AIOTIC

Documentation revision 3

Integrator Guide

Connecting AIOTIC purchase-order processing to your ERP or business software: contracts, flows, master-data sync, Python SDK and API reference.

Documentation revision 3 · Published 8 September 2026 · commit 6862d5e
Verified against AIOTIC API v1.0.0 · Online edition with search, downloadable spec and LLM-native docs: https://developers.aiotic.ai
AIOTIC is a product of DevOps Company. This document may be shared with integration partners.

Contents

START HERE
What AIOTIC does
Quick start (15 minutes)
Integration models
Onboarding checklist

CONCEPTS
Tenants, URLs & environments
Authentication & keys
Order lifecycle & statuses
The purchase-order model
Reference data
Errors, idempotency & limits

RECEIVING ORDERS
The ERP receive endpoint
Response contract
Idempotency & failure handling
Field mapping cookbook
Validation when your ERP has no functional API
Testing your endpoint

KEEPING DATA IN SYNC
Initial load
Event-driven sync
Reconciliation
Multi-language products
Archiving & superseded customers

DRIVING ORDERS VIA THE API
Submitting documents
Polling & notifications
ATTENTION, FAILED & retries
Sending to the ERP
The processing webhook
Rejected e-mails
Headless limits today

PYTHON SDK
Overview & install
Bootstrapping a service
Client reference
Validation & business-rule pipeline
ERP adapters (functional vs data API)
Sync engine
Webhooks & receivers
Order watcher
CLI
Mock AIOTIC server

API REFERENCE
Overview
Health & status
Orders — intake
Orders — status & files
ERP hand-off
Rejected e-mails
Customers
Products
Customer item mappings
Mailbox
Outbound: webhooks
Schemas

SECURITY & OPERATIONS
Security checklist
Data, privacy & retention
Monitoring

AI-ASSISTED INTEGRATION
LLM-native docs (llms.txt)
AIOTIC MCP server

APPENDIX
Proposal: headless API additions
Status reference
Glossary
FAQ
Changelog

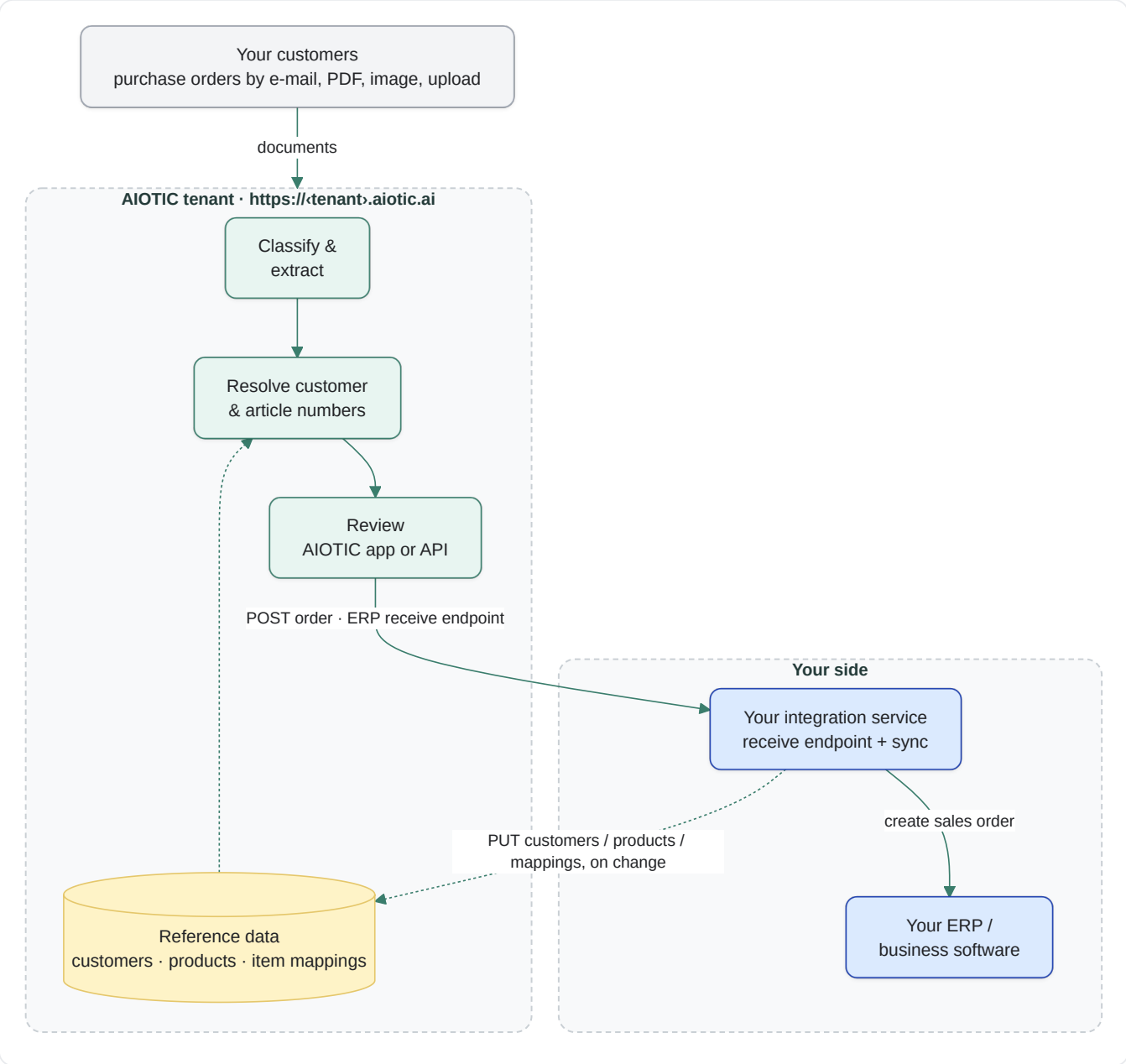
START HERE

What AIOTIC does

AIOTIC reads the purchase orders your customers send you — as e-mail attachments, PDFs, scans, photos or direct uploads — and turns each one into a structured, validated order that your ERP can book as a sales order. Operators review anything the AI is not sure about in the AIOTIC app; you receive the final result through one HTTPS call.

The 30-second version

1. **Intake.** A purchase order arrives (mailbox watched by AIOTIC, or your system uploads a file through the API).
2. **Classification.** AIOTIC decides whether the e-mail is a purchase order at all. Quotations, invoices, delivery notes and marketing mail are recorded as *rejected* and never extracted.
3. **Extraction.** AIOTIC reads the document and the e-mail body and produces a purchase order: header, customer, ship-to, line items. E-mail instructions ("change line 3 to 200 pieces") override the attachment.
4. **Resolution.** In a series of deterministic stages, AIOTIC identifies **which of your customers** sent the order (using the customer records you synced) and resolves each line to **your article number** (using your product catalog and customer item mappings). This is where the product's ontology and semantic layer do their work.
5. **Review.** Clean orders land as `PROCESSED`. Anything uncertain — an unknown article, an unidentified customer, an unreadable quantity — lands as `ATTENTION` for a person to decide.
6. **Hand-off.** When an operator (or your code) sends the order, AIOTIC `POST`s it to **your ERP receive endpoint** and stores the order reference your ERP returns.



One tenant per company. Your side is the blue part: a receive endpoint and a sync job.

What you build

Component	Required?	Typical build time for your team
ERP receive endpoint — one `POST` handler that books an order and answers `{"success": true, "order_number": "…"}`	Yes	A few hours
Master-data sync — push customers, products and customer item mappings to AIOTIC whenever they change	Yes (accuracy depends on it)	Half a day to a day, depending on whether your ERP emits change events
Order polling / own review UI — only if you do not want operators to use the AIOTIC app	No	Several days
Processing webhook receiver — early signal when extraction finishes	No	Under an hour

The last column is an indication of the development effort on **your** side when you use the Python SDK or follow the contracts in this guide; it excludes testing against your ERP and the master-data clean-up that most projects discover along the way. The SDK implements all four components, including a validation pipeline for ERPs that cannot validate orders themselves.

What AIOTIC is not

- It is **not a document archive** — it keeps originals and artifacts for processing and support, but your ERP remains the system of record.
- It **never sends an order to your ERP on its own**. Sending is an explicit action: an operator in the app, or your code calling `POST /erp/send/{request_id}`.
- It does **not** need to know your prices or stock. It needs to know *which articles exist* and *which customers exist*.

Vocabulary

Term	Meaning
Tenant	One AIOTIC deployment for one company, with its own base URL (`https://<tenant>.aiotic.ai`) and keys.
Purchase order	The document your customer sends you: their order, from their point of view a *purchase*. This is what AIOTIC reads and what the API field `purchase_order` contains.
Sales order	What your ERP creates from it. Your receive endpoint turns a purchase order into a sales order and returns its number.
Order / **request**	The neutral word for one purchase order moving through AIOTIC, identified by a `request_id` (UUID).
Supplier	*You* — the company receiving the purchase order. Pinned from tenant configuration, not extracted.
Customer	The debtor who sent the order. Identified against your customer records; carried as `customer_id` (= your debtor number).
Article number	Your SKU. Line items are resolved to it; unknown ones trigger `ATTENTION`.
Customer item mapping	"Customer 58931 calls PROD-001 *LT-ART-001*". Lets AIOTIC translate customer codes to yours.
ERP receive endpoint	The URL AIOTIC calls with a reviewed order.
Processing webhook	Optional URL AIOTIC calls the moment extraction finishes, before review.

Next: Integration models →

START HERE

Quick start (15 minutes)

You will run a **mock AIOTIC tenant**, a **receive endpoint** with the SDK's validation pipeline, upload an order, and watch it flow into a (demo) ERP. Nothing here touches a real tenant.

> **Prerequisites** Python 3.11+ and `pip` (or `uv`). No AIOTIC account needed for this exercise.

1. Install the SDK

```
<span class="tab">pip</span>
python -m venv .venv && source .venv/bin/activate
pip install "aiotic-sdk[all]"
```

```
<span class="tab">uv</span>
uv venv && source .venv/bin/activate
uv pip install "aiotic-sdk[all]"
```

2. Start a mock tenant

```
aiotic mock
# Mock AIOTIC tenant on http://localhost:8080
# keys: mock-integration-key (all endpoints), mock-sync-key (master data only)
```

The mock implements the public API with realistic behaviour: uploads move through `QUEUED` → `PROCESSING` → `PROCESSED` in a few seconds, a file name containing `attention` produces an `ATTENTION` order with an unresolved article, `fail` produces `FAILED`, and `/erp/send` really calls your endpoint.

3. Bootstrap your service

In a second terminal:

```
aiotic init           # answers: base URL http://localhost:8080, key mock-integration-key
aiotic doctor
```

`init` writes a `.env` (base URL, keys, a freshly generated `AIOTIC_ERP_RECEIVE_KEY`) and a `service.py`. `doctor` checks connectivity, both keys and whether master data is present:

```
| check                          | result                        |
| GET /healthcheck               | OK ok                         |
| GET /order_status/list (key)   | OK 0 orders                   |
| master data: customers         | OK 1 records                  |
| master data: products          | OK 3 records                  |
```

4. Run the receive endpoint

```
aiotic serve          # http://localhost:9000  (POST /aiotic/orders, /aiotic/processing, /erp/events)
```

`service.py` is three lines — the in-memory demo ERP behind the default pipeline:

```
from aiotic.service import build_app
from aiotic.erp.memory import InMemoryErp   # replace with your adapter later

app = build_app(erp=InMemoryErp())
```

Tell the mock tenant where your endpoint lives (in production you give this to the AIOTIC team; the mock has a helper):

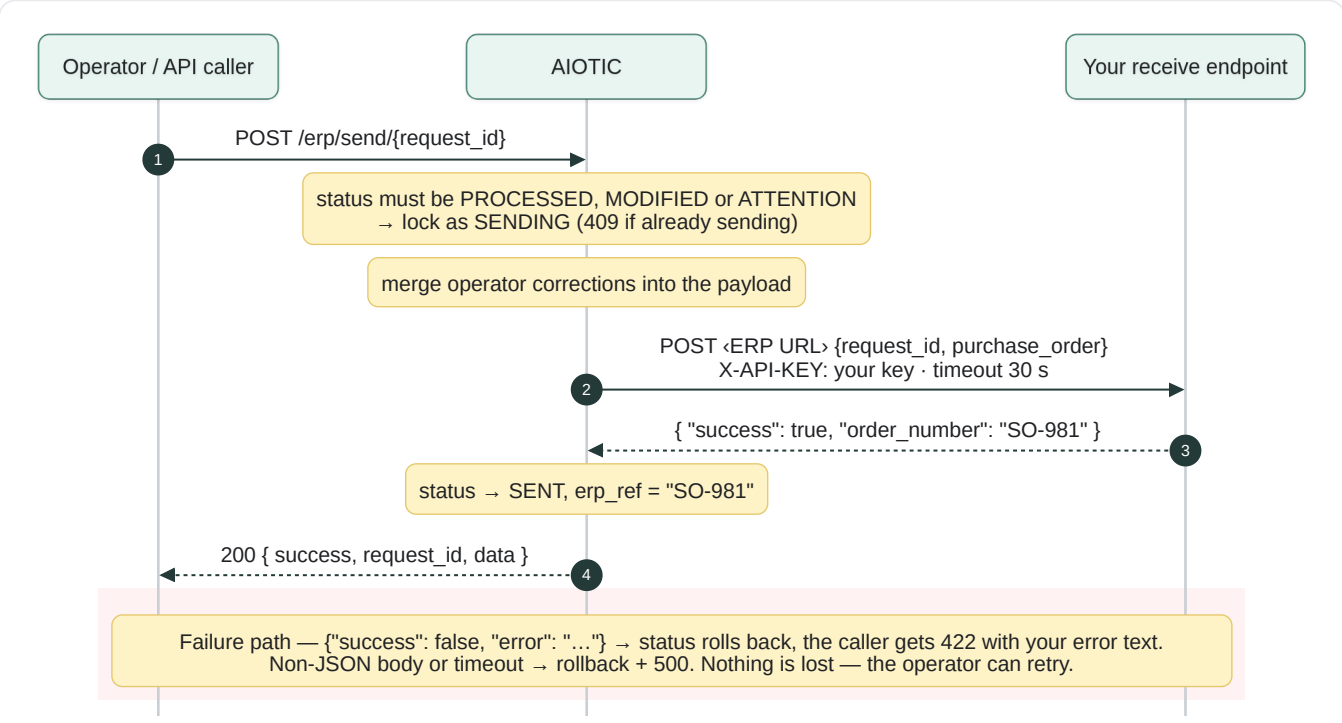
```
curl -X POST localhost:8080/_mock/config -H 'Content-Type: application/json' \
  -d "{\"erp_url\": \"http://localhost:9000/aiotic/orders\", \"erp_key\": \"$(grep AIOTIC_ERP_RECEIVE_KEY
.env | cut -d= -f2)\"}"
```

5. Upload an order and send it

```
echo "fake pdf" > PO-4711.pdf
aiotic orders upload PO-4711.pdf
# uploaded → request_id 7f0c…   then:  7f0c…: PROCESSED — PO-4711, 2 lines

aiotic orders send 7f0c…
# sent — ERP reference SO-1000
```

What happened, in sequence:



The mock did what a real tenant does: locked the order, POSTed it to your endpoint with the key, read success, stored the ERP reference.

