SIR Giving Partner API

Complete reference and integration guide for the SIR Giving Partner API. HMAC-authenticated REST endpoints for actions, users, donations, campaigns, token pools, and webhooks.

- 1. Start Here
- 2. Get Credentials
- 3. Authentication & HMAC Signing
- 4. API Reference
- 5. Webhooks
- 6. End-to-End Scenarios

1. Start Here

Use the SIR Giving Partner API to add SIR rewards to your product. You can create donation links, reward users for actions in your own system, track balances, and receive webhook events when rewards are issued or reversed.

Most teams use one of two paths:

If you want to...	Start with
Add a donation button or embedded donation flow	A publishable key (`pk_...`) and the donation endpoints
Reward users for purchases, volunteering, referrals, or other actions	A secret key (`sk_...`), HMAC signing, and the actions endpoints
Listen for reward status changes	Webhooks
Test before launch	Sandbox credentials and a sandbox token pool

What you need before you call the API

You need three things:

1. A partner account.
2. An API key pair for the environment you are using.
3. A funded token pool if you want to distribute SIR tokens.

Creating credentials lets you authenticate. It does not automatically mean you can issue real rewards. Rewards draw from a token pool, and production token pools require SIR Giving approval.

Base URLs

Environment	Base URL	Key prefix
Sandbox	`https://devapi.sirgiving.org`	`pk_test_...`, `sk_test_...`
Production	`https://api.sirgiving.org`	`pk_live_...`, `sk_live_...`

All partner integration endpoints are under `/v1/partner/`.

Interactive API docs are available at `/partner-api` on each host.

Choose your first tutorial

Goal	Use this
I want to make my first signed backend request	Authentication & HMAC Signing
I want to add a donation button	End-to-End Scenarios: Donation widget
I want to reward a user for an action	End-to-End Scenarios: Reward a user
I want to receive events	Webhooks

Key concepts

Partner

Your organization or developer account in SIR Giving. API keys, token pools, campaigns, users, and webhooks all belong to a partner.

API key pair

Each key issuance returns:

- A publishable key, such as `pk_test_...`, safe for browser use.
- A secret key, such as `sk_test_...`, for backend use only.
- An HMAC signing secret, used to sign server-to-server requests.

The secret key and HMAC secret are shown once. Store them in a secret manager.

Token pool

A funded bucket of SIR tokens allocated to your partner. Reward actions debit this pool. If there is no active pool in the same environment as your API key, action submission can authenticate but still fail during processing.

Action

The unit of reward work you send to SIR Giving. For example: a purchase, volunteer shift, donation, referral, or advocacy event. Actions include an `idempotencyKey` so retries do not duplicate rewards.

Partner user

A user from your system mirrored into SIR Giving using your stable `externalUserId`. Partner users receive balances when actions reward them.

Webhook

An outbound event sent from SIR Giving to your server when important things happen, such as `action.completed` or `token_pool.low_balance`.

The shortest path to a working integration

1. Create or receive sandbox credentials.
2. Store the `sk_...` key and HMAC secret in your backend environment.
3. Make a signed `GET /v1/partner/users` request.
4. If you are building a widget, create a donation link with the `pk_...` key.
5. If you are issuing rewards, confirm you have an active sandbox token pool.
6. Submit a test action with a unique `idempotencyKey`.
7. Register a webhook and send a test event.
8. Repeat the same flow in production after your production token pool is approved.

Common confusion

My key works, but actions fail. Why?

Authentication only proves your key is valid. Reward actions also need an active token pool in the same environment.

Can I use a publishable key for server actions?

No. Mutation endpoints such as action submission and webhook registration require a secret key.

Can I use sandbox keys against production?

Do not do this. Use `test` keys with `devapi.sirgiving.org` and `live` keys with `api.sirgiving.org`.

2. Get Credentials

This page explains how you get access, what credentials mean, and what still has to be approved before rewards can be issued.

Sandbox vs production

Environment	Purpose	What you can do
Sandbox	Build and test your integration	Create keys, make signed API calls, test donation links, test reward flows if a sandbox token pool is provisioned
Production	Issue real rewards	Requires approval, production credentials, and a production token pool

Self-service credentials

If you have a SIR Giving account with access to partner keys, creating your first API key automatically creates a Partner record linked to your user.

