxy, service unavailability	Test /v1/health with a short timeout; preserve whether the POST result is uncertain.
401 on every request	Missing, invalid, revoked, malformed, or wrong-environment key	Verify X-API-Key injection and key status without logging the secret.
403	Valid identity lacks permission, plan capability, role, or route access	Confirm endpoint authorization and public-versus-admin boundary.
404	Wrong ID, wrong tenant, wrong path, or resource not yet created	Verify identifier source, tenant context, and exact route.
409	Idempotency conflict or resource state conflict	Compare the original idempotency key and request body.
422	Schema validation failure	Correct required fields, types, constraints, and unknown properties.
429	Rate limit or quota enforcement	Honor Retry-After, back off, inspect traffic and quota usage.
500 / 503	Server or dependency failure	Retry only when safe, with bounded backoff and idempotency.
Decision has no retry day	Intentional WAIT/STOP/review outcome or insufficient evidence	Follow explicit decision and reason; do not invent a date.
Batch appears incomplete	Asynchronous processing or pagination	Check status, counts, has_more, offset, and upload_job_id.
Webhook duplicates	At-least-once delivery or provider retry	Deduplicate durably and make the consumer repeat-safe.
Investigation run not visible	Admin authorization or tenant context mismatch	Verify /v1/admin access and authenticated tenant context.

C.4 Evidence Packet for Support

- Environment and base URL (without embedded credentials).
- HTTP method and path.
- UTC timestamp and client timeout value.
- Safe API key ID/fingerprint.
- Status code, response headers, and redacted response body.
- event_id, batch_id, upload_job_id, request_id, decision_id, outcome_id, or subscription_id as applicable.
- Idempotency-Key and whether the body changed between attempts.
- Minimal redacted request that reproduces the problem.
- Expected behavior versus observed behavior.

C.5 Connectivity, DNS, and TLS

C.5.1 Health and version checks

```
curl -i --connect-timeout 5 --max-time 15 \
  https://api.example.com/v1/health

curl -i --connect-timeout 5 --max-time 15 \
  https://api.example.com/v1/version
```

A successful health response confirms basic reachability and service response. It does not prove that a merchant key is valid for authenticated routes.

C.5.2 Common connectivity failures

Failure	Diagnostic direction
Could not resolve host	Verify hostname spelling, DNS records, VPN, and local resolver.
Connection refused	Confirm port, reverse proxy, service state, and firewall policy.
Connection timed out	Check routing, security groups, proxy egress, and service capacity.
TLS certificate error	Confirm hostname matches the certificate and the trust store is current.
Unexpected redirect	Use the documented HTTPS base URL; do not send credentials across unsafe redirects.
HTML returned instead of JSON	Check route, proxy configuration, and whether an operator login page intercepted the request.

Uncertain POST result

A client timeout does not prove the server rejected the request. Before creating a new logical operation, query by the stable event or decision identifiers, or repeat only with the same idempotency key where supported.

C.6 Authentication and Authorization

C.6.1 Diagnosing HTTP 401

1. Confirm the header name is exactly X-API-Key.
2. Confirm the key is being read from the intended secret or environment variable.
3. Check for leading/trailing whitespace, line breaks, shell quoting, or accidental truncation.
4. Confirm the key belongs to the selected development, staging, or production environment.
5. Confirm the key is active and has not been revoked or rotated out.

```
curl -i -X POST https://api.example.com/v1/payment-events \
  -H 'Content-Type: application/json' \
  -H 'X-API-Key: zk_live_REPLACE_ME' \
  -d '{"events":[{"event_id":"evt_auth_test_001"}]}'
```

C.6.2 Diagnosing HTTP 403

HTTP 403 normally means the credential was recognized but the caller is not allowed to perform the operation. Review plan capability, route authorization, merchant/tenant assignment, and whether the endpoint is administrative.

Public versus administrative access

A merchant X-API-Key should not be assumed to authorize /v1/admin/* routes. Administrative investigation-run and governance APIs require separately approved administrative context.

C.6.3 Suspected key compromise

- Revoke the affected key immediately.
- Create a replacement through the approved lifecycle.
- Review audit/activity by key ID, tenant, route, time, and source network.
- Inspect quota spikes and unusual endpoint access.
- Preserve evidence and notify authorized contacts.