6. See a rejection

```
echo "x" > PO-4712-attention.pdf
aiotic orders upload PO-4712-attention.pdf      # → ATTENTION (one line has no article number)
aiotic orders send <request_id>
# ERP error: Line 3 has no article number (items[2].article_number)
```

The pipeline rejected the order **before** it reached the ERP, and the message is exactly what an operator would see in the AIOTIC app. Change `service.py` to use `DataApiAdapter` (a SQLite demo ERP with only tables) and you will see why

that matters:

```
from aiotic.service import build_app
from aiotic.erp.data_api import DataApiAdapter, sqlite_demo

app = build_app(erp=DataApiAdapter(sqlite_demo("erp-demo.db")))
```

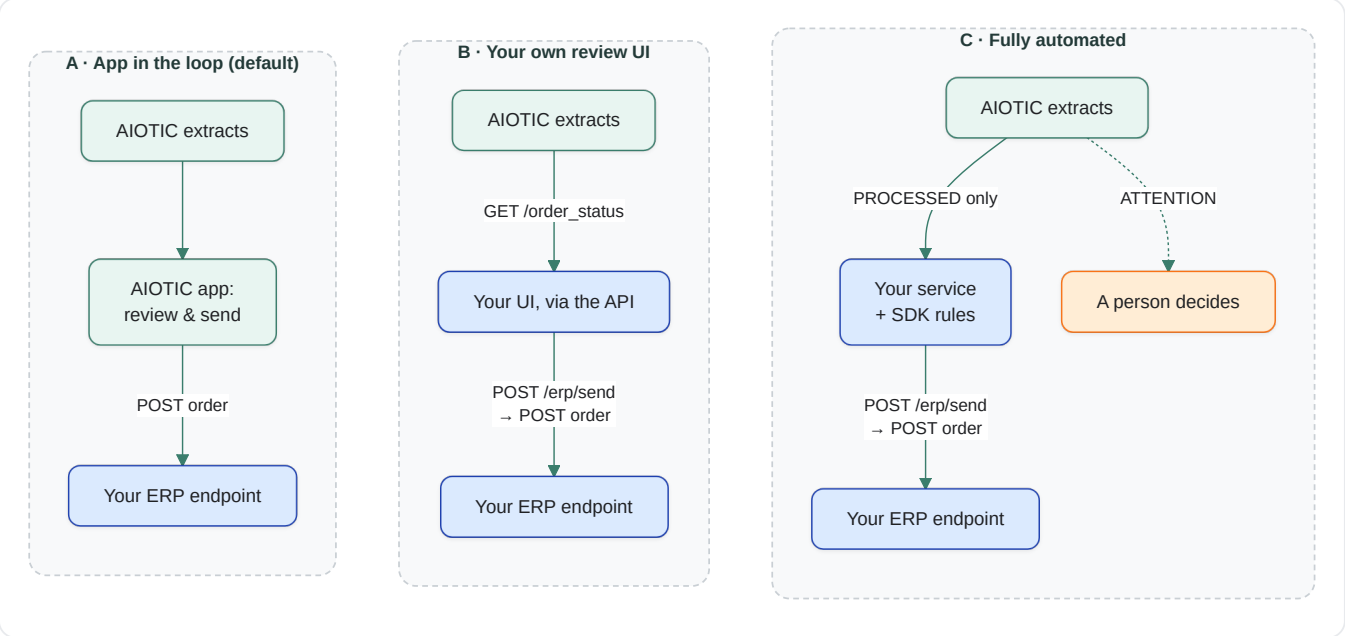
Where to go next

- Understand the contract you just implemented: The ERP receive endpoint
- Replace the demo ERP: ERP adapters
- Feed AIOTIC your real customers and products: Initial load → Event-driven sync
- Go to production: Onboarding checklist

START HERE

Integration models

There are three ways to put AIOTIC into your process. They differ only in **who reviews** and **what triggers the send**. The ERP receive endpoint and master-data sync are the same in all three.



A: operators use the AIOTIC app. B: you build the review step. C: your rules decide.

A — App in the loop (default)

Operators work in the AIOTIC app: they see incoming orders, fix what the AI flagged, and press *Send to ERP*. Your integration is the receive endpoint plus the sync job.

- **Best for:** most companies. Fastest to go live; the app already handles corrections, cancellations, split e-mails, rejected mails and audit trails.
- **You build:** receive endpoint, sync. Optionally the processing webhook as an early signal.

B — Your own review UI

Your application shows the extracted order to your users (for example inside the ERP), and calls AIOTIC's API to send it. Documents still arrive through the AIOTIC mailbox or through your uploads.

- **Best for:** vendors who want the review inside their own product.
- **You build:** everything in A, plus polling of `GET /order_status/list` and a *Send* action on `POST /erp/send/{request_id}`.
- **Know before you start:** today the public API has **no endpoint to submit corrections or to cancel** an order. Corrections made in your UI must be applied on *your* side (in the receive endpoint) before booking; the payload AIOTIC sends is the AI's reading. See Headless limits today and the proposal appendix.

C — Fully automated

Your service polls for `PROCESSED` orders, runs its own rules, and sends without a human. `ATTENTION` orders are routed to a person (a ticket, an e-mail, or the AIOTIC app).

- **Best for:** high-volume, low-risk order streams where the customer base and catalog are complete in AIOTIC.

- **You build:** everything in B, plus a rule set. The SDK's pipeline is designed for exactly this: it applies the validation a functional ERP API would perform, so a data-only ERP does not receive garbage.
- **Non-negotiable:** never auto-send `ATTENTION`. It is AIOTIC explicitly asking for a human.

Choosing

Question	A	B	C
Operators willing to use a second screen?	✔	—	—
Need corrections before booking?	app	your side	your rules
Catalog and customers complete in AIOTIC?	helpful	helpful	required
Time to first order in production	days	weeks	weeks

Most integrations start with **A** and add **C** for a subset of trusted customers later. Nothing in the API forces you to choose up front.

START HERE

Onboarding checklist

What you exchange with the AIOTIC team, and in which order things have to happen for a smooth go-live.

What you receive from AIOTIC

Item	Notes
Tenant base URL	`https://<tenant>.aiotic.ai`. One per company; test and production are separate tenants when both exist.
Sync key	Accepted on `/customer/*`, `/product/*`, `/customer-product/*`. Enough for a standard integration: master data in, orders out through your receive endpoint. Rotatable by a tenant admin.
Integration key (only for headless integrations)	Accepted on every endpoint. Issued per project by the AIOTIC team when you build your own review UI or a fully automated flow. Store it as a secret.
App login (tenant admin)	To enter the ERP endpoint URL and key in the AIOTIC app's settings screen, or ask your AIOTIC contact to do it.

What you give to AIOTIC

Item	Notes
ERP receive endpoint URL	Public HTTPS, e.g. `https://integration.example.com/aiotic/orders`.
ERP endpoint key	The value AIOTIC will send in `X-API-KEY` on every call to that URL. You generate it (`aiotic init` does).
Processing webhook URL + key (optional)	Only if you want the early, pre-review signal.
Allow-list requirements (optional)	If your endpoint is behind an IP allow-list, ask for the tenant's egress addresses.

Sequence

1. **Sandbox first.** Build against the mock tenant until the receive endpoint and sync work end to end.
2. **Load master data** into the test tenant: customers → products → mappings (Initial load). Run `aiotic doctor` — all three counts should be > 0.
3. **Connect the receive endpoint** (send URL + key to AIOTIC, or enter them in the app as tenant admin). Use **Test ERP connection** in the app, or ask for a test send.
4. **Process real documents** in the test tenant. Watch the ratio of `PROCESSED` to `ATTENTION`: a high `ATTENTION` rate almost always means missing products or customer item mappings, not AI trouble.
5. **Switch on incremental sync** (events or reconciliation) so master data stays current without full dumps.
6. **Production tenant:** repeat 2–3 with production keys; keep the test tenant for regression.
7. **Operate:** monitor `/system-status`, your endpoint's error rate, and orders stuck in `ATTENTION`.

Go-live gate

- [ ] Receive endpoint answers `{"success": true, "order_number": …}` within a few seconds and is idempotent on `request_id`
- [ ] Receive endpoint verifies `X-API-KEY` in constant time and is only reachable over HTTPS
- [ ] Business rejections return `{"success": false, "error": "<clear text>"}` — never a bare HTTP 500 with HTML
- [ ] Master data complete: every sellable article in `/product`, every active debtor in `/customer`

- [ ] Incremental sync running; no scheduled full re-upload
- [ ] Keys stored as secrets, not in code or in a browser
- [ ] Someone owns the `ATTENTION` queue (in the AIOTIC app or your own tool)

CONCEPTS

Tenants, URLs & environments

One tenant per company

Every company runs in its own isolated AIOTIC tenant: its own base URL, data, mailbox configuration, keys and master data. Nothing is shared between tenants. There is no "account id" to pass — the base URL *is* the tenant.

```
https://<tenant>.aiotic.ai            ← API base URL (all paths in this guide are relative to it)
```

The AIOTIC app operators use is a separate web application that talks to the same tenant.

Environments

Test and production are simply two tenants with two sets of keys. Keep them apart in configuration, never share a receive-endpoint key between them, and point each tenant to a matching environment of your ERP.

For development without a tenant, use the mock server (`aiotic mock`) — it speaks the same API on `http://localhost:8080` .

Base URL rules

- Always HTTPS in production. HTTP is only for the local mock.
- No trailing slash; paths start with `/` .
- Paths are case-sensitive and exactly as listed in the API reference.
- Path parameters such as customer numbers or article numbers must be URL-encoded (`620206/01` → `620206%2F01`). The SDK does this for you.

Versioning

The API has no version prefix in the path. Changes are **additive**: new fields, new endpoints, new status values. Your code must therefore:

- ignore unknown JSON fields,
- treat unknown `status` values as "not one I act on" rather than failing,
- tolerate `detail` being either a string or an object in error responses.

This guide states which API version it was verified against in the footer; the changelog lists what changed.

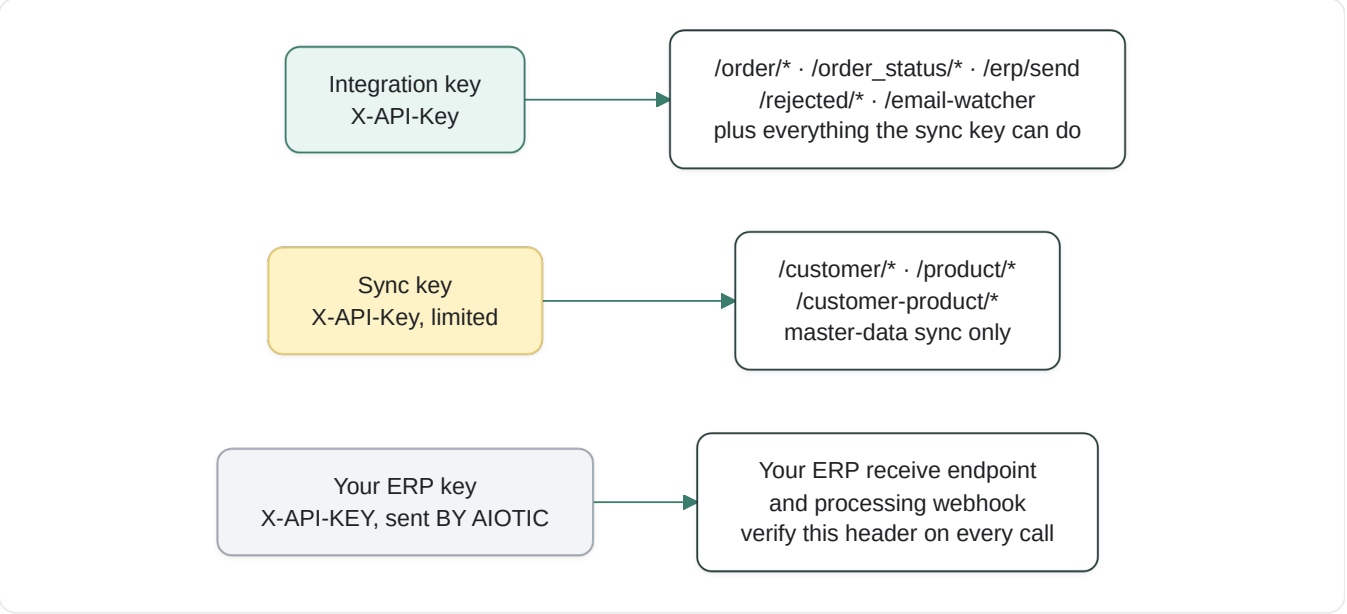
Limits worth knowing

Topic	Today
Rate limiting	None enforced server-side. Be a good neighbour: the SDK defaults to 10 requests/s.
Upload size	Keep files under ~10 MB; the app enforces 10 MB, the API accepts larger but processing time grows.
Page size on list endpoints	`size` ≤ 1000
ERP call timeout	AIOTIC waits up to 30 s (tenant-configurable) for your receive endpoint.

CONCEPTS

Authentication & keys

Three credentials play a role. Two you receive from AIOTIC, one you create for AIOTIC. For a standard integration the **sync key** is all you need on the AIOTIC side.



Which key opens which door.

Calls *to* AIOTIC: `X-API-Key`

Every data-plane request carries the key in a header:

```
GET /order_status/list HTTP/1.1
Host: acme.aiotic.ai
X-API-Key: <key>
```

Key	Accepted on	Who gets it
Sync key	`/customer/*`, `/product/*`, `/customer-product/*` only	every integration: master-data sync job, ERP-side scripts
Integration key	every endpoint	issued per project by the AIOTIC team for headless integrations and custom review UIs

Most integrations never need the integration key: master data goes in with the sync key, and orders come out through the receive endpoint that AIOTIC calls on your side. Ask your AIOTIC contact for an integration key only when you build one of the headless models.

A missing or wrong key returns `401`:

```
{ "detail": "Invalid or missing API key" }
```

Two endpoints need no key: `GET /healthcheck` and `GET /system-status`.

> **Why two keys** The sync job often runs close to the ERP (a scheduled task, a stored procedure calling a script). Giving it the *sync key* means a leak there cannot read or send orders. The SDK routes master-data calls to the sync key automatically when `AIOTIC_SYNC_API_KEY` is set.

Calls *from* AIOTIC: `X-API-KEY`

When AIOTIC calls **your** endpoints (the ERP receive endpoint and the optional processing webhook) it sends the key **you** provided during onboarding:

```
POST /aiotic/orders HTTP/1.1
Host: integration.example.com
Content-Type: application/json
X-API-KEY: <the key you gave AIOTIC>
```

Verify it on every call, with a constant-time comparison, and reject with `401` otherwise. There is no signature or timestamp today (a signed scheme is proposed); transport security is TLS.

Tenant configuration

The tenant's ERP receive endpoint (URL, key, timeout) and the processing webhook are configured in the AIOTIC app by a tenant admin, or by the AIOTIC team during onboarding. This configuration is not reachable with API keys; hand the URL and key to your AIOTIC contact or enter them in the app's settings screen.

Handling keys safely

- Keep keys in a secret store or environment variables — never in source code, never in a browser or mobile app.
- Use one key per service. Rotate on staff changes; the sync key is admin-rotatable, the integration key is rotated by the AIOTIC team on request.
- Log the *fact* that a request was rejected, never the key value.
- In the SDK, `Settings.from_env()` reads `AIOTIC_API_KEY`, `AIOTIC_SYNC_API_KEY`, `AIOTIC_ERP_RECEIVE_KEY` and `AIOTIC_WEBHOOK_KEY`.