Use:

```
POST /v1/partner/keys
Authorization: Bearer <your user JWT>
Content-Type: application/json

{
  "name": "Sandbox Integration",
  "environment": "sandbox",
  "keyType": "secret"
}
```

The response includes:

```
{
  "id": "key_...",
  "partnerId": "...",
```

```
  "name": "Sandbox Integration",
  "environment": "sandbox",
  "publicKey": "pk_test_...",
  "secretKey": "sk_test_...",
  "hmacSecret": "64-character-hex-string"
}
```

Store `secretKey` and `hmacSecret` immediately. They are not shown again.

Production approval

Production rewards require approval because SIR tokens have real value and every reward draws from a token pool.

Before production launch, SIR Giving needs:

Item	Why it matters
Partner name and legal entity	Account and compliance record
Contact email	Operational alerts and key expiry notices
Use case summary	Confirms acceptable use and token economics
Expected monthly action volume	Sets rate limits and token pool size
Action types	Confirms what events you will submit
Stakeholder model	Confirms who receives rewards
Webhook URL	Lets SIR Giving send reward lifecycle events

After approval, SIR Giving provides or enables:

- Production API credentials.
- A production token pool.
- Campaign or reward rules, if needed.
- Production webhook configuration.

What each key is for

Key	Where it belongs	Use it for
`pk_test_...` or `pk_live_...`	Browser or mobile app	Donation links, public config, organization lookup

Key	Where it belongs	Use it for
`sk_test_...` or `sk_live_...`	Backend only	Users, actions, campaigns, token pools, webhooks, dashboard data
`hmacSecret`	Backend secret manager	Signing server-to-server requests

Check your partner record

After creating a key, verify your Partner record:

```
GET /v1/partner/keys/partner-info
Authorization: Bearer <your user JWT>
```

You should see your partner ID, name, slug, status, enabled features, and rate limit.

Check your keys

```
GET /v1/partner/keys
Authorization: Bearer <your user JWT>
```

This lists keys but does not return secret values.

Go-live checklist

Before switching to production:

- You have a `pk_live_...` and `sk_live_...` key.
- Your backend signs requests with the production `hmacSecret`.
- You have an active production token pool.
- Your webhook endpoint is reachable from the public internet.
- Your webhook handler verifies `X-SIR-Signature`.
- Your action requests use stable, unique `idempotencyKey` values.
- You have tested retries and rate limit handling.

3. Authentication & HMAC Signing

Server-to-server requests must be signed. Browser widget requests only need a publishable key.

Which authentication do I use?

Endpoint type	Key	Headers
Browser donation/config endpoints	`pk_...`	`X-Partner-Key`
Backend read endpoints	`sk_...` plus HMAC	`X-Partner-Key`, `X-Timestamp`, `X-Signature`
Backend write endpoints	`sk_...` plus HMAC	`X-Partner-Key`, `X-Timestamp`, `X-Signature`

If an endpoint says it requires a secret key, `pk_...` keys are rejected.

Make your first signed request

Set environment variables:

```
export SIR_BASE_URL="https://devapi.sirgiving.org"
export SIR_SECRET_KEY="sk_test_..."
export SIR_HMAC_SECRET="your-hmac-secret"
```

Call a read endpoint:

```
TS=$(date +%s)
PATH_="/v1/partner/users"
BODY=""
BODY_HASH=$(printf '%s' "$BODY" | shasum -a 256 | awk '{print $1}')
SIG=$(printf '%s' "${TS}GET${PATH_}${BODY_HASH}" \
  | openssl dgst -sha256 -hmac "$SIR_HMAC_SECRET" -hex \
  | awk '{print $2}')

curl "$SIR_BASE_URL$PATH_" \
```

```
  -H "X-Partner-Key: $SIR_SECRET_KEY" \
  -H "X-Timestamp: $TS" \
  -H "X-Signature: $SIG"
```

If the request succeeds, your key, timestamp, and signature are valid.

Signature algorithm

The server computes the same value and compares it to `X-Signature`.

```
bodyHash      = SHA256_hex(rawRequestBody)
signedPayload = timestamp + METHOD + pathWithQueryString + bodyHash
signature     = HMAC_SHA256_hex(hmacSecret, signedPayload)
```

Rules:

- `timestamp` is the same string sent in `X-Timestamp`.
- `METHOD` is uppercase, such as `GET` or `POST`.
- `pathWithQueryString` includes the query string.
- `rawRequestBody` must be the exact bytes sent over HTTP.
- Empty bodies use the SHA-256 hash of the empty string.
- The timestamp must be within 5 minutes of server time.