C.7 Validation and Request Construction

C.7.1 HTTP 400 versus 422

Status	Typical meaning	Client action
400	Malformed JSON, invalid business condition, or route-specific bad request	Correct the request; do not retry unchanged.
422	Schema validation failure	Read field-level errors and update serialization/types.

C.7.2 Strict models

Key Zahlen request models forbid unknown top-level properties. A misspelled field is a client bug, not an extension point. Place only approved custom data inside the documented metadata object where available.

```
{
  "events": [{
    "event_id": "evt_001",
    "amount_mnor": 2999
  }]
}

// Incorrect: amount_mnor is misspelled and should be amount_minor.
```

C.7.3 Frequent validation causes

- Missing required event_id for payment events.
- Empty events array or more than 10,000 payment events.
- More than 500 events in a legacy retry-decision batch.
- attempt_number below the documented minimum.
- Negative amount or latency values where a nonnegative constraint applies.
- String values sent for integer fields such as amount_minor.
- Unknown top-level properties.
- callback_url outside the supported length or an empty webhook events list.

C.7.4 Local validation practice

Use typed request models, JSON Schema validation, and contract tests before sending traffic. Test null, omitted optional fields, maximum batch size, and one unknown-field failure.

C.8 Resource Visibility, 404, and Tenant Isolation

C.8.1 Why a valid ID can still return 404

- The identifier belongs to another tenant and is intentionally not visible.
- The client used an event ID where a batch ID was required.
- The resource has not been created or downstream processing is incomplete.
- The path is incorrect, including singular/plural or nested resource differences.
- The caller used the wrong environment.

Do not bypass tenant filters

Cross-tenant empty or 404 results are expected security behavior. Never add tenant_id to a request body, query string, or form to force visibility. Tenant ownership is resolved from authenticated context.

C.8.2 Identifier chain

Identifier	Created by	Troubleshooting use
event_id	Merchant	Find one submitted event and avoid duplicate logical events.
batch_id / payment_event_batch_id	Zahlen	Retrieve batch state, summaries, decisions, and processor results.
upload_job_id	Zahlen	Correlate ingestion with asynchronous processing and investigation runs.
request_id	Zahlen	Trace one decision request across logs and support.
decision_id	Zahlen	Link recommendation to the observed retry outcome.
outcome_id	Zahlen	Confirm the recovery-learning record was accepted.
subscription_id	Zahlen	Manage and diagnose one webhook subscription.

C.9 Conflicts, Idempotency, and Duplicate Effects

C.9.1 Diagnosing HTTP 409

A conflict can indicate that an idempotency key was reused with a different request body, that the resource already exists in an incompatible state, or that the requested transition is not allowed.

6. Locate the original local operation record.
7. Compare the exact idempotency key and serialized body.
8. Confirm whether the first request returned a durable identifier.
9. Do not generate a new key merely to force a second effect.
10. Reconcile the server resource before retrying.

C.9.2 Idempotency rules

- One logical operation gets one stable key.
- The same logical operation reuses the same key and body.
- A materially different operation uses a new key.
- Store the key until the operation is terminal and reconciled.
- Never base the key only on the HTTP attempt number.

```
Idempotency-Key: order-8842-attempt-2
```

Payment attempt identity

The attempt number in an idempotency key identifies the scheduled payment attempt. It does not authorize additional attempts beyond Day 1, Day 2, Day 6, and Day 14.

C.10 Rate Limits, Quotas, and HTTP 429

C.10.1 Immediate client response

11. Stop immediate repeated retries.
12. Read Retry-After and rate/quota metadata when supplied.
13. Use bounded exponential backoff with randomized jitter.
14. Preserve the idempotency key for the same logical operation.
15. Alert when throttling is sustained or increases suddenly.

```
delay = min(max_delay, base_delay * (2 ** retry_number))
delay = delay * random.uniform(0.75, 1.25)
```

C.10.2 Diagnose before increasing capacity

Pattern	Possible cause	Action
Single key dominates	Integration loop or compromised credential	Stop the client, rotate/revoke if needed, inspect activity.
Large synchronized spike	Fleet restart or missing jitter	Spread retries and add concurrency control.
Steady legitimate growth	Plan or quota no longer fits volume	Coordinate an approved capacity change.
Many small requests	Inefficient integration	Batch where appropriate without exceeding schema limits.
Repeated same operation	Missing idempotency/reconciliation	Fix logical retry behavior before raising limits.