Example

```
<span class="tab">curl</span>
curl -s https://acme.aiotic.ai/customer/list?size=5 -H "X-API-Key: $AIOTIC_SYNC_API_KEY"
```

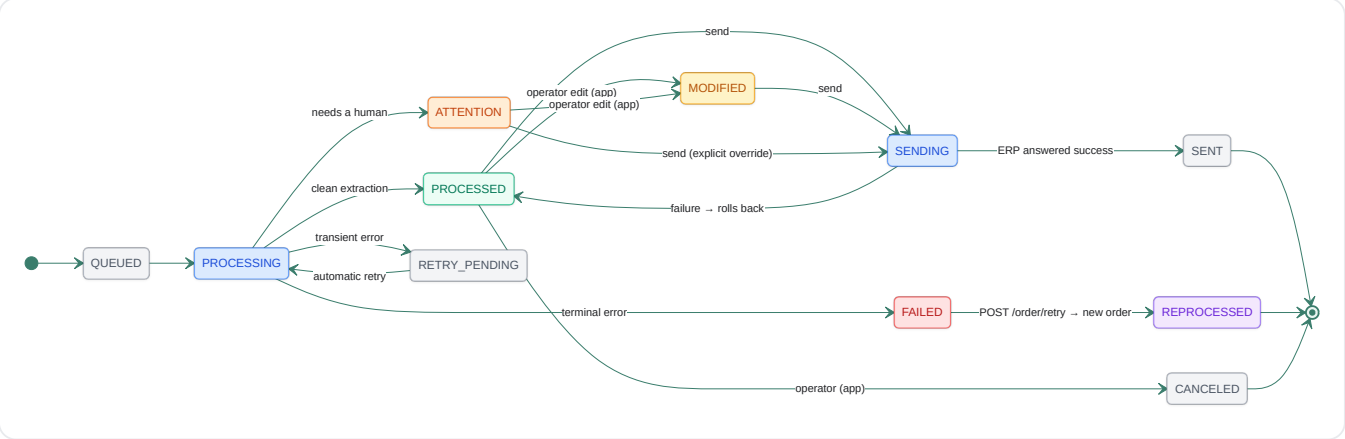
```
<span class="tab">Python SDK</span>
from aiotic import AioticClient

client = AioticClient(base_url="https://acme.aiotic.ai", api_key="…integration…", sync_api_key="…sync…")
client.customers.list(size=5)      # sent with the sync key
client.orders.list(size=5)         # sent with the integration key
```

CONCEPTS

Order lifecycle & statuses

Every order is identified by a `request_id` and moves through a small set of statuses. You read them from `GET /order_status/{request_id}` and `GET /order_status/list`.



Solid arrows happen automatically or through the API; dashed arrows involve an operator or a retry.

Status reference

Status	Meaning	What you do
`QUEUED`	Accepted, waiting to be processed.	Wait.
`PROCESSING`	Extraction, validation and resolution are running (seconds to a few minutes).	Wait.
`PROCESSED`	Clean extraction: every line resolved to a known article, customer identified, totals consistent.	Sendable. Review or auto-send per your model.
`ATTENTION`	Landed, but something needs a human: an unknown article, an unidentified customer, an unreadable quantity, a mismatch between document and master data. `result` is present.	Route to a person. Sendable only as an explicit override.
`FAILED`	Terminal processing error (`last_error` says why).	`POST /order/retry/{id}` once; then alert.
`RETRY_PENDING`	Transient failure (temporary upstream outage, timeout); AIOTIC will retry automatically (`next_retry_at` , `retry_count`).	Wait.
`MODIFIED`	An operator edited fields in the AIOTIC app.	Sendable.
`SENDING`	Locked while AIOTIC calls your ERP receive endpoint.	Wait; a concurrent send returns `409` .
`SENT`	Your ERP accepted the order; `erp_ref` holds your order number. Final.	Nothing — the order is in your ERP.
`REPROCESSED`	This order was retried; a **new** order with a new `request_id` continues the work. Final for this id.	Track the new id.
`CANCELED`	Cancelled by an operator in the app. Final.	Nothing.

Sendable = `PROCESSED` , `MODIFIED` , `ATTENTION` . **Landed** (safe to read `result`) = `PROCESSED` , `ATTENTION` , `MODIFIED` , plus `FAILED` (no result).

Rules that never change

- A send is never implicit. Parsing, editing or polling never triggers a call to your ERP.
- `SENT` cannot be sent again. A retry of a *send* only happens after a failure rolled the status back.

- A retry of *processing* creates a new order. The old id becomes `REPROCESSED` and keeps its history.
- Corrections are non-destructive: the AI's original reading is kept next to the operator's edits; you receive the merged result.
- `ATTENTION` is a normal outcome, not an error. The product prefers flagging over guessing.

Reading the payload

`result` holds the purchase-order model. Fields that tell you why an order is in `ATTENTION`:

- `result.items[].article_number` is `null` → the line could not be resolved to your catalog.
- `result.customer.customer_id` is `null` → the sender could not be matched to a customer record.
- `result.items[].quantity_state == "Unrecognised"` → a quantity was present but unreadable; `quantity` is `null`.
- `state` (free-form) carries validation details used by the app.

Timing expectations

Step	Typical
Upload → `PROCESSING`	seconds
`PROCESSING` → landed	20 s – 3 min depending on page count
`SENDING` → `SENT`	as fast as your endpoint answers (timeout 30 s)

Poll with backoff (2 s, 3 s, 5 s, … capped at 15 s). The SDK's `orders.wait()` does this.

CONCEPTS

The purchase-order model

Two closely related shapes exist:

- **`PurchaseOrder`** — what you read in `OrderStatus.result` (the stored extraction, with AIOTIC's additive fields).
- **`ErpPurchaseOrder`** — what your ERP receive endpoint gets (a fixed subset, with operator corrections applied).

Both follow the same rules: **every key is always present**, optional values are `null` (never omitted), dates are ISO `YYYY-MM-DD`, amounts are decimal numbers, quantities are whole numbers.

`PurchaseOrder` (in `result`)


```
{
  "order_number": "EB2500011645",
  "order_date": "2026-01-14",
  "delivery_date": "2026-02-01",
  "delivery_date_from": null,
  "delivery_date_to": null,
  "supplier": { "company": "Acme Supplies BV", "contact_person": null, "email": "orders@acme.example",
                "address": { "street": "Industrieweg 5", "postal_code": "1234 AB", "city": "Amsterdam",
"country": "NL" } },
  "customer": { "customer_id": "58931", "company": "LUMITECH INSTALLATIES", "contact_person": "J. de
Boer",
                "email": "info@lumitech.example", "phone": "+31 55 123 4567", "branch": null,
                "vat_id": "NL001234567B01", "iban": null, "bic": null,
                "address": { "street": "Ambachtsweg 12", "postal_code": "7327 AA", "city": "Apeldoorn",
"country": "NL" } },
  "shipping_details": { "recipient": { "company": "LUMITECH INSTALLATIES", "contact_person": "J. de
Boer", "department": null,
                        "email": null, "phone": null,
                        "address": { "street": "Ambachtsweg 12", "postal_code": "7327 AA", "city":
"Apeldoorn", "country": "NL" } },
                        "special_instructions": null },
  "items": [
    { "article_number": "PROD-001", "customer_item_number": "LT-ART-001", "description": "LED Driver
48V",
      "quantity": 10, "quantity_state": "Valid", "unit": "ST", "price": 12.34, "currency": "EUR",
"line_total": 123.40 }
  ],
  "total_price": 123.40,
  "currency": "EUR",
  "additional_information": null
}
```


Header

Field	Type	Notes
`order_number`	string	The customer's PO number. `/` is replaced by `-` (many ERPs cannot store it).
`order_date`	string	ISO date where the document allowed it.
`delivery_date`	string | null	The single date your ERP consumes. When the document states a *window*, AIOTIC collapses it to one end according to the tenant's preference (earliest by default).
`delivery_date_from` / `delivery_date_to`	string | null	The window bounds, for audit. `null` for single-date orders.
`currency`	string | null	ISO code as printed (`EUR`).

Field	Type	Notes
`total_price`	number | null	The printed document total — **may include VAT**. When e-mail instructions changed the lines, this is the recalculated net sum and the original total is noted in `additional_information`.
`additional_information`	string | null	Free text from document and e-mail, plus notes AIOTIC appends.

Supplier

`supplier` is **you**. It is pinned from tenant configuration and identical on every order; it is not extracted from the document. Do not map it — it is there so the payload is self-describing.

Customer

Field	Notes
`customer_id`	Your customer (debtor) number, exactly as you synced it via `PUT /customer/{number}`. `null` when AIOTIC could not identify the sender with confidence — the order is then in `ATTENTION`.
`company`, `address`, `email`, `phone`, `vat_id`	When a customer was identified with high confidence, these are **canonicalised from your master record**; otherwise they are as read from the document.
`branch`	The issuing branch/location named on the document (chains with one debtor per branch).
`iban`, `bic`	Rarely present on purchase orders; passed through when found.

Shipping details

`shipping_details.recipient` is the ship-to block. When the document has no explicit one, AIOTIC fills it from the identified customer. `special_instructions` carries delivery remarks ("deliver before noon").

Line items

Field	Notes
`article_number`	**Your** SKU, after resolution through the catalog and customer item mappings (how). `null` = unresolved → `ATTENTION`.
`customer_item_number`	The customer's own code as printed, when present. Useful to create a mapping afterwards.
`quantity`	Whole units. `null` only when `quantity_state` is `Unrecognised`. Lines with a blank or zero quantity (assortment listings) are dropped before storage.
`quantity_state`	`Valid`, `Unrecognised` (kept for review), or absent on older orders.
`unit`	As printed (`ST`, `PCS`, `Stk`, `m`, `KG` …). Map it on your side (SDK sanitizer).
`price`, `line_total`	Unit price and line amount as printed, when present.

`ErpPurchaseOrder` (what your endpoint receives)

The hand-off payload is a **fixed allow-list** of the above:

- header: `order_number`, `order_date`, `delivery_date`, `currency`, `total_price`, `additional_information`, `supplier`
- `customer`: `customer_id`, `company`, `contact_person`, `email`, `phone`, `iban`, `bic`, `vat_id`, `address{street, postal_code, city, country}`
- `shipping_details.recipient`: `company`, `department`, `contact_person`, `email`, `phone`, `address{…}` and `special_instructions`
- `items[]`: `article_number`, `description`, `quantity`, `unit`, `price`, `currency`, `line_total`

Not included: `quantity_state`, `customer_item_number`, `branch`, `delivery_date_from/to`. Operator edits from the AIOTIC app are merged in before sending. The full JSON Schema is in the API reference.

Working with it in Python

```
from aiotic.models import ErpReceiveRequest, PurchaseOrder

req = ErpReceiveRequest.model_validate_json(body)      # in your receive endpoint
for line in req.purchase_order.items:
    ...

status = client.orders.get(request_id)                  # when polling
po: PurchaseOrder | None = status.result
if po and po.unresolved_items:
    ...
```

CONCEPTS

Reference data

AIOTIC needs three tables from you. They are not "nice to have": they are what turns a document into a bookable order.

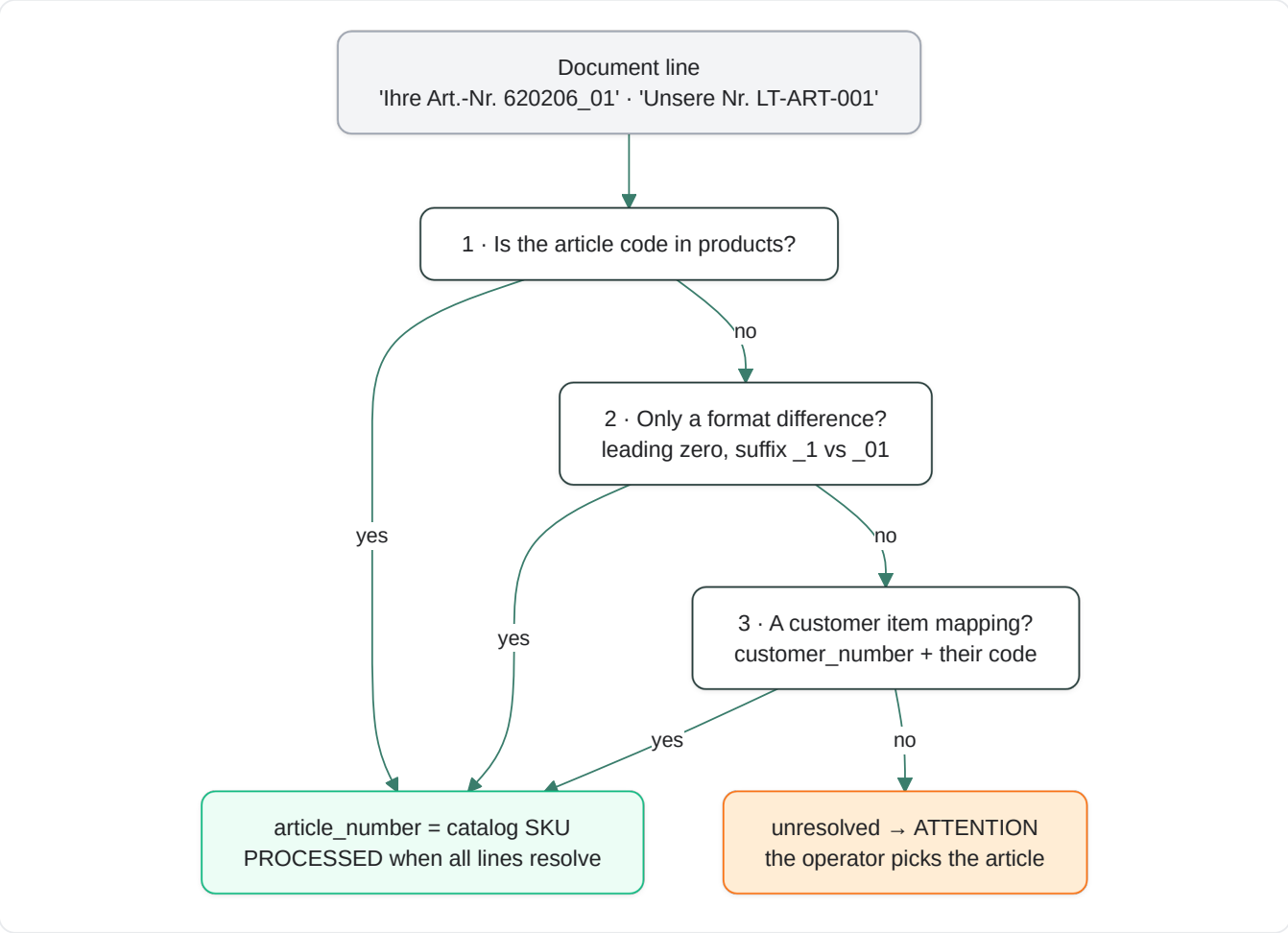
Table	Key	Used for
Customers (`/customer`)	your debtor number	Identifying *who* sent the order → `customer.customer_id`
Products (`/product`)	article number + language code	Deciding whether an article number is *valid* — the list of everything your ERP will accept
Customer item mappings (`/customer-product`)	debtor number + the customer's own article code	Translating a customer's code to your article number

Why AIOTIC synchronises your data instead of querying your ERP live

Integrators regularly ask why AIOTIC keeps its own copy of customers, products and mappings rather than looking them up in the ERP at the moment an order is processed. It is a deliberate design choice, for five reasons.

1. **Performance you can rely on.** Extraction, identification and article resolution run against data that sits next to the AI processing, on AIOTIC's own infrastructure. That is fast, it stays fast under load, and it can be optimised end to end. A live lookup would make every order as slow, and as available, as the ERP's API on that particular day.
2. **Normalisation happens at sync time.** Master data in most ERPs is not standardised: inconsistent postal codes, VAT numbers with and without spaces, article numbers in three formats, names with legacy markers. When a record is synchronised, AIOTIC normalises it into a canonical form. Documents are matched against that clean form, not against whatever the ERP happens to contain.
3. **Independence from the ERP.** AIOTIC does not depend on the vendor, version, API style or availability of your business software. Every tenant's data is organised the same way, which is what lets one product deliver consistent results across very different ERPs. Your side of the integration stays small: push changes, receive orders.
4. **Enrichment for context.** During synchronisation AIOTIC derives additional context from your records — search indexes, alternative identifiers, relations between customers, branches and article codes — so that the processing has as much evidence as possible for every decision. That enrichment is what turns a bare ERP export into an ontology the AI can be held to.
5. **Continuous benchmarking per tenant.** Because the synchronised data is validated, enriched and organised, AIOTIC can measure results against it and build tenant-specific improvement data: which customers, article formats and document layouts your organisation actually deals with. Accuracy therefore improves for *your* tenant over time, based on *your* data.