Node.js helper

```
import crypto from 'crypto';

export function signSirRequest(method: string, path: string, body?: unknown) {
  const timestamp = Math.floor(Date.now() / 1000).toString();
  const rawBody = body === undefined ? '' : JSON.stringify(body);
  const bodyHash = crypto.createHash('sha256').update(rawBody).digest('hex');
  const signedPayload = `${timestamp}${method.toUpperCase()}${path}${bodyHash}`;
  const signature = crypto
    .createHmac('sha256', process.env.SIR_HMAC_SECRET!)
    .update(signedPayload)
    .digest('hex');

  return {
    rawBody,
```

```
    headers: {
      'X-Partner-Key': process.env.SIR_SECRET_KEY!,
      'X-Timestamp': timestamp,
      'X-Signature': signature,
      'Content-Type': 'application/json',
    },
  };
}
```

Use the exact `rawBody` string returned by the signing function as the request body. Do not let your HTTP client re-serialize it after signing.

Common errors

Error	Meaning	Fix
`INVALID_API_KEY`	Missing, malformed, inactive, expired, or wrong key	Check the key value and environment
`TIMESTAMP_EXPIRED`	Timestamp missing or outside the 5-minute window	Sync server time and regenerate signature
`INVALID_SIGNATURE`	HMAC did not match	Check path, query string, body bytes, and HMAC secret
`PARTNER_NOT_ACTIVE`	Partner status is not active	Check partner approval/status
`PARTNER_SUSPENDED`	Partner is suspended	Contact SIR Giving support

Idempotency

Action submissions require an `idempotencyKey`. Use a stable key for the real-world operation, such as `order_98765` or `volunteer_shift_abc123`.

If a network call times out, retry with the same `idempotencyKey`. SIR Giving returns the original result instead of issuing duplicate rewards.

4. API Reference

All paths are relative to:

- Sandbox: `https://devapi.sirgiving.org`
- Production: `https://api.sirgiving.org`

Auth legend

Label	Meaning
JWT	User login token from the SIR Giving app
Widget	`X-Partner-Key: pk_...` only
HMAC	`X-Partner-Key`, `X-Timestamp`, and `X-Signature`
HMAC, secret key	HMAC request with `sk_...`; publishable keys are rejected

Partner key management

These endpoints are for signed-in users managing their own partner credentials.

Method	Path	Auth	Purpose
POST	`/v1/partner/keys`	JWT	Create an API key pair
GET	`/v1/partner/keys`	JWT	List your API keys
GET	`/v1/partner/keys/partner-info`	JWT	Get your Partner record
GET	`/v1/partner/keys/:id`	JWT	Get one key record
PATCH	`/v1/partner/keys/:id`	JWT	Rename, deactivate, or set expiry
DELETE	`/v1/partner/keys/:id`	JWT	Delete/revoke a key

Partner auth

These endpoints support the partner portal.

Method	Path	Auth	Purpose
POST	`/v1/partner/auth/login`	Email/password	Login to partner portal
GET	`/v1/partner/auth/me`	JWT	Get current partner user and publishable key

Widget/config endpoints

Method	Path	Auth	Purpose
GET	`/v1/partner/config`	Widget	Get partner widget config
GET	`/v1/partner/organizations/:slug`	Widget	Look up nonprofit details
POST	`/v1/partner/donations/create-link`	Widget	Create a donation checkout link
GET	`/v1/partner/donations/status/:partnerDonationId`	Widget	Check donation reward status

Users

Method	Path	Auth	Purpose
GET	`/v1/partner/users`	HMAC	List partner users
POST	`/v1/partner/users`	HMAC, secret key	Create a partner user
GET	`/v1/partner/users/:externalId`	HMAC	Get a user by your external ID
PATCH	`/v1/partner/users/:externalId`	HMAC, secret key	Update user metadata
GET	`/v1/partner/users/:externalId/balance`	HMAC	Get user balance
GET	`/v1/partner/users/:externalId/transactions`	HMAC	Get user action history

Actions

Method	Path	Auth	Purpose
POST	`/v1/partner/actions/submit`	HMAC, secret key	Submit one reward action
GET	`/v1/partner/actions`	HMAC	List actions
GET	`/v1/partner/actions/:id`	HMAC	Get action status/details

Method	Path	Auth	Purpose
POST	`/v1/partner/actions/:actionId/reverse`	HMAC, secret key	Reverse a completed action
POST	`/v1/partner/actions/bulk`	HMAC, secret key	Submit up to 100 actions synchronously

`GET /v1/partner/actions/bulk/:jobId` is deprecated. Bulk actions are processed synchronously and return their result from `POST /bulk`.