Payment-event ingestion accepts 1 to 10,000 events per request, while the legacy retry-decision batch accepts up to 500. These are request-schema ceilings, not guaranteed throughput targets.

C.11 Server Errors, Timeouts, and Safe Retries

C.11.1 HTTP 500 and 503

Treat 500 and 503 as potentially transient, but do not retry every POST blindly. A server may have completed the operation before a downstream failure or network interruption prevented the response from reaching the client.

Operation	Automatic retry guidance
GET resource	Usually safe with bounded backoff.
POST retry decision	Reuse the same idempotency key and body.
POST retry outcome	Reuse stable identifiers and idempotency where supported.
POST payment-event batch	Use stable event IDs and reconcile batch/job identifiers.
Validation failure	Do not retry until corrected.

C.11.2 Retry budget

- Set connect and total request timeouts.
- Cap retry count and total elapsed time.
- Use exponential backoff with jitter.
- Open a circuit or reduce concurrency during sustained dependency failure.
- Move irrecoverable work to a durable review queue instead of looping forever.

Never confuse transport and payment retries

Retrying an HTTP request is a technical recovery mechanism. Retrying a card authorization is a payment action governed by the Day 1, Day 2, Day 6, and Day 14 schedule.

C.12 Payment Events and Batch Processing

C.12.1 Event not found after submission

16. Check whether the POST succeeded and returned payment_event_batch_id or batch_id.
17. Verify event_id spelling and environment.
18. Retrieve the batch and inspect accepted/rejected counts.
19. Check upload_job_id and processing status.
20. Allow for asynchronous completion before declaring loss.

C.12.2 Batch counts do not match expectations

Field	Meaning
submitted	Number presented to the batch endpoint.
accepted	Number accepted for processing.
rejected	Number rejected at ingestion.
event_count / total_events	Known events associated with the batch.
returned	Number included in the current paginated response.
has_more	Whether another page remains.

- Use offset >= 0 and limit 1 through 1,000.
- Continue while has_more is true.
- Do not compare returned with total_events as if they were the same metric.
- Inspect invalid_rows, error_count, and ingestion details when available.

C.12.3 Decision or processor result missing

A payment event, its retry decision, and its processor result are separate resources and may become available at different times. Check batch status, decision source, processor-results source, and upload job processing before escalating.

C.13 Retry Decisions and Outcomes

C.13.1 Decision has no retry day

A nullable retry day can be intentional. Follow the explicit decision/action, reason code, reason detail, and policy source. Do not invent a date or fall back to an ad hoc retry schedule.

C.13.2 Unexpected decision

- Verify the integration is using the intended legacy or next-generation contract.
- Confirm attempt_number and decline evidence.
- Compare issuer BIN, card brand, amount units, currency, and timestamps.
- Review policy_source, matched_policy_id, reason codes, and confidence.
- Confirm that optional fields were not silently omitted by client serialization.

C.13.3 Outcome does not match a decision

- Send request_id and decision_id whenever available.
- Use the same token and attempt_number as the executed operation.
- Report the actual processor result and timestamp.
- Inspect matched_by in the outcome response.
- Do not report recovery merely because a retry was scheduled.

Confidence is not a guarantee

Confidence describes evidence strength. It does not promise authorization success, and it should not override the explicit decision or fixed retry schedule.

C.14 Investigation Runs

C.14.1 Run is not listed

- Confirm the caller has administrative authorization.
- Confirm the correct tenant and environment.
- Verify the upload_job_id or job_id from the ingestion response.
- Check whether the run has been created yet.
- Do not assume a merchant API key authorizes /v1/admin/investigation-runs.