The trade-off is that your master data must be kept current, which is why this guide spends several pages on doing that efficiently (event-driven, never full dumps). A live connection would trade that small, one-time integration effort for slower processing, ERP-dependent behaviour and none of the normalisation and enrichment above.



Where each line's article number comes from, and why a missing product record ends in ATTENTION.

Customers

One record per debtor you can book an order for. The more fields you fill, the better the identification: company name, street, postal code, city, VAT number, e-mail, phone, Chamber of Commerce number and website all count as evidence.

```
PUT /customer/58931
{
  "name": "LUMITECH INSTALLATIES",
  "address": "Ambachtsweg 12",
  "postal_code": "7327 AA",
  "city": "Apeldoorn",
  "vat_number": "NL001234567B01",
  "email": "info@lumitech.example",
  "phone_number": "+31 55 123 4567",
  "contact_person": "J. de Boer",
  "coc_number": "12345678",
  "home_page": "https://www.lumitech.example"
}
```

Things AIOTIC does with this record that you should know:

- **Identification is evidence-based and multi-stage.** Hard identifiers printed on the order (a customer number, a VAT number, a known e-mail address) weigh more than softer evidence such as address similarity or the customer's own article codes. A confident match sets `customer_id`; an ambiguous one leaves it `null` and flags the order for a person.

- **Multi-branch customers.** If you keep one debtor per branch (same name, same VAT, different town), AIOTIC can still tell them apart when the document names the branch. Keep `city` accurate per branch.
- **Archiving by name.** A record whose name carries a legacy marker (`formerly`, `voorheen`, `(old)`, `do not use`, `closed`, `failliet`, …) is archived automatically: it is never *chosen*, but explicit references to it still resolve for audit. See Archiving.
- **"See customer N" markers.** A name like `*** ZIE 12306 ***` makes the record redirect to debtor 12306. See Archiving.
- On a confident match, the customer block in the payload is **canonicalised from your record** — which is why stale addresses in AIOTIC show up in orders.

Products

One record per **article number per language**. The description is what the product is called in that language; `remark` is free.

```
PUT /product/PROD-001/nl
{ "description": "LED Driver 48V 100W", "remark": "Incl. montagebeugel" }
```

- The product table is **the definition of a valid article number**. After extraction, every line's `article_number` must exist here; otherwise the order goes to `ATTENTION`. A missing product record is the single most common cause of `ATTENTION`.
- AIOTIC tolerates *format* differences deterministically: leading zeros (`620206_01` vs `0620206_01`), suffix padding (`_1` vs `_01`), and separator variants (`.` / `-` vs `_`) are matched to the canonical SKU when the match is unique. It never invents a SKU.
- Language codes are yours to choose (`nl`, `de`, `en`…). If you only maintain one language, use one code consistently and pass the same code in mappings.

Customer item mappings

Many customers print *their* article numbers on purchase orders. A mapping says: when **customer 58931** writes **LT-ART-001**, they mean **PROD-001**.

```
PUT /customer-product/58931/LT-ART-001
{ "item_number": "PROD-001", "language_code": "nl" }
```

Both the customer and the product must already exist (`404` otherwise). Mappings also act as identification evidence: three or more of a customer's codes on one document strongly suggest who sent it.

How article numbers are resolved

For each line AIOTIC works through a fixed series of stages and stops at the first hit:

1. the article code printed on the document, if it exists in **products**;
2. a format-only variant of it (leading zero, suffix padding, separator) that maps to exactly one product;
3. the customer's own code through **customer item mappings** (needs an identified customer).

Anything still unresolved leaves `article_number` `null` and the order in `ATTENTION`. When operators fix such a line in the AIOTIC app, consider adding the mapping so the next order resolves automatically — the `customer_item_number` on the line tells you what to map.

Sizing

Data	Typical size	Sync approach
Customers	1 000 – 50 000	event-driven, nightly reconcile
Products	5 000 – 500 000	**event-driven only**; reconcile weekly or on demand
Mappings	0 – 100 000	event-driven; grow from operator corrections

See Keeping data in sync.

CONCEPTS

Errors, idempotency & limits

Error shape

Every error is JSON with a `detail` member. Usually a string:

```
{ "detail": "Document with ID 550e8400-e29b-41d4-a716-446655440000 not found" }
```

A few endpoints return a **structured** `detail` so you can branch on a stable code instead of parsing text. Today that is `POST /order/raw/upload` when the e-mail is not a purchase order:

```
{ "detail": { "error": "not_a_purchase_order", "message": "…", "request_id": "…", "category":
"quotation",
              "subject": "Offer 10531", "language": "de", "line_item_count": 25 } }
```

Write your client so that `detail` may be a string **or** an object. FastAPI-style validation errors (`422`) carry a list under `detail` .

Status codes

Code	Meaning	Typical cause
`200` / `202` / `204`	OK / upsert accepted / deleted	
`400`	Bad request	malformed UUID, unsupported file, order not in `FAILED` for a retry, e-mail not a PO
`401`	Unauthorized	missing/invalid key, or the sync key on a non-master-data path
`404`	Not found	unknown order / customer / product / mapping; mapping references a missing customer or product
`409`	Conflict	order not sendable or already `SENDING` ; rejected e-mail already overridden
`415`	Unsupported media type	file extension not accepted
`422`	Unprocessable	your ERP answered `success: false` on a send (`detail` = your error text); validation error
`500`	Server error	your ERP returned a non-JSON body or timed out during a send; unexpected failure
`503`	Unavailable	tenant not initialised, classification or ERP integration not configured

Retrying safely

Operation	Safe to retry?	How
`POST /order/upload`	Yes, **if** you pass your own `request_id`	Same UUID → same order; without it a retry creates a duplicate order
`PUT` master data	Yes	Upserts are idempotent
`DELETE` master data	Yes	A second delete returns `404` — treat as success
`POST /erp/send/{id}`	Yes	AIOTIC locks the order; a concurrent send gets `409` ; a failed send rolls back so you can send again
`POST /order/retry/{id}`	No (creates a new order each time it succeeds)	Check the status first
`GET` anything	Yes	

Retry on `408`, `429`, `502`, `503`, `504` and on connection errors with exponential backoff and jitter (the SDK: 0.5 s · 2^attempt, capped at 30 s, three attempts). Do **not** retry a `503` on `/erp/send` blindly — it means the tenant has no ERP endpoint configured.

Idempotency on your side

Your ERP receive endpoint is called with a `request_id` that is stable across retries. Store `request_id → your order number` before you answer, and answer the same thing again if you see the id twice. Details in Idempotency & failure handling.

Correlating orders

- Put your own reference in the upload as extra form fields; it comes back in `metadata` on every status read.
- Use `request_id` in your ERP as the external reference. It appears in the AIOTIC app, in support tickets and in the hand-off payload.

Limits

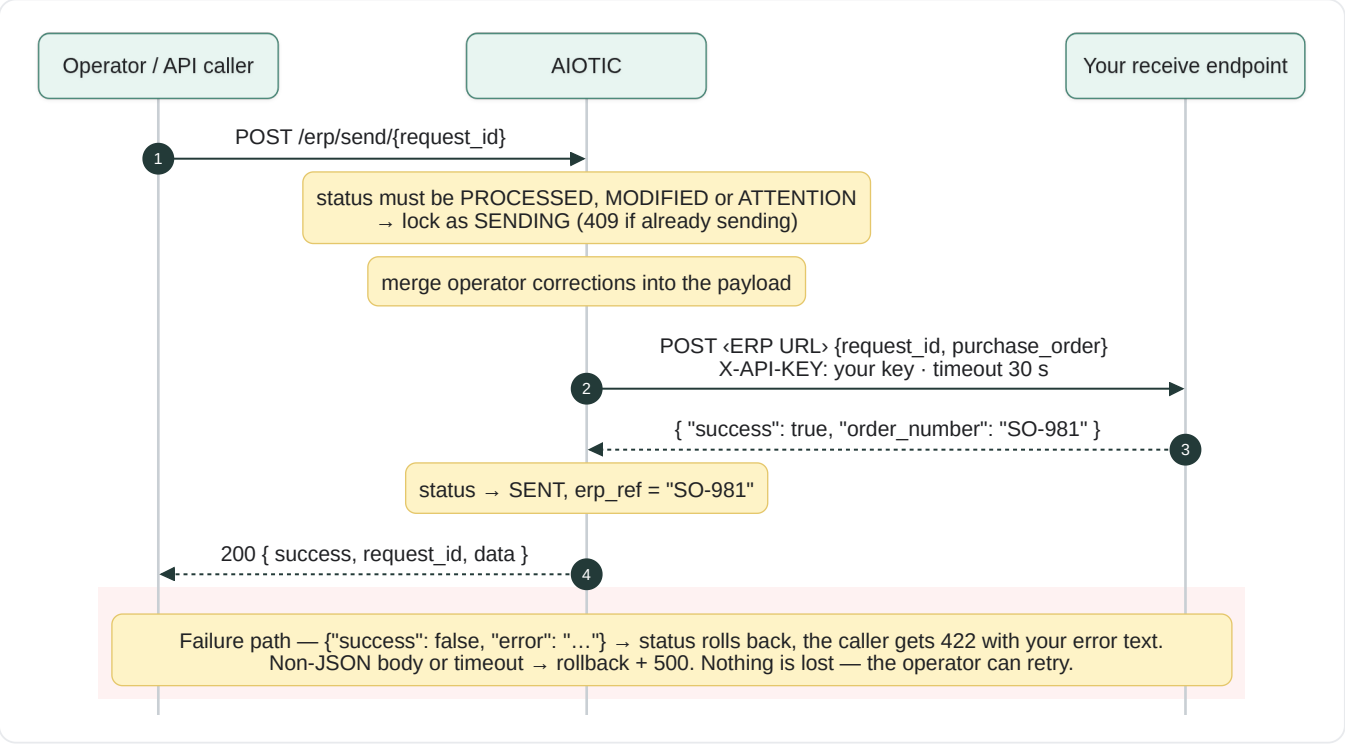
Limit	Value
`size` on list endpoints	1 … 1000
`top_k` on customer search	1 … 1000
Files per upload	any; all files form **one** order
Accepted upload types	`.pdf .jpg .jpeg .png .txt .md` (+ `.eml` on the raw endpoint)
ERP receive timeout	30 s default
Server-side rate limit	none — self-limit (SDK default 10 req/s)

RECEIVING ORDERS

The ERP receive endpoint

For: ERP developers, integration partners

This is the one thing every integration must implement: an HTTPS endpoint that AIOTIC calls with a reviewed purchase order, and that answers whether your ERP accepted it — normally by creating a sales order and returning its number.



Triggered by an operator in the app or by POST /erp/send/{request_id}. AIOTIC waits for your answer.

The request

```
POST https://integration.example.com/aiotic/orders HTTP/1.1
Content-Type: application/json
X-API-KEY: <the key you gave AIOTIC>

{
  "request_id": "550e8400-e29b-41d4-a716-446655440000",
  "purchase_order": {
    "order_number": "EB2500011645",
    "order_date": "2026-01-14",
    "delivery_date": "2026-02-01",
    "currency": "EUR",
    "total_price": 1234.56,
    "additional_information": "Please deliver before noon.",
    "supplier": { "company": "Acme Supplies BV", "contact_person": "P. Jansen", "email":
"orders@acme.example",
                  "address": { "street": "Industrieweg 5", "postal_code": "1234 AB", "city": "Amsterdam",
"country": "NL" } },
    "customer": { "customer_id": "58931", "company": "LUMITECH INSTALLATIES", "contact_person": "J. de
Boer",
                  "email": "info@lumitech.example", "phone": "+31 55 123 4567", "vat_id":
"NL001234567B01", "iban": null, "bic": null,
                  "address": { "street": "Ambachtsweg 12", "postal_code": "7327 AA", "city": "Apeldoorn",
"country": "NL" } },
    "shipping_details": { "recipient": { "company": "LUMITECH INSTALLATIES", "department": null,
"contact_person": "J. de Boer",
                          "email": null, "phone": null,
                          "address": { "street": "Ambachtsweg 12", "postal_code": "7327 AA", "city":
"Apeldoorn", "country": "NL" } },
                          "special_instructions": null },
    "items": [
      { "article_number": "PROD-001", "description": "LED Driver 48V", "quantity": 10, "unit": "ST",
        "price": 12.34, "currency": "EUR", "line_total": 123.40 }
    ]
  }
}
```

Part	Notes
`X-API-KEY` header	Verify on every call (constant-time compare). Reject with `401` otherwise.
`request_id`	AIOTIC's order id. **Stable across retries** — the key for idempotency. Store it on your sales order as external reference.
`purchase_order`	The `ErpPurchaseOrder`: reviewed values with operator corrections applied. Every key present; unknown values `null`.
`purchase_order.customer.customer_id`	Your debtor number. Can be `null` only if an operator explicitly sent an unidentified order — decide whether you accept that.
`purchase_order.items[].article_number`	Your SKU. Can be `null` only on an explicit override send of an `ATTENTION` order.

The complete field-by-field reference of this payload — every JSON path with type, nullability and meaning — is in the API reference → Outbound: webhooks, and the JSON Schema is in the OpenAPI document under `webhooks.erpReceiveOrder`.

The response

Answer with JSON. **The body decides; the HTTP status does not.**

```
<span class="tab">Accepted</span>
{ "success": true, "order_number": "SO-2026-00981" }
```

```
<span class="tab">Business rejection</span>
{ "success": false, "error": "Unknown article number: PROD-999" }
```

Details in Response contract.

Minimal implementations

```
<span class="tab">Python SDK</span>
from aiotic.service import build_app
from aiotic.erp.memory import InMemoryErp      # → your adapter, see /sdk/erp-adapters

app = build_app(erp=InMemoryErp())             # POST /aiotic/orders with key check, idempotency,
pipeline
```

```
<span class="tab">Python, bare FastAPI</span>
import hmac, os
from fastapi import FastAPI, Header, Request

app = FastAPI()
KEY = os.environ["AIOTIC_ERP_RECEIVE_KEY"]

@app.post("/aiotic/orders")
async def receive(request: Request, x_api_key: str = Header(alias="X-API-KEY")):
    if not hmac.compare_digest(x_api_key, KEY):
        return {"success": False, "error": "unauthorized"}
    body = await request.json()
    rid, po = body["request_id"], body["purchase_order"]
    existing = erp.find_by_external_ref(rid)          # idempotency
    if existing:
        return {"success": True, "order_number": existing}
    try:
        number = erp.create_sales_order(external_ref=rid, **map_order(po))
    except BusinessError as e:                        # unknown item, blocked customer, …
        return {"success": False, "error": str(e)}
    return {"success": True, "order_number": number}
```

```
<span class="tab">C# (ASP.NET minimal API)</span>
app.MapPost("/aiotic/orders", async (HttpRequest http, ErpService erp) =>
{
    if (!CryptographicOperations.FixedTimeEquals(
            Encoding.UTF8.GetBytes(http.Headers["X-API-KEY"].ToString()),
            Encoding.UTF8.GetBytes(config.ReceiveKey)))
        return Results.Json(new { success = false, error = "unauthorized" }, statusCode: 401);

    var body = await http.ReadFromJsonAsync<ErpReceiveRequest>();
    var existing = await erp.FindByExternalRef(body.RequestId);
    if (existing != null) return Results.Json(new { success = true, order_number = existing });

    try {
        var number = await erp.CreateSalesOrder(body.RequestId, body.PurchaseOrder);
        return Results.Json(new { success = true, order_number = number });
    } catch (BusinessRuleException e) {
        return Results.Json(new { success = false, error = e.Message });
    }
});
```

Checklist for this endpoint

- [ ] HTTPS with a valid certificate; reachable from the internet (or allow-listed for the tenant's egress IP)
- [ ] `X-API-KEY` verified, constant-time
- [ ] Idempotent on `request_id` (same id → same `order_number`, no second booking)
- [ ] Answers within a few seconds; heavy work (PDF archiving, e-mails) happens after the response
- [ ] Business problems → `success: false` with a human-readable `error`; infrastructure problems → any status, ideally also `success: false`
- [ ] Never returns HTML (a reverse-proxy error page is treated as a server error and the send rolls back)
- [ ] Logs the `request_id` with every line so support can correlate

Next: Response contract →

RECEIVING ORDERS

Response contract

AIOTIC reads exactly one thing from your answer: the `success` field.