Token pools

Method	Path	Auth	Purpose
GET	`/v1/partner/token-pools`	HMAC	List allocated token pools
GET	`/v1/partner/token-pools/:id/balance`	HMAC	Get pool balance
POST	`/v1/partner/token-pools/request`	HMAC, secret key	Request more allocation
GET	`/v1/partner/token-pools/requests`	HMAC	List allocation requests

Campaigns

Method	Path	Auth	Purpose
GET	`/v1/partner/campaigns`	HMAC	List campaigns
GET	`/v1/partner/campaigns/active`	HMAC	Get active campaign
GET	`/v1/partner/campaigns/:id`	HMAC	Get campaign details
GET	`/v1/partner/campaigns/:id/analytics`	HMAC	Get campaign analytics
POST	`/v1/partner/campaigns`	HMAC, secret key	Create campaign
PATCH	`/v1/partner/campaigns/:id`	HMAC, secret key	Update campaign

Transactions

Method	Path	Auth	Purpose
GET	`/v1/partner/transactions`	HMAC	List partner transactions

Method	Path	Auth	Purpose
GET	`/v1/partner/transactions/:id`	HMAC	Get transaction details

Dashboard

Method	Path	Auth	Purpose
GET	`/v1/partner/dashboard/summary`	HMAC	Pool, activity, and user summary
GET	`/v1/partner/dashboard/trends`	HMAC	Daily activity trends
GET	`/v1/partner/dashboard/top-earners`	HMAC	Top rewarded users
GET	`/v1/partner/dashboard/webhooks/health`	HMAC	Webhook health
GET	`/v1/partner/dashboard/api-usage`	HMAC	API usage metrics
GET	`/v1/partner/dashboard/recent-activity`	HMAC	Recent actions
GET	`/v1/partner/dashboard/integration-health`	HMAC	Integration health
GET	`/v1/partner/dashboard/campaigns/:id/analytics`	HMAC	Campaign performance

Webhooks

Method	Path	Auth	Purpose
POST	`/v1/partner/webhooks`	HMAC, secret key	Register webhook
GET	`/v1/partner/webhooks`	HMAC	List webhooks
DELETE	`/v1/partner/webhooks/:id`	HMAC, secret key	Delete webhook
GET	`/v1/partner/webhooks/:id/deliveries`	HMAC	List deliveries
POST	`/v1/partner/webhooks/:id/deliveries/:deliveryId/retry`	HMAC, secret key	Retry delivery
POST	`/v1/partner/webhooks/:id/test`	HMAC, secret key	Send test webhook

5. Webhooks

Webhooks let SIR Giving notify your backend when something changes. For example, you can receive an event when an action completes, an action fails, a token pool runs low, or a campaign changes status.

Use webhooks when your system needs a durable server-side record of reward outcomes.

How webhook delivery works

1. You register an HTTPS endpoint.
2. SIR Giving returns a webhook signing secret once.
3. SIR Giving sends events to your endpoint.
4. Your server verifies the signature.
5. Your server stores or queues the event.
6. Your server returns a 2xx response quickly.

Register a webhook

```
POST /v1/partner/webhooks
X-Partner-Key: sk_test_...
X-Timestamp: <timestamp>
X-Signature: <signature>
Content-Type: application/json

{
  "url": "https://your-app.example.com/webhooks/sir",
  "description": "Sandbox webhook",
  "eventTypes": ["action.completed", "action.failed"],
  "receiveAllEvents": false
}
```

Response:

```
{
  "id": "webhook-id",
  "url": "https://your-app.example.com/webhooks/sir",
```

```
  "eventTypes": ["action.completed", "action.failed"],
  "secret": "whsec_...",
  "isActive": true
}
```

Store `secret` immediately. It is shown once.