C.14.2 Run is complete but reporting is empty

21. Confirm run status and row counts.
22. Confirm Recovery Truth population.
23. Confirm radar and issuer-health generation.
24. Confirm monitoring-event and timeline population.
25. Confirm cohort memory and classification persistence.
26. Confirm reporting or command-center composition.

Backfill is remediation, not the normal path

On a clean deployment, completed runs should populate downstream resources through automatic bridges. Use previewed, tenant-scoped backfill only after locating a specific missing bridge.

C.14.3 Polling never reaches terminal state

Increase the polling interval, inspect runtime health and worker status, and stop after an application-defined timeout. Do not create a second run merely because the first is slow.

C.15 Webhook Troubleshooting

C.15.1 Subscription creation fails

- Confirm callback_url is present and within 8 to 2,048 characters.
- Provide 1 to 20 event names.
- Remove unknown top-level fields.
- Verify merchant authentication and plan capability.
- Confirm HTTPS and production callback policy with the active deployment contract.

C.15.2 Deliveries fail

Check	Why
Subscription status and URL	The subscription may be deleted, disabled, or misconfigured.
DNS and TLS	Zahlen must reach and trust the callback endpoint.
Consumer response time	Slow handlers can cause timeouts and redelivery.
HTTP response code	Non-success responses normally trigger delivery failure handling.
Signature/verification policy	A stale secret or wrong raw-body handling can reject valid deliveries.
Dependency health	The callback may be up while its database or queue is unavailable.

C.15.3 Duplicate and out-of-order deliveries

- Store a stable delivery/event identifier with a uniqueness constraint.
- Acknowledge quickly and process asynchronously.
- Make business effects idempotent.
- Do not assume ordering across retries or event types.
- Quarantine unknown event types rather than failing the entire consumer.

Verification contract is deployment-specific

The subscription schema does not define a universal signing algorithm or header. Use the active Zahlen webhook outcome and verification contract; never invent one from examples.

C.16 Observability and Correlation

C.16.1 Minimum structured log fields

Category	Recommended fields
Request	timestamp, environment, method, route, status, latency_ms
Identity	safe key_id/fingerprint, merchant_id where returned
Correlation	event_id, batch_id, upload_job_id, request_id, decision_id, outcome_id
Retry	idempotency_key fingerprint, retry_count, next_delay_ms
Error	error category, safe message, response request ID
Webhook	subscription_id, event type, delivery ID, verification result

C.16.2 Do not log

- Full API keys or webhook secrets.
- Full payment card numbers, CVV, passwords, or raw bank credentials.
- Unredacted payment tokens or customer data unless explicitly approved.
- Entire metadata objects without a classification and retention policy.

C.16.3 Monitoring signals

- Authentication failure rate.
- HTTP 422 rate by field/path.
- HTTP 429 rate and quota utilization.
- Decision latency and error rate.
- Outcome reporting lag and match rate.
- Webhook failure, retry, and duplicate rate.
- Unresolved or long-running investigation runs.

C.17 Escalation and Production Checklist

C.17.1 Escalate immediately when

- One tenant can view another tenant's data.
- An API key is exposed or used after revocation.
- Duplicate payment authorizations may have occurred.
- Audit or correlation evidence is missing for a privileged action.
- Sustained errors affect many tenants or production payment flows.

C.17.2 Troubleshooting checklist

Area	Ready when
Connectivity	Health/version checks work from the production network.
Authentication	Key injection, rotation, and revocation are tested.
Contracts	Required, optional, null, unknown-field, and maximum-size tests pass.
Idempotency	Uncertain POST results reconcile without duplicate effects.
Rate limits	429 handling honors Retry-After and uses jittered backoff.
Payment schedule	No technical retry can create attempts outside Days 1, 2, 6, and 14.
Correlation	All durable identifiers appear in structured logs.
Webhooks	Verification, deduplication, asynchronous processing, and replay are tested.
Investigation runs	Completed runs can be traced into downstream reporting.
Security	Secrets and sensitive payment data are redacted from logs and support evidence.

Final rule

Fix the first broken boundary or durable link in the evidence chain. Do not hide failures with synthetic data, bypass tenant filters, raise quotas without diagnosis, or create uncontrolled payment retries.