Field	Type	Required	Effect in AIOTIC
`success`	boolean	**yes**	`true` → order becomes `SENT`. `false` → status rolls back, operator sees `error`.
`order_number`	string	on success	Stored as `erp_ref`, shown in the app, returned to API callers in `data.order_number`. Use your real sales-order number.
`error`	string	on failure	Shown verbatim to the operator. Make it actionable: *"Unknown article number PROD-999 on line 3"* beats *"validation failed"*.

Anything else in the body is passed through untouched to `POST /erp/send` callers (`data`), so you may add diagnostic fields.

What AIOTIC does with each outcome

Your response	HTTP status	AIOTIC
`{"success": true, "order_number": "SO-1"}`	any 2xx (or even 5xx)	`SENT`, `erp_ref = "SO-1"`
`{"success": false, "error": "…"}`	any	rollback to previous status; API caller gets `422` with your text; operator sees your text
JSON without `success`	any	treated as **success** (legacy behaviour — always send the field)
Non-JSON (HTML error page, empty body)	any	rollback; API caller gets `500 "ERP server error"`; operator sees a generic message
No answer within the timeout (30 s default)	—	rollback; `500`; operator can retry

> **HTTP status is not the signal** Some ERPs answer business errors with `500`. AIOTIC deliberately ignores the status and trusts `success`. Do the same in your tests: an accepted order with a `500` is still `SENT`.

Choosing between rejection and failure

Situation	Answer
Unknown article, blocked customer, credit limit, missing mandatory field	`success: false` with the reason — the operator can fix it in AIOTIC and send again
Duplicate: same `request_id` seen before	`success: true` with the **same** `order_number` (idempotent replay)
Duplicate: different `request_id`, same customer PO number	Your policy. Usually `success: false` "already booked as SO-…" so a human decides
ERP down, database timeout	`success: false` with a generic message (or let it 5xx). Both roll back; the operator retries later

Timing

Respond fast. AIOTIC holds the order in `SENDING` and the operator waits on the button. If your ERP is slow to create orders, accept synchronously (validate, persist a queued record, return a provisional number) and finish asynchronously — as long as `success: true` really means "this will be booked".

Test cases you should have

1. valid order → `success: true` + number, order visible in ERP
2. same payload again → same number, nothing new in ERP
3. unknown article → `success: false`, clear text
4. wrong `X-API-KEY` → `401`, nothing written
5. malformed JSON → `4xx` with `success: false`
6. ERP unavailable → `success: false`, nothing half-written

RECEIVING ORDERS

Idempotency & failure handling

The receive endpoint sits between two systems that both retry. Get three things right and nothing gets booked twice or lost.

1. Idempotency on `request_id`

`request_id` is AIOTIC's identity for the order and it **never changes across retries** of a send. Your rule: *one `request_id` → at most one sales order.*

```
existing = erp.find_order_by_request_id(request_id)     # external reference column, indexed
if existing:
    return {"success": True, "order_number": existing.number}
```

Persist the link **in the same transaction** as the order (or before you answer). If you keep it in a side table, write it before returning `success: true`.

The SDK's `ErpReceiver` does this with an `IdempotencyStore` (in-memory, SQLite, or your own) **and** asks the adapter's `find_order_by_request_id` as a second line of defence.

2. Transactions

Create header and lines atomically. A half-written order that later answers `success: false` leaves debris in the ERP that no one knows about. With a data API this means one database transaction; with a functional API, one create call (or compensating delete on failure).

3. Failure paths, one by one

What went wrong	Right behaviour	Result in AIOTIC
Wrong key	`401`, `{"success": false, "error": "unauthorized"}`	rollback; operator sees error
Payload does not validate (your rules)	`{"success": false, "error": "<why>"}`	rollback; operator fixes and resends
ERP rejects for business reasons	`{"success": false, "error": "<ERP text>"}`	same
ERP unreachable / timeout	`{"success": false, "error": "ERP unavailable, retry later"}` — nothing written	rollback; retry later
Your endpoint crashes after the ERP created the order but before answering	AIOTIC times out and rolls back; the operator retries; **your idempotency lookup finds the order** and answers `success: true`	consistent
Your endpoint is down entirely	AIOTIC gets a connection error, rolls back	operator retries when you are back

The one case you must design for is the **crash after create**: the external reference must be written with the order, not afterwards.

4. Duplicates that are not retries

A customer can send the same PO twice; an operator can upload it twice. Those arrive with **different** `request_id`s. Detect by `(customer_id, order_number)` and decide:

- reject with `success: false` "PO EB2500011645 already booked as SO-981" (recommended — a person decides), or

- accept and create a new order if your business allows repeat POs with the same number.

The SDK's `NoDuplicateOrder` rule implements the first policy; the key is remembered only after a successful booking, so a rejected order can be corrected and sent again.

5. What to log

Per call: `request_id`, outcome, `order_number` or error text, duration. Never the key. Keep the payload for a while (it is small) — support questions come days later.

6. What to monitor

- rate of `success: false` per error text (a spike in "unknown article" means your product sync is behind)
- p95 response time (must stay well under 30 s)
- calls with an unknown key (someone is probing)

RECEIVING ORDERS

Field mapping cookbook

How the hand-off payload usually maps onto a sales order. Column names are illustrative; the *decisions* are what matter.

Header

AIOTIC	Sales order	Notes
`request_id`	external reference / your-ref field	Index it. Used for idempotency and support.
`purchase_order.order_number`	customer PO number ("Your reference")	Already `/`-free. Truncate to your field length — many ERPs allow 20–35 characters.
`order_date`	order date	ISO string → date.
`delivery_date`	requested delivery date	May be `null` → your default lead time.
`currency`	currency code	Validate against your allowed set; default to the customer's currency when `null`.
`total_price`	*do not book*	Informational. It is the printed total and may include VAT. Your ERP computes totals.
`additional_information`	internal note / order remark	Free text; can be long.
`supplier`	*ignore*	It is you.

Customer

AIOTIC	Sales order	Notes
`customer.customer_id`	sell-to customer number	Your debtor number as synced. `null` → reject (or your fallback).
`customer.company`, `address`, `vat_id`, `email`, `phone`	*compare, do not overwrite*	On confident matches these already equal your master record. A difference means the customer printed something new — worth a review, not an automatic master-data update.
`customer.contact_person`	contact / attention	Document-authoritative: the person who signed *this* PO.
`customer.iban`, `bic`	*usually ignore*	Purchase orders rarely carry bank data.

Ship-to

AIOTIC	Sales order	Notes
`shipping_details.recipient.company` + `address`	ship-to address	Match against the customer's known ship-to addresses if you keep them; create a one-off address otherwise.
`recipient.contact_person`, `phone`, `email`	ship-to contact	For the carrier.
`recipient.department`	address line 2	
`special_instructions`	shipment note	"Deliver before noon", dock numbers, etc.

Lines

AIOTIC	Sales order line	Notes
`items[].article_number`	item number	Your SKU. `null` only on override sends → reject or route to a person.
`items[].quantity`	quantity	Whole units in the **base unit** unless `unit` says otherwise. `null` only on override sends.
`items[].unit`	unit of measure	As printed: `ST`, `PCS`, `Stk`, `stuks`, `m`, `KG`, `rol` … Map to your codes; unknown → base unit or reject.
`items[].price`	unit price	**Customer-stated** price. Most suppliers book their own price list and keep the customer's price as reference; some honour it. Decide explicitly.
`items[].line_total`	*check only*	`quantity × price`; useful to detect a mis-read quantity.
`items[].description`	description	The customer's wording; keep in a note if you use your own catalog text.
`items[].currency`	line currency	Normally equals the header's.

Business Central flavour

If you target Microsoft Dynamics 365 Business Central through its API:

- `Sales Order` header: `customerNumber` ← `customer_id`, `externalDocumentNumber` ← `order_number` (35 chars), `requestedDeliveryDate` ← `delivery_date`, `currencyCode` ← `currency`.
- Ship-to fields: `shipToName`, `shipToAddressLine1`, `shipToPostCode`, `shipToCity`, `shipToCountry`.
- Lines: `lineType: Item`, `lineObjectNumber` ← `article_number`, `quantity`, `unitOfMeasureCode` ← mapped `unit`. Omit `unitPrice` to take the price list.
- Business Central returns `number` — that is your `order_number` in the response.
- BC rejects unknown items with a 400 whose `error.message` reads *"The Item does not exist. Identification fields and values: No.='…'"* — pass it through as `error`.

Data-API ERPs (writing to tables)

Everything above still applies, plus: nobody validates for you. Use the SDK pipeline with at least *catalog membership*, *customer exists / not blocked*, *positive quantities* and *duplicate PO* before writing. See Validation when your ERP has no functional API.

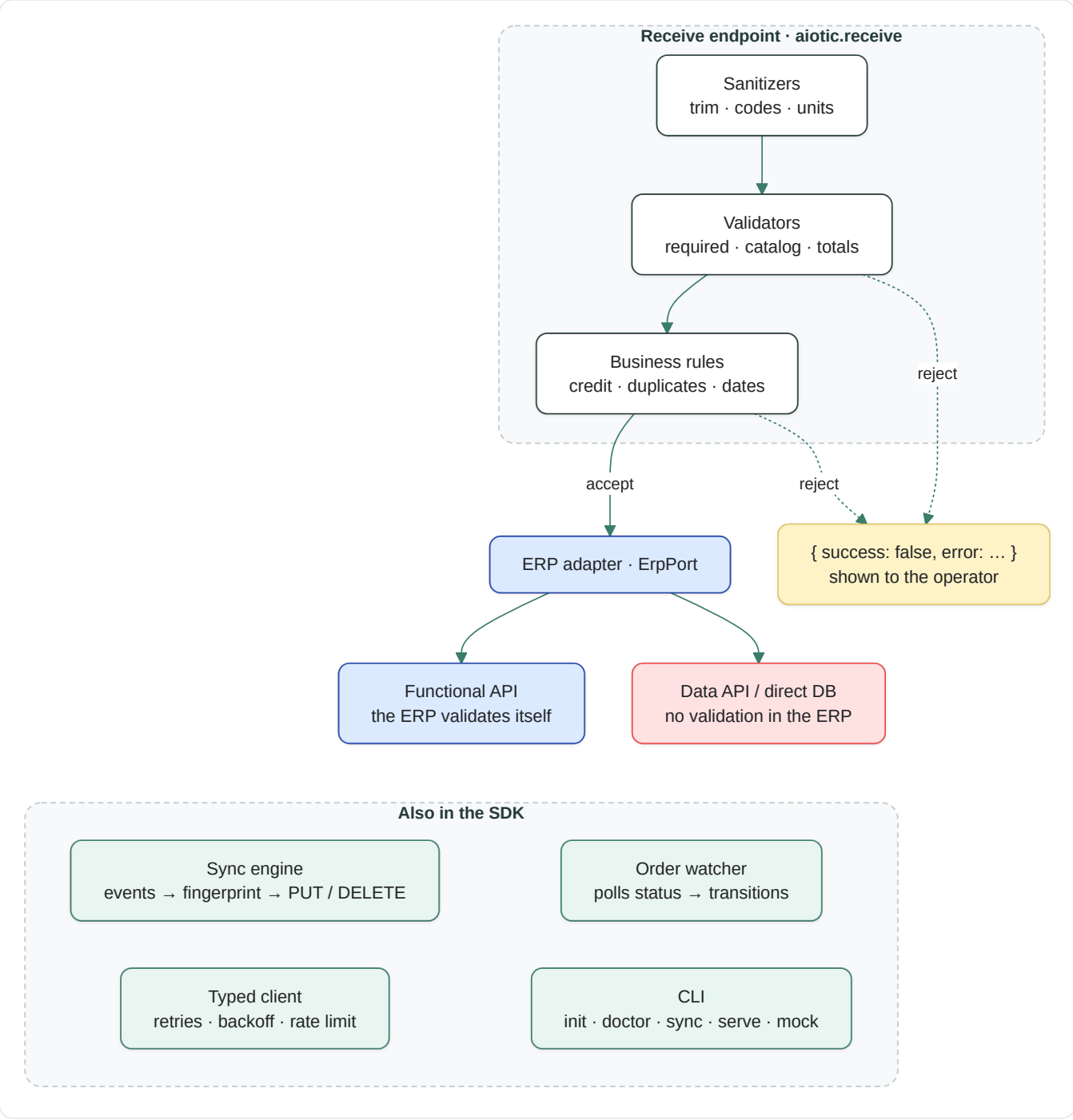
RECEIVING ORDERS

Validation when your ERP has no functional API

Two kinds of ERP show up in integrations:

	Functional API	Data API / direct database
How you create an order	`POST /salesOrders` or a "create order" function	`INSERT` into order tables (or a generic "data" API that does the same)
Who validates	the ERP: unknown item, blocked customer, credit limit, unit conversion…	**nobody** — whatever you write lands; the ERP trips over it later
Failure mode	immediate, explicit error you can pass to the operator	silent corruption discovered days later by someone else

With a data API, **the integration layer has to be the validation layer**. That is what the SDK's pipeline is for.



Sanitizers → validators → business rules → adapter. With a data-API ERP the pipeline is not optional.

What to validate, minimum set

Check	Why	SDK step
Article exists and is not blocked	the #1 way to create unbookable orders	`validators.ArticlesInCatalog(erp)`
Customer exists	orphan orders	`validators.CustomerResolved(erp)`
Customer not blocked / credit hold	policy	`rules.CustomerNotBlocked(erp)`
Quantities are positive integers	AIOTIC sends `null` for unreadable quantities on override sends	`validators.PositiveQuantities()`

Check	Why	SDK step
Unit of measure known	`stuks` vs `ST` vs `PCS`	`sanitizers.MapUnits({...})` (+ reject unknown)
Currency allowed	wrong currency = wrong price	`validators.CurrencyAllowed(["EUR"])`
Delivery date plausible	typo detection	`validators.DeliveryDateSane()`
Duplicate PO for the same customer	double bookings	`rules.NoDuplicateOrder(store)`
Ship-to address complete	undeliverable orders	`rules.ShipToAddressComplete()`
Line total vs quantity × price	mis-read quantities (`18.000` vs `18`)	`validators.LineTotalsConsistent()` (warning)

Putting it together

```
from aiotic.erp.data_api import DataApiAdapter
from aiotic.pipeline import Pipeline, sanitizers as S, validators as V, rules as R
from aiotic.receive import ErpReceiver, SqliteStore
import pyodbc

erp = DataApiAdapter(lambda: pyodbc.connect(DSN), paramstyle="qmark")   # implements ErpPort +
CatalogPort + CustomerPort
store = SqliteStore("integration.db")

pipeline = Pipeline(
    sanitizers=[S.StripWhitespace(), S.NormalizeCountryCodes(), S.NormalizeCurrency(),
S.MapUnits({"doos": "BOX"}), S.FillShippingFromCustomer()],
    validators=[V.RequiredFields(), V.CustomerResolved(erp), V.ArticlesInCatalog(erp),
V.PositiveQuantities(), V.CurrencyAllowed(["EUR"])],
    rules=[R.CustomerNotBlocked(erp), R.NoDuplicateOrder(store), R.ShipToAddressComplete()],
)
receiver = ErpReceiver(erp, api_key=RECEIVE_KEY, pipeline=pipeline, store=store)
```

A rejecting verdict becomes `{"success": false, "error": "Unknown article number: PROD-999 (items[2].article_number); Customer 10577 is blocked for new orders"}` — the operator reads it in the AIOTIC app, fixes the order (or the master data) and sends again.