Delivery headers

Header	Value
`Content-Type`	`application/json`
`User-Agent`	`SIRGiving-Webhooks/1.0`
`X-SIR-Signature`	`sha256=<hex>`
`X-SIR-Timestamp`	Unix timestamp in seconds

Event envelope

```
{
  "id": "evt_abc123",
  "type": "action.completed",
  "createdAt": "2026-05-10T09:12:00Z",
  "data": {
    "actionId": "...",
    "tokensDistributed": 50
  }
}
```

Verify signatures

SIR Giving signs the timestamp and raw request body:

```
signedPayload = X-SIR-Timestamp + "." + rawRequestBody
expected = "sha256=" + HMAC_SHA256_hex(webhookSecret, signedPayload)
```

Compare `expected` to `X-SIR-Signature` using constant-time comparison.

Node.js Express example

```
import crypto from 'crypto';
import express from 'express';

const app = express();

app.use('/webhooks/sir', express.raw({ type: 'application/json' }));

app.post('/webhooks/sir', (req, res) => {
  const signature = req.header('X-SIR-Signature') ?? '';
  const timestamp = req.header('X-SIR-Timestamp') ?? '';
  const rawBody = req.body as Buffer;

  const signedPayload = `${timestamp}.${rawBody.toString()}`;
  const expected = 'sha256=' + crypto
    .createHmac('sha256', process.env.SIR_WEBHOOK_SECRET!)
    .update(signedPayload)
    .digest('hex');

  const valid =
    signature.length === expected.length &&
    crypto.timingSafeEqual(Buffer.from(signature), Buffer.from(expected));

  if (!valid) return res.status(401).end();

  const ageSeconds = Math.abs(Date.now() / 1000 - Number(timestamp));
  if (ageSeconds > 300) return res.status(401).end();

  const event = JSON.parse(rawBody.toString());
  // Store or enqueue the event here.

  return res.status(200).end();
});
```

Retry behavior

- Return a 2xx response when you accept the event.
- Anything else is treated as a failed delivery.
- Failed deliveries are retried with backoff.
- Slow handlers can cause duplicate deliveries, so return quickly and process asynchronously.
- You can inspect delivery history with `GET /v1/partner/webhooks/:id/deliveries`.
- You can manually retry a failed delivery with `POST /v1/partner/webhooks/:id/deliveries/:deliveryId/retry`.

Test your webhook

```
POST /v1/partner/webhooks/:id/test
X-Partner-Key: sk_test_...
X-Timestamp: <timestamp>
X-Signature: <signature>
```

Then check:

```
GET /v1/partner/webhooks/:id/deliveries
```

Common event types

Event	When it fires
`action.completed`	An action processed successfully
`action.failed`	Action processing failed
`action.reversed`	An action was fully or partially reversed
`transaction.completed`	Token distribution transaction completed
`transaction.reversed`	Token distribution transaction reversed
`token_pool.low_balance`	Pool dropped below threshold
`token_pool.depleted`	Pool has no remaining balance
`token_pool.refilled`	Pool was topped up
`token_pool_request.approved`	Allocation request approved
`token_pool_request.rejected`	Allocation request rejected
`campaign.activated`	Campaign moved to active
`campaign.paused`	Campaign paused
`campaign.completed`	Campaign ended

Event	When it fires
`api_key.expiring`	One of your keys is nearing expiry

6. End-to-End Scenarios

Use these tutorials when you want to build a real integration path from start to finish.

Scenario 1: Add a donation button

You will create a donation checkout link from the browser using a publishable key.

What you need

- A publishable key, such as `pk_test_...`.
- A nonprofit slug, such as `american-red-cross`.
- A page or component where the user clicks Donate.

Step 1: Create the donation link

```
const response = await fetch('https://devapi.sirgiving.org/v1/partner/donations/create-link',
{
  method: 'POST',
  headers: {
    'X-Partner-Key': 'pk_test_...',
    'Content-Type': 'application/json',
  },
  body: JSON.stringify({
    nonprofitSlug: 'american-red-cross',
    amount: 2500,
    currency: 'USD',
    frequency: 'once',
    provider: 'stripe',
    donorEmail: 'jane@example.com',
    firstName: 'Jane',
    lastName: 'Doe'
  }),
});

const { donationLink, partnerDonationId } = await response.json();
```

```
window.location.href = donationLink;
```

Step 2: Store the donation ID

Store `partnerDonationId` in your session or database. You use it later to show status.