Your own rules

Anything that would live in an ERP's "before insert" logic can be a rule:

```
from aiotic.pipeline import Issue
from aiotic.pipeline.rules import rule

@rule("no_weekend_delivery")
def no_weekend(order, ctx):
    from datetime import date
    if order.delivery_date and date.fromisoformat(order.delivery_date).weekday() >= 5:
        yield Issue("weekend_delivery", "Delivery date falls on a weekend", path="delivery_date")

pipeline.rules.append(no_weekend)
```

`ctx` carries the `request_id` and anything earlier steps stored (for example the ERP customer record from `CustomerResolved`), so rules can use ERP data without a second lookup.

Sanitize before you validate

Order matters. Sanitizers normalise what the document printed (`Nederland` → `NL` , `€` → `EUR` , `7327aa` → `7327 AA` , `stuks` → `ST`); validators then judge the cleaned order. Everything returns a *new* order — the payload AIOTIC sent is never mutated, so you can log both.

Where the same checks live with a functional API

Keep the pipeline anyway, lighter: sanitizers plus `RequiredFields` and `NoDuplicateOrder` . Let the ERP do catalog and customer checks and pass its error text through (`ErpRejected`). You get one code path for both ERP styles, which is the point.

RECEIVING ORDERS

Testing your endpoint

With curl (contract test)

```
curl -s -X POST https://integration.example.com/aiotic/orders \
  -H "Content-Type: application/json" -H "X-API-KEY: $RECEIVE_KEY" \
  -d @erp-receive-sample.json
# → {"success":true,"order_number":"SO-2026-00981"}

# idempotency: same payload again must return the same order_number
curl -s -X POST … -d @erp-receive-sample.json
# → {"success":true,"order_number":"SO-2026-00981"}

# wrong key
curl -s -o /dev/null -w "%{http_code}\n" -X POST … -H "X-API-KEY: wrong" -d @erp-receive-sample.json
# → 401
```

The sample file is the request example from the API reference — download it as `erp-receive-sample.json`.

With the mock tenant (flow test)

The mock performs a *real* send: `POST /erp/send/{id}` on the mock calls your endpoint with the documented headers and interprets your answer exactly like production.

```
aiotic mock &                                              # :8080
curl -X POST localhost:8080/_mock/config -H 'Content-Type: application/json' \
  -d '{"erp_url": "http://localhost:9000/aiotic/orders", "erp_key": "'"$RECEIVE_KEY"'"}'

echo x > PO-1234.pdf
aiotic orders upload PO-1234.pdf          # waits until PROCESSED
aiotic orders send <request_id>           # → your endpoint is called; erp_ref stored on success
aiotic orders get <request_id> | grep erp_ref
```

File-name tricks in the mock: `…attention.pdf` → an `ATTENTION` order with one unresolved line (tests your rejection path); `…fail.pdf` → `FAILED` (tests retry handling).

In pytest

```
from aiotic.erp.memory import InMemoryErp
from aiotic.receive import ErpReceiver, InMemoryStore
from aiotic.service import default_pipeline

def test_rejects_unknown_article(sample_payload):
    erp = InMemoryErp(products={"PROD-001"})
    receiver = ErpReceiver(erp, api_key="k", pipeline=default_pipeline(erp, InMemoryStore()))
    out = receiver.handle(sample_payload, api_key_header="k")
    assert out.response.success is False
    assert "Unknown article number" in out.response.error
```

`ErpReceiver.handle()` is framework-free, so you test the contract without HTTP.

From a real tenant

Once your URL and key are configured for the tenant, a tenant admin can run **Test ERP connection** in the AIOTIC app. It performs a connectivity check and reports the outcome. Then send one real order from the app and check `erp_ref`.

What "done" looks like

Scenario	Expected
valid order	`success: true`, order in ERP, `erp_ref` set in AIOTIC
replayed `request_id`	same `order_number`, one order in ERP
unknown article	`success: false`, text mentions the article, nothing in ERP
blocked customer	`success: false`, nothing in ERP
wrong key	`401`, nothing in ERP
ERP down	`success: false` (or timeout), nothing half-written, order back in its previous status

KEEPING DATA IN SYNC

Initial load

Before the first real order, AIOTIC needs your customers, your sellable articles and any customer item mappings you already know. This page is the one-time import; the next pages keep it current. If you are wondering why AIOTIC keeps a synchronised copy at all instead of querying your ERP live, see why AIOTIC synchronises your data.

Order of operations

A mapping must reference an existing customer and product (the API answers `404` otherwise), so load in this order:

1. **Customers** — `PUT /customer/{number}` for every active debtor
2. **Products** — `PUT /product/{item_number}/{language_code}` for every sellable article, per language
3. **Customer item mappings** — `PUT /customer-product/{customer_number}/{customer_item_number}`

Throughput

There is no batch endpoint today (one request per record) and no server-side rate limit. Upserts are cheap for products and mappings; customer upserts also update the identification index and take a little longer.

Data set	Requests	With 8 parallel requests
5 000 customers	5 000	~5 min
200 000 products	200 000	~1–2 h
20 000 mappings	20 000	~10 min

Run the initial load once, from a machine close to the tenant, with **4–8 concurrent requests**. More does not help and hurts other tenants.

With the CLI

Export CSV/JSON from your ERP and let the SDK do the rest (only changed records are sent, so re-running is cheap):

```
aiotic sync customers --from customers.csv
aiotic sync products  --from products.csv
aiotic sync mappings  --from mappings.csv
```

Column names follow the API field names:

```
<span class="tab">customers.csv</span>
number,name,address,postal_code,city,vat_number,email,phone_number,contact_person,coc_number,home_page
58931,LUMITECH INSTALLATIES,Ambachtsweg 12,7327 AA,Apeldoorn,NL001234567B01,info@lumitech.example,+31 55 123 4567,J. de Boer,,
```

```
<span class="tab">products.csv</span>
item_number,language_code,description,remark
PROD-001,nl,LED Driver 48V 100W,
620206_01,nl,Kabel 3x1.5 mm² (100 m),
```

```
<span class="tab">mappings.csv</span>
customer_number,customer_item_number,item_number,language_code
58931,LT-ART-001,PROD-001,nl
```

With the SDK

```
from aiotic import AioticClient
from aiotic.sync import ChangeEvent, HashStateStore, SyncEngine

client = AioticClient()                                    # AIOTIC_* env vars
engine = SyncEngine(client, state=HashStateStore("sync-state.db"), concurrency=6)

report = engine.apply_many(
    [ChangeEvent.customer_upsert(c.number, name=c.name, address=c.street, postal_code=c.zip, city=c.city,
                                 vat_number=c.vat, email=c.email, phone_number=c.phone) for c in
erp.customers()]
    + [ChangeEvent.product_upsert(p.sku, "nl", description=p.name) for p in erp.products()]
    + [ChangeEvent.mapping_upsert(m.debtor, m.their_code, item_number=m.sku, language_code="nl") for m in
erp.mappings()]
)
print(report)          # sent=… deleted=0 unchanged=… failed=0 in …s
```

`apply_many` sorts by kind (customers → products → mappings), runs each kind in parallel, records a fingerprint per record, and reports failures with the reason. Re-running after a failure only sends what is still missing.

With plain HTTP

```
curl -X PUT https://acme.aiotic.ai/customer/58931 -H "X-API-Key: $AIOTIC_SYNC_API_KEY" -H "Content-Type:
application/json" \
  -d '{"name":"LUMITECH INSTALLATIES","address":"Ambachtsweg 12","postal_code":"7327
AA","city":"Apeldoorn","vat_number":"NL001234567B01","email":"info@lumitech.example"}'
# 202 Accepted

curl -X PUT https://acme.aiotic.ai/product/PROD-001/nl -H "X-API-Key: $AIOTIC_SYNC_API_KEY" -H "Content-
Type: application/json" \
  -d '{"description":"LED Driver 48V 100W"}'

curl -X PUT https://acme.aiotic.ai/customer-product/58931/LT-ART-001 -H "X-API-Key: $AIOTIC_SYNC_API_KEY"
-H "Content-Type: application/json" \
  -d '{"item_number":"PROD-001","language_code":"nl"}'
```

What to include, what to leave out

Include	Leave out
Every debtor who can place an order, incl. branches	Prospects, suppliers, employees
Every sellable article, every language you sell in	Discontinued articles (delete them — see Archiving)
Known customer codes from previous orders / EDI	Guesses

Verify

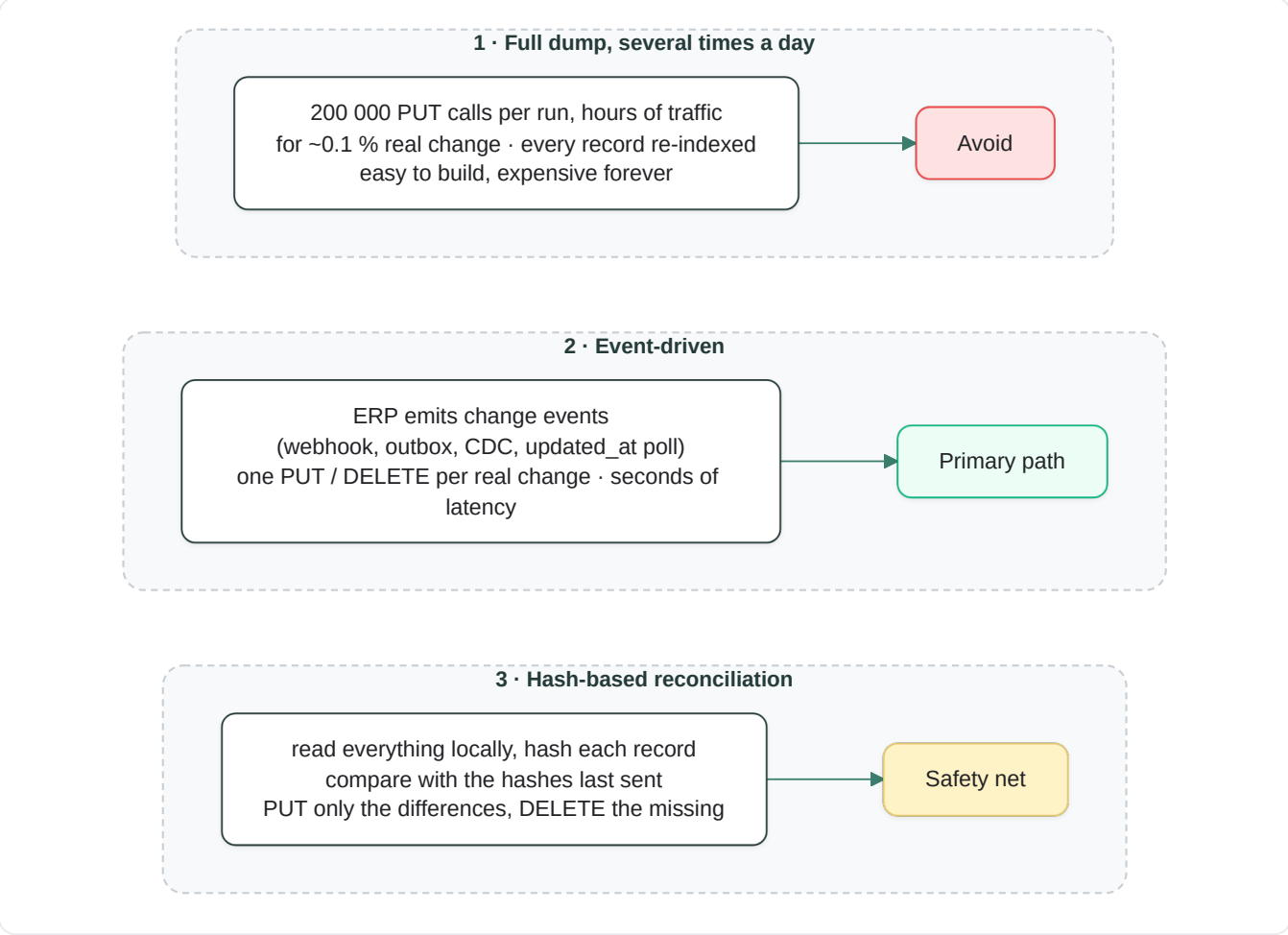
```
aiotic doctor                       # counts per table
curl -s "https://acme.aiotic.ai/customer-product/list?customer_number=58931" -H "X-API-Key: $KEY"
curl -s "https://acme.aiotic.ai/customer/search/lumitech%20apeldoorn?top_k=3" -H "X-API-Key: $KEY"
```

The search endpoint uses the same similarity AIOTIC uses for identification — a quick way to see whether a customer resolves from the text a document would contain.

KEEPING DATA IN SYNC

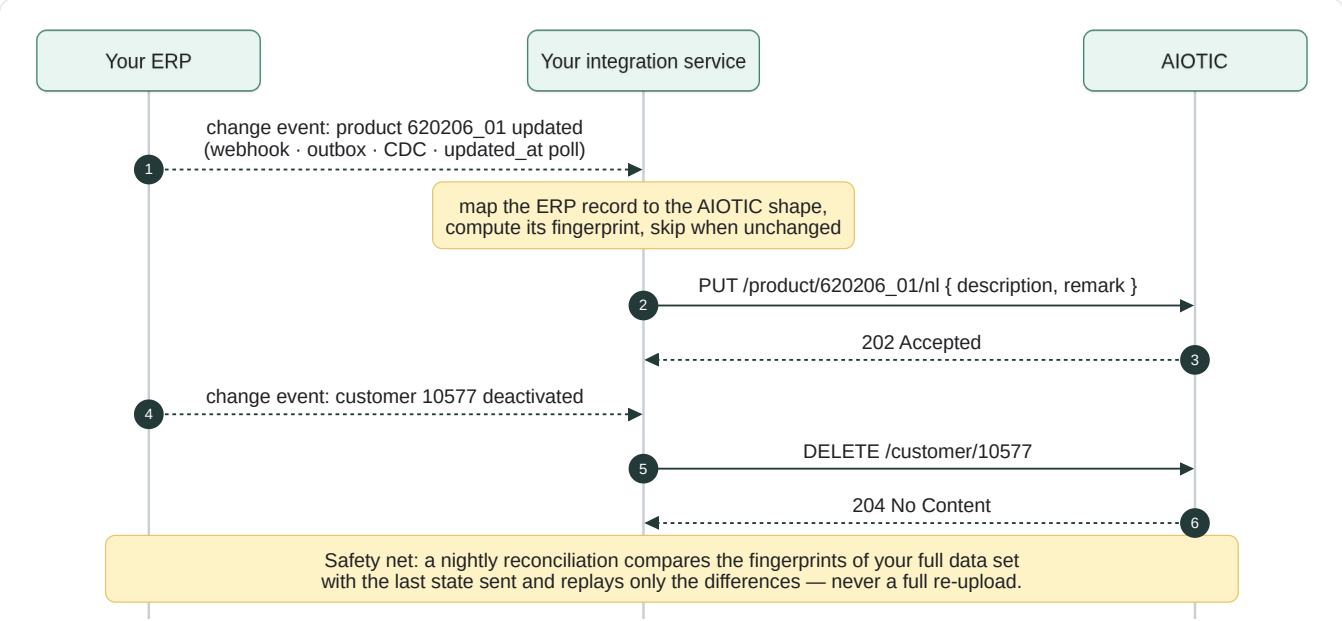
Event-driven sync

Do **not** re-upload your catalog four times a day. A 200 000-article catalog changes by a few dozen records per day; sending all of them costs hours of traffic and re-indexes every customer record for nothing. Send changes when they happen.



Full dumps are the expensive default everyone starts with. Events are the primary path; reconciliation is the safety net.

The pattern



One change in the ERP → one request to AIOTIC.

1. Your ERP produces a **change event** (customer/product/mapping created, updated, deactivated).
2. Your integration service maps it to the AIOTIC shape and computes a fingerprint.
3. If the fingerprint differs from the last one sent, `PUT` (or `DELETE`) the record. Otherwise skip.
4. Remember the fingerprint.

The SDK's `SyncEngine` implements 2–4; you implement 1 with whatever your ERP offers.

Where change events come from

ERP capability	How to hook it	Latency
Outbound webhooks (Business Central, Exact Online, Odoo, Shopify-style)	Subscribe to `customer.*`, `item.*` events; POST them to your service's `/erp/events`	seconds
Outbox table / event log in the ERP database	A small worker reads new rows and calls the sync engine	seconds–minutes
Change Data Capture (SQL Server CDC, Debezium)	Consume the change stream	seconds
`updated_at` columns	Poll `WHERE updated_at > :watermark` every few minutes	minutes
Nothing (legacy)	Hash-based reconciliation on a schedule	hours

Receiving events in your service

The SDK ships a generic event endpoint (`POST /erp/events`) that accepts a small JSON format and feeds the engine — wire your ERP's webhook or an outbox worker to it:

```
POST /erp/events
X-API-KEY: <your own key for this endpoint>
{
  "events": [
    { "kind": "product",  "op": "upsert", "item_number": "620206_01", "language_code": "nl",
"description": "Kabel 3x1.5 mm² (100 m)" },
    { "kind": "customer", "op": "upsert", "number": "58931", "name": "LUMITECH INSTALLATIES", "city":
"Apeldoorn", "vat_number": "NL001234567B01" },
    { "kind": "mapping",  "op": "upsert", "customer_number": "58931", "customer_item_number": "LT-ART-
001", "item_number": "PROD-001", "language_code": "nl" },
    { "kind": "customer", "op": "delete", "number": "10577" }
  ]
}
```

Or call the engine directly from your own code:

```python
from aiotic.sync import ChangeEvent, SyncEngine, HashStateStore

engine = SyncEngine(client, state=HashStateStore("sync-state.db"))

def on_item_changed(item):                       # your ERP's callback
    if item.blocked or item.discontinued:
        engine.apply(ChangeEvent.product_delete(item.no, "nl"))
    else:
        engine.apply(ChangeEvent.product_upsert(item.no, "nl", description=item.description))
```

Polling on `updated_at`

For ERPs without events, the engine has a watermark-based poller:

```python
from aiotic.sync import PollingChangeSource, ChangeEvent

def fetch_changed_since(watermark):
    rows = db.query("SELECT no, description, blocked, updated_at FROM items WHERE updated_at > ? ORDER BY
updated_at", watermark or "1970-01-01")
    events = [ChangeEvent.product_delete(r.no, "nl") if r.blocked else ChangeEvent.product_upsert(r.no,
"nl", description=r.description) for r in rows]
    return events, (rows[-1].updated_at.isoformat() if rows else watermark)

PollingChangeSource("items", fetch_changed_since, engine, interval=120).run_forever()
```

The watermark only advances when every event in the batch succeeded, so a transient failure is retried on the next run instead of being lost.

Mapping rules that keep AIOTIC accurate

ERP situation	AIOTIC action
Article blocked / discontinued	`DELETE /product` — it must stop being a valid article
Article re-activated	`PUT /product`
Customer deactivated	`DELETE /customer` (or rename with a legacy marker — see Archiving)
Customer merged into another	rename the old one `*** ZIE <new number> ***` so orders redirect, or delete it
Address / VAT / e-mail changed	`PUT /customer` — identification uses these fields
New language description	`PUT /product/{no}/{lang}`

ERP situation	AIOTIC action
Operator fixed an article in AIOTIC and you learned the customer's code	`PUT /customer-product` so the next order resolves automatically

Throttling and ordering

- The engine sends customers first, then products, then item mappings, because a mapping must reference an existing customer and product. Deletes go the other way round.
- Keep concurrency at 4–8. The client's token bucket (default 10 req/s) protects the tenant.
- Events for the same record can arrive out of order; the fingerprint is computed from the *record*, so the last write wins as long as your source delivers the latest state (not a diff).

KEEPING DATA IN SYNC

Reconciliation

Events get lost: a webhook fails while your service is being deployed, an outbox worker crashes, someone bulk-imports articles with a script that bypasses triggers. A **reconciliation run** compares the complete current data set with what was last sent and pushes only the differences. It is a safety net — schedule it nightly or weekly, and run it on demand after incidents.

How it works

1. Read the full data set from the ERP (a `SELECT`, not an API call to AIOTIC).
2. For each record compute the fingerprint of its AIOTIC shape.
3. Compare with the fingerprint stored from the last successful send:
   - different or unknown → `PUT`
   - identical → skip
4. Records that were sent before but are no longer in the ERP data set → `DELETE`.

Only steps 3 and 4 touch AIOTIC. A catalog of 200 000 articles with 50 changes results in 50 requests.

```
from aiotic.sync import ChangeEvent, HashStateStore, SyncEngine

engine = SyncEngine(client, state=HashStateStore("sync-state.db"), concurrency=6)

report = engine.reconcile(
    customers=(ChangeEvent.customer_upsert(c.no, name=c.name, city=c.city, vat_number=c.vat,
email=c.email) for c in erp.active_customers()),
    products=(ChangeEvent.product_upsert(p.no, "nl", description=p.description) for p in
erp.sellable_items()),
    mappings=(ChangeEvent.mapping_upsert(m.debtor, m.code, item_number=m.item, language_code="nl") for m
in erp.customer_items()),
    delete_missing=True,
)
print(report)   # sent=52 deleted=3 unchanged=204871 failed=0 in 41.2s
```

`aiotic sync reconcile --customers a.csv --products b.csv --mappings c.csv` does the same from files.

First run after an existing integration

If AIOTIC already holds data that your state store has never seen, the first reconcile would re-send everything once. Avoid that by seeding the state from AIOTIC:

```
aiotic sync bootstrap-state     # reads /customer, /product, /customer-product and records fingerprints
```

Deleting what disappeared

`delete_missing=True` removes records that were sent earlier but are absent from the current data set. Make sure the data set you pass is *complete* for that kind — passing only "changed" rows with `delete_missing=True` would delete everything else. When in doubt run with `delete_missing=False` first and look at the report.

Fingerprints and what counts as a change

The fingerprint covers the AIOTIC fields only (what you would `PUT`). A change in an ERP field you do not map (e.g. a price) does not trigger a request. Empty strings are treated as `null`.

Scheduling

Data	Events available	Reconcile
Customers	yes	nightly
Customers	no	every 15–60 min (small table)
Products	yes	weekly + after bulk imports
Products	no	nightly (reading 200k rows locally is cheap; the diff is small)
Mappings	yes	nightly

Reconciliation is idempotent and safe to run concurrently with event-driven sync: both write the same fingerprints.

KEEPING DATA IN SYNC

Multi-language products

Products are keyed by **article number + language code**. The language code is yours: `nl`, `de`, `en`, `fr` — whatever you use in your ERP. AIOTIC does not interpret it; it only uses it as part of the key and passes it through in mappings.

One language

Pick one code (say `nl`) and use it everywhere: every `PUT /product/{no}/nl`, every mapping's `language_code: "nl"`. Nothing else to do.

Several languages

Create one product record per language for the same article number:

```
PUT /product/620206_01/nl   { "description": "Kabel 3x1,5 mm² (100 m)" }
PUT /product/620206_01/de   { "description": "Kabel 3x1,5 mm² (100 m)" }
PUT /product/620206_01/en   { "description": "Cable 3x1.5 mm² (100 m)" }
```

Why bother, if the article number is the same? Two reasons:

1. **Descriptions help resolution.** When a document prints only a description (no usable article number) the text in the matching language is what gets compared.
2. **Mappings point at one language record.** `customer_number + customer_item_number → item_number + language_code`. Use the language the customer orders in, or a fixed default; the resolved `article_number` is the same either way.

Filtering

`GET /product/list?language_code=de` lists one language. Deletions are per language record: to retire an article completely, delete every language.

Sync engine

`ChangeEvent.product_upsert(item_number, language_code, description=…)` takes the language explicitly; emit one event per language when a description changes, and one delete per language when the article is discontinued.

KEEPING DATA IN SYNC

Archiving & superseded customers

Customer records in ERPs are rarely deleted; they are renamed, merged or marked inactive. AIOTIC understands the common conventions so you can mirror what your ERP does without a separate "status" field.

Deleting

`DELETE /customer/{number}` removes the record entirely. Use it when the debtor is gone for good. Orders that previously resolved to it will resolve to nobody (`customer_id: null` → `ATTENTION`) — which is what you want for a dead debtor.

Archiving by name marker

If your ERP keeps inactive customers with a marker in the name, keep sending the record as-is. AIOTIC **archives** a customer whose name contains a legacy or closure marker such as:

`formerly`, `voorheen`, `(old)`, `inactive`, `deprecated`, `do not use`, `niet gebruiken`, `niet meer leveren`, `stopt`, `sluiting`, `gesloten`, `geschlossen`, `vervallen`, `closed`, `failliet`, `fout`

An archived record:

- is never *chosen* by identification (it cannot steal an order from the live customer it resembles),
- still resolves when a document explicitly prints its customer number, so the operator sees "this is the archived record 12305" instead of nothing,
- is un-archived automatically when the marker disappears from the name on the next sync.

Archiving is evaluated on **every write**, so it follows your ERP naturally.

Redirects: "see customer N"

When a debtor is renumbered or merged, ERPs often keep the old record with a pointer in the name:

```
*** ZIE 12306 ***      ZIE KLANT 12306      SEE 10564      *** 11867 ***
```

AIOTIC parses this into a redirect. The old record stays searchable (its address, VAT and e-mail are exactly what the customer still prints on documents), but an order matched to it is filed under the **live successor** (`customer_id` = 12306). Chains are followed; cycles and archived targets stop the redirect at the last live number.

> **Not a redirect** `leveren gaat via 11649` ("deliveries go through 11649") is a delivery-routing note, not an identity change, and is deliberately ignored.

Products

There is no archiving for products: an article that must not be ordered is **deleted** (`DELETE /product/{no}/{lang}`). A deleted article is no longer a valid article number, so orders mentioning it go to `ATTENTION` instead of into your ERP.

Mappings

Delete a mapping when the customer stops using that code or when the target article is retired. Deleting a customer or product does not delete its mappings automatically; delete the mappings first in your sync order (the SDK does).

DRIVING ORDERS VIA THE API

Submitting documents

Most tenants receive orders through the mailbox AIOTIC watches. The API adds two more intake paths — for portals, scanners, EDI gateways, or your own mail handling.

Files → one order

POST `/order/upload` — auth: integration key

`multipart/form-data`. All `files` together form **one** purchase order (a PO with a separate price list, or a scan of two pages). Optional `request_id` (UUID v4) and any extra form fields, which are stored as `metadata`.

```
<span class="tab">curl</span>
curl -X POST https://acme.aiotic.ai/order/upload -H "X-API-Key: $KEY" \
  -F "files=@PO-4711.pdf;type=application/pdf" \
  -F "request_id=7f0c2b6e-1c2a-4f0e-9d4c-2c1a0b7e5a11" \
  -F "source=portal" -F "my_reference=TICKET-8812"
# {"request_id":"7f0c2b6e-1c2a-4f0e-9d4c-2c1a0b7e5a11","split":false}
```

```
<span class="tab">Python SDK</span>
import uuid
up = client.orders.upload(["PO-4711.pdf"], request_id=uuid.uuid4(), metadata={"source": "portal",
"my_reference": "TICKET-8812"})
status = client.orders.wait(up.request_id)          # polls with backoff until it lands
```

Accepted types: `.pdf .jpg .jpeg .png .txt .md`. Keep files under ~10 MB.

> **Always pass your own request_id** It makes the upload idempotent (a retry after a network error does not create a second order) and gives you the id before the call returns — handy for logging and for correlating in your ERP.

Raw e-mail → one or more orders

POST `/order/raw/upload` — auth: integration key

Upload a complete `.eml` (headers, body, attachments). AIOTIC classifies it first:

- **Purchase order** → processed like a mailbox mail. The e-mail body counts as input ("please change line 3 to 200 pcs" overrides the attachment).
- **Not a purchase order** → `400` with a structured `detail` (`error: not_a_purchase_order`, the detected `category`) and the mail is recorded under rejected e-mails so an operator can override.
- **Several orders in one mail** (tenant feature *order splitting*) → `split: true` plus one child order per detected order. **Poll the children**, not the source id.

```python
from aiotic import AioticValidationError

try:
    up = client.orders.upload_raw_email("mail.eml")
except AioticValidationError as e:
    if isinstance(e.detail, dict) and e.detail.get("error") == "not_a_purchase_order":
        log.info("skipped %s: %s", e.detail["category"], e.detail["subject"])
        raise SystemExit(0)
    raise

for rid in up.request_ids:                    # 1 id, or the children when split
    status = client.orders.wait(rid)
if up.split:
    group = client.orders.group(up.email_group_id)   # aggregate view
```

Classify without processing (dry run): `POST /order/raw/classify` → `{"category": "purchase_order"}`.

After the upload

You get a `request_id` immediately; processing runs in the background. Continue with Polling & notifications.

What you cannot do through upload

- Split multipart uploads into several orders — use the raw e-mail path.
- Attach a customer id up front. AIOTIC identifies the customer itself. (Your extra form fields land in `metadata` but are not used for identification.)
- Provide corrections — see Headless limits today.

DRIVING ORDERS VIA THE API

Polling & notifications

There is no push notification for status changes in the public API today (the AIOTIC app is updated through a channel that is not part of the public API; the optional processing webhook fires only once, before review). Polling is the mechanism, and it is cheap if done right.

One order

GET `/order_status/{request_id}` — auth: integration key

Poll with backoff — 2 s, then ×1.6, capped at 15 s — until the status is *landed* (`PROCESSED` , `ATTENTION` , `FAILED` , `MODIFIED`) or terminal:

```
status = client.orders.wait(request_id, timeout=600)     # exactly that loop
if status.status == "PROCESSED":
    ...
elif status.status == "ATTENTION":
    ...
```

`result` is populated once landed; `erp_ref` after a send; `last_error` , `retry_count` , `next_retry_at` during retries.

Many orders: the transition watcher

GET `/order_status/list?page=1&size=200` — auth: integration key

The list is newest first. A watcher polls the first page or two every 15–30 s, remembers the last status it saw per `request_id` , and emits a **transition** whenever it changes. The SDK ships one:

```
from aiotic.watch import OrderWatcher, SqliteWatchState

def on_change(t):
    print(t.request_id, t.from_status, "→", t.to)
    if t.to == "ATTENTION": notify_team(t.order)
    if t.to == "SENT":      mark_done_in_erp(t.order.erp_ref)

OrderWatcher(client, on_transition=on_change, state=SqliteWatchState("watch.db"), interval=20,
pages=2).run_forever()
```

`aiotic orders watch` runs the same loop and prints JSON lines — useful to pipe into a queue.

Costs: two requests every 20 s, regardless of volume. Orders older than the pages you look at are assumed settled; raise `pages` for very high volumes.

Split e-mails

GET `/order/group/{email_group_id}` — auth: integration key

Returns every child order of one source e-mail with their statuses — one call instead of N.

Files and artifacts

GET `/order/{request_id}/{filename}` — auth: integration key

Download an original upload (by file name as listed in `attachments`) or `latest_result.json` (the extracted order as a file). `GET /order/{request_id}/{filename}/preview` returns the same file with an inline `Content-Disposition` for showing inside your UI.

Choosing an interval

Need	Interval
Show live status in your UI	per-order `wait()` while the user looks; 5 s
Book `PROCESSED` orders automatically	watcher, 20–30 s
Nightly reporting	list pages once

Do not poll a single order faster than every 2 s, and stop polling terminal orders.