Step 3: Check status

```
GET /v1/partner/donations/status/:partnerDonationId
X-Partner-Key: pk_test_...
```

What you built

You now have a browser-based donation flow. The user clicks your button, SIR Giving creates a checkout link, and you can poll for reward status or listen for webhook events.

Scenario 2: Reward a user for an action

You will submit a backend action when something valuable happens in your system, such as a purchase or completed volunteer shift.

What you need

- A secret key, such as `sk_test_...`.
- The matching HMAC secret.
- An active token pool in the same environment.
- A stable user identifier from your system.

Step 1: Decide your idempotency key

Use a key that maps to one real-world event:

```
purchase_98765
volunteer_shift_abc123
referral_signup_555
```

If you retry the same action, reuse the same key.

Step 2: Submit the action

```
POST /v1/partner/actions/submit
X-Partner-Key: sk_test_...
X-Timestamp: <timestamp>
X-Signature: <signature>
Content-Type: application/json

{
  "idempotencyKey": "purchase_98765",
  "actionType": "PURCHASE",
  "amount": 49.99,
  "currency": "USD",
  "stakeholders": [
    {
      "stakeholderTypeCode": "CUSTOMER",
      "partnerUserId": "user_42",
      "userEmail": "customer@example.com",
      "userFirstName": "Jane",
      "userLastName": "Doe"
    }
  ],
  "autoCreateUsers": true,
  "metadata": {
    "orderId": "98765"
  }
}
```

Step 3: Store the action ID

Response:

```
{
  "actionId": "65f1...",
  "idempotencyKey": "purchase_98765",
  "status": "COMPLETED",
  "tokensDistributed": 50,
  "transactionIds": ["65f2..."]
}
```

Store `actionId` with your order. You need it for refunds or support.

What you built

You now have a backend reward flow. Your system sends one signed request, SIR Giving debits your token pool, credits the partner user, and returns a completed result.

Scenario 3: Reverse a reward after a refund

If the real-world action is reversed, reverse the SIR reward too.

```
POST /v1/partner/actions/65f1.../reverse
X-Partner-Key: sk_test_...
X-Timestamp: <timestamp>
X-Signature: <signature>
Content-Type: application/json

{
  "reversalPercentage": 100,
  "reason": "Order refunded",
  "refundIdempotencyKey": "refund_order_98765"
}
```

If the user already spent some rewards, SIR Giving may create a debt that offsets future earnings.

Scenario 4: Register and test a webhook

Step 1: Register the webhook

```
POST /v1/partner/webhooks
X-Partner-Key: sk_test_...
X-Timestamp: <timestamp>
X-Signature: <signature>
Content-Type: application/json
```

```
{
  "url": "https://your-app.example.com/webhooks/sir",
  "eventTypes": ["action.completed", "action.failed"],
  "receiveAllEvents": false
}
```

Store the returned webhook `secret`.

Step 2: Send a test event

```
POST /v1/partner/webhooks/:id/test
X-Partner-Key: sk_test_...
X-Timestamp: <timestamp>
X-Signature: <signature>
```

Step 3: Inspect delivery history

```
GET /v1/partner/webhooks/:id/deliveries
X-Partner-Key: sk_test_...
X-Timestamp: <timestamp>
X-Signature: <signature>
```

Scenario 5: Preflight before a batch run

Before submitting a large batch:

1. `GET /v1/partner/dashboard/integration-health`
2. `GET /v1/partner/token-pools/:id/balance`
3. `GET /v1/partner/campaigns/active`
4. `GET /v1/partner/dashboard/webhooks/health`
5. `GET /v1/partner/dashboard/api-usage?days=1`

Then submit up to 100 actions at a time:

```
POST /v1/partner/actions/bulk
```

Troubleshooting

Problem	Likely cause	Fix
`INVALID_SIGNATURE`	Signed path/body does not match request	Sign exact path and exact body bytes
`NO_SANDBOX_POOL`	Sandbox key is valid but no sandbox pool exists	Ask SIR Giving to provision a sandbox pool
`No active token pool found`	Production pool missing or inactive	Confirm pool allocation before launch
`403` on write endpoint	You used `pk_...` or missing scope	Use `sk_...` and request needed scope
Duplicate reward concern	Retried request after timeout	Reuse the same `idempotencyKey`
Webhook repeats	Your endpoint did not return 2xx fast enough	Queue work and return quickly