DRIVING ORDERS VIA THE API

ATTENTION, FAILED & retries

`ATTENTION` — a person is needed

`ATTENTION` is not an error. Extraction finished (`result` is there) but at least one thing needs judgement. Find out what:

Signal in `result`	Meaning	Typical fix
`items[i].article_number == null`	line could not be resolved to your catalog	add the product or a customer item mapping; operator picks the article
`customer.customer_id == null`	sender not identified with confidence	check the customer record (VAT, e-mail, address); operator picks the customer
`items[i].quantity_state == "Unrecognised"`	quantity present but unreadable (handwritten, smudged)	operator enters it
`state` mentions a master-data disagreement	document says X, your record says Y	operator confirms; you may update the master record
all lines dropped (assortment sheet, nothing ordered)	nothing to book	operator cancels

What you can do headless:

- **Route it.** Create a ticket / send an e-mail with the AIOTIC app link (`https://<app>/orders?...`) and the reason. The `OrderWatcher` gives you the transition.
- **Learn from it.** When an operator resolves an unknown article and the line carries a `customer_item_number` , create the mapping so the next order from that customer resolves automatically.
- **Do not auto-send it.** Sending an `ATTENTION` order via `POST /erp/send` is allowed — it is an explicit override — but it hands your ERP a payload with `null` article numbers or customer. Reserve it for a human who looked.

`FAILED` — terminal error

Something structural went wrong (`last_error` says what: unreadable file, no text, a processing error that exhausted retries). One retry is reasonable, then alert:

POST `/order/retry/{request_id}` — auth: integration key

```
if status.status == "FAILED" and status.retry_count == 0:
    new = client.orders.retry(status.request_id)    # 400 if not FAILED
    track(new.request_id)                          # the OLD id becomes REPROCESSED
```

A retry creates a **new order** with the same files; the original becomes `REPROCESSED` and keeps its history. Track the new id from then on.

`RETRY_PENDING` — AIOTIC is retrying

Transient failures (temporary upstream outage, timeout) are retried automatically with backoff. `next_retry_at` tells you when; `retry_count` how often. Do nothing — do **not** call `/order/retry` , it is only for `FAILED` .

Rejected e-mails

Not an order status at all: a mail the classifier decided is *not* a purchase order. See Rejected e-mails.

Decision table for an automated service

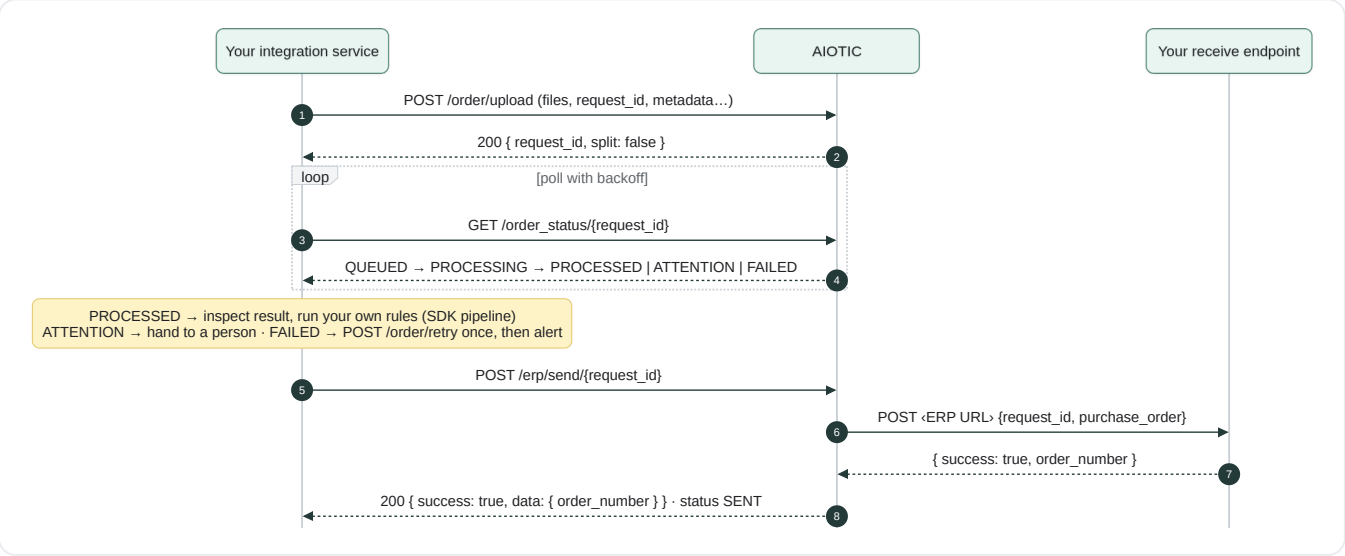
```
match status.status:
    case "PROCESSED":                       book_or_send(status)
    case "ATTENTION":                       route_to_human(status)
    case "FAILED" if status.retry_count == 0: client.orders.retry(status.request_id)
    case "FAILED":                          alert(status.last_error)
    case "RETRY_PENDING" | "QUEUED" | "PROCESSING" | "SENDING": pass
    case "SENT":                            reconcile_erp_ref(status.erp_ref)
    case "MODIFIED":                        book_or_send(status)   # operator edited in the app
    case "REPROCESSED" | "CANCELED":        stop_tracking(status.request_id)
    case _:                                 log.warning("unknown status %s", status.status)  # forward
compatible
```

DRIVING ORDERS VIA THE API

Sending to the ERP

POST `/erp/send/{request_id}` — auth: integration key

Hands a sendable order to your ERP receive endpoint. Synchronous: AIOTIC calls your endpoint, waits for the answer (30 s timeout by default) and returns its outcome.



A fully API-driven flow: upload, poll, decide, send.

Preconditions

- Status is `PROCESSED`, `MODIFIED` or `ATTENTION` — otherwise `409`.
- The tenant has an ERP endpoint configured — otherwise `503`.
- No other send is in progress for this order — otherwise `409` (the order is locked as `SENDING`).

Outcomes

Your endpoint answered	AIOTIC returns	Order status
`{"success": true, "order_number": "SO-981"}`	`200 {"success": true, "request_id": "…", "data": {…your body…}}`	`SENT`, `erp_ref = "SO-981"`
`{"success": false, "error": "…"}`	`422 {"detail": "ERP error: …"}`	rolled back (e.g. `PROCESSED`)
non-JSON / timeout / unreachable	`500 {"detail": "ERP server error …"}`	rolled back

```
<span class="tab">curl</span>
curl -X POST https://acme.aiotic.ai/erp/send/7f0c2b6e-1c2a-4f0e-9d4c-2c1a0b7e5a11 -H "X-API-Key: $KEY"
# {"success":true,"request_id":"7f0c…","data":{"success":true,"order_number":"SO-2026-00981"}}
```

```
<span class="tab">Python SDK</span>
from aiotic import AioticErpRejectedError, AioticConflictError

try:
    res = client.erp.send(request_id)
    print("booked as", res.erp_order_number)
except AioticErpRejectedError as e:
    print("ERP said no:", e.erp_error)          # the text your endpoint returned
except AioticConflictError:
    print("not sendable right now (status or concurrent send)")
```

Automating the decision (model C)

```
status = client.orders.get(request_id)
po = status.result
safe = (
    status.status == "PROCESSED"
    and po.customer and po.customer.customer_id in trusted_customers
    and not po.unresolved_items
    and all(i.quantity for i in po.items)
)
if safe:
    client.erp.send(request_id)
else:
    route_to_human(status)
```

Put the real checks in your **receive endpoint** (the pipeline) rather than here: the endpoint sees the final payload, and a rejection there shows up for operators as a clear message. The send decision above only needs to answer "is a human required first?".

Overriding `ATTENTION`

Allowed, logged on the AIOTIC side as an explicit override. Your endpoint then receives `null` article numbers or customer — make sure it rejects those unless a person confirmed.

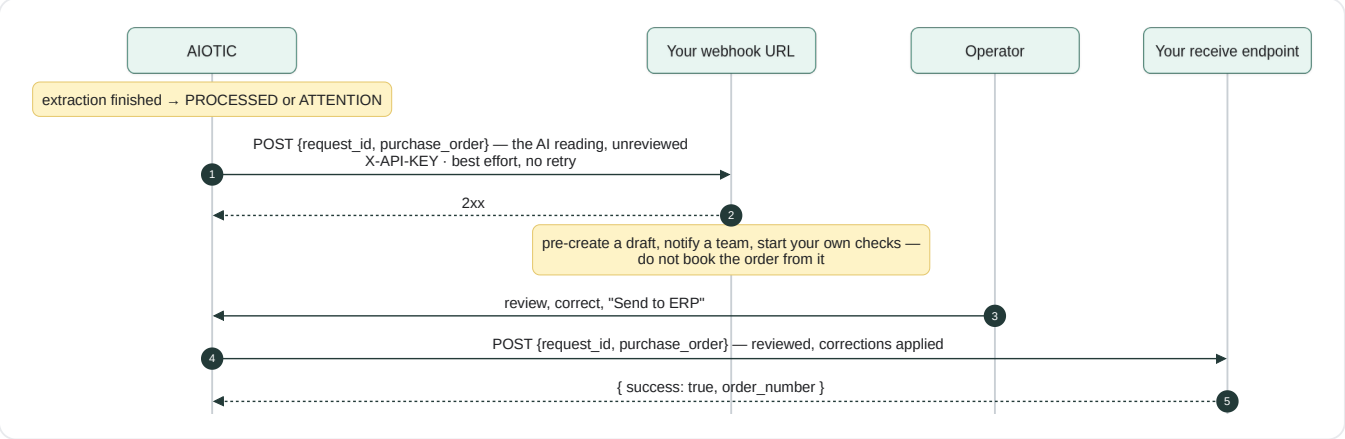
Sending from your own UI

Show the order (from `result`), let the user confirm, call `send`. Remember that **corrections cannot be submitted through the API today**; if your user changes a value, apply the change in your ERP after receiving the payload (match on `request_id`), or make the correction in the AIOTIC app. See Headless limits today.

DRIVING ORDERS VIA THE API

The processing webhook

An optional tenant feature: the moment extraction finishes — **before** any human review, and regardless of `PROCESSED` or `ATTENTION` — AIOTIC `POST`s the extracted order to a URL you provide.



Two moments, two payloads: the AI's reading (webhook) and the reviewed order (ERP hand-off).

Contract

```
POST https://integration.example.com/aiotic/processing
Content-Type: application/json
X-API-KEY: <webhook key you provided>

{ "request_id": "…", "purchase_order": { …PurchaseOrder as in result… } }
```

Respond `2xx`. That is all: delivery is **best effort** — no retries, no signature, no event type, no status field. If your endpoint is down, the signal is lost (the order itself is not; it is still in AIOTIC).

The body is the `PurchaseOrder` shape (with `quantity_state`, `customer_item_number`, `delivery_date_from/to`), *not* the ERP hand-off subset.

What it is good for

- Pre-create a draft in your system so the ERP hand-off is a state change, not a new record.
- Notify a team that an order from customer X arrived.
- Start your own enrichment (price check, stock) in parallel with the human review.
- Statistics: volume per customer, time from arrival to send.

What it is not

- **Not the hand-off.** The payload is unreviewed. Booking from it means booking the AI's mistakes.
- **Not a status feed.** It fires once per order. For `SENT` / `CANCELED` use the watcher.
- **Not guaranteed.** Design for a missed call: the watcher (or a nightly list) is the reconciliation.

Receiving it with the SDK

```
from aiotic.webhooks import ProcessingWebhookReceiver, create_webhook_routers

def on_processed(req):                # req: ProcessingWebhookRequest
    drafts.create(req.request_id, req.purchase_order)

app.include_router(create_webhook_routers(processing=ProcessingWebhookReceiver(WEBHOOK_KEY,
on_processed)))
# → POST /aiotic/processing
```

`build_app()` wires this automatically when `AIOTIC_WEBHOOK_KEY` is set.

Enabling it

A tenant admin sets *output → webhook* (URL, key, enabled) in the AIOTIC app; the AIOTIC team can do it during onboarding. The mock tenant takes `MOCK_WEBHOOK_URL` / `MOCK_WEBHOOK_KEY` or a `POST /_mock/config`.

Future

Signed, typed, retried event webhooks (`order.processed`, `order.attention`, `order.sent`, …) are proposed. The SDK already contains the signature verifier so your endpoint can adopt them without a rewrite.

DRIVING ORDERS VIA THE API

Rejected e-mails

Not every mail in the order mailbox is an order. Quotations, order confirmations, delivery notes, invoices, internal requisitions and marketing mail are **classified and rejected** — recorded with the classifier's reason, never extracted, and never sent to your ERP. The product prefers a false "not an order" (recoverable by a person) over a phantom order in your ERP.

Listing

GET `/rejected/list?status=pending` — auth: integration key


```
{ "items": [ { "request_id": "…", "email_type": "invoice", "sender": "supplier@example.com", "from_name":
"Supplier",
               "subject": "Invoice 12345", "timestamp": "…", "classification_reason": "Document reads as
an invoice, not an order",
               "rejection_status": "pending", "original_email_type": null, "message_id": "
<abc@example.com>", "metadata": {} } ],
  "total": 12, "limit": 100, "offset": 0 }
```


`status=pending` (default) — awaiting a decision. `status=overridden` — already reprocessed by someone.

Overriding

POST `/rejected/{request_id}/reprocess` — auth: integration key

Forces the mail through order processing under the **same** `request_id`. The original classification is preserved (`original_email_type`) and the item moves to `overridden`. From then on it is a normal order: poll `GET /order_status/{request_id}`.

```
for mail in client.rejected.list().items:
    if looks_like_an_order(mail):                    # your heuristic, e.g. known sender + "bestelling"
in subject
        client.rejected.reprocess(mail.request_id)
```

`409` if already overridden or a concurrent override won; `404` for ids that are not rejections.

Uploaded raw e-mails

When *you* upload an `.eml` that turns out not to be an order, `POST /order/raw/upload` answers `400` with the structured `detail` **and** records it here, so the same override path applies.

Typical automation

- Daily digest of `pending` rejections to the sales inbox (sender, subject, reason).
- Auto-override for a short allow-list of senders whose mails are always orders but look like something else (e.g. a customer whose "confirmation" is really an order).
- Never auto-override globally: the classifier is right far more often than not.

DRIVING ORDERS VIA THE API

Headless limits today

This page is deliberately blunt. If you plan to integrate **without** the AIOTIC app (models B and C), these are the things the public API does not offer yet, and what to do instead. Proposed additions are in the appendix; nothing there exists until the changelog says so.

Capability	Status	Workaround today
Submit corrections to an extracted order before sending	Not available. Corrections are made in the AIOTIC app; `POST /erp/send` merges them.	Apply corrections on *your* side: your receive endpoint gets the AI's reading plus app corrections; adjust in your ERP after booking (match on `request_id`), or route the order to the app for the fix.
Cancel an order	Not available (`CANCELED` is set from the app).	Leave it unsent; track it as ignored in your system. Ask an operator to cancel in the app if the queue must stay clean.
Status-change notifications (webhook per transition)	Not available.	Order watcher polling the list; the processing webhook for the single "landed" moment.
Signed / retried webhooks	Processing webhook is static-key, best effort.	Verify the key; treat as a hint; reconcile by polling.
Batch upserts for master data	One request per record.	Parallel upserts (4–8) via the SDK engine; only changed records.
`updated_since` / ETag on list endpoints	Not available.	Keep your own state (fingerprints for master data, last status for orders).
Filtering `order_status/list` by status or date	Not available; newest-first pagination only.	Watcher over the first pages; nightly full page walk for reporting.
Scoped / read-only keys	Two keys: integration (all) and sync (master data).	Keep the integration key in one service; give everything else the sync key or no key.
Reprocess with instructions ("ignore page 2")	App only.	Use the app for that order.
Per-order customer hint on upload	Not available; identification is evidence-based.	Ensure the customer record has VAT/e-mail/address; add item mappings.

What *is* fully available headless

Upload (files and raw e-mails, incl. split), classify, status, list, group view, file/artifact download, retry, rejected-mail triage and override, ERP send with correct locking and rollback, complete master-data CRUD and search, health and platform status.

That is enough for **model C** (automated send of clean orders, humans for the rest) and for a read-only **model B** (your UI shows and sends, corrections happen in the app or in your ERP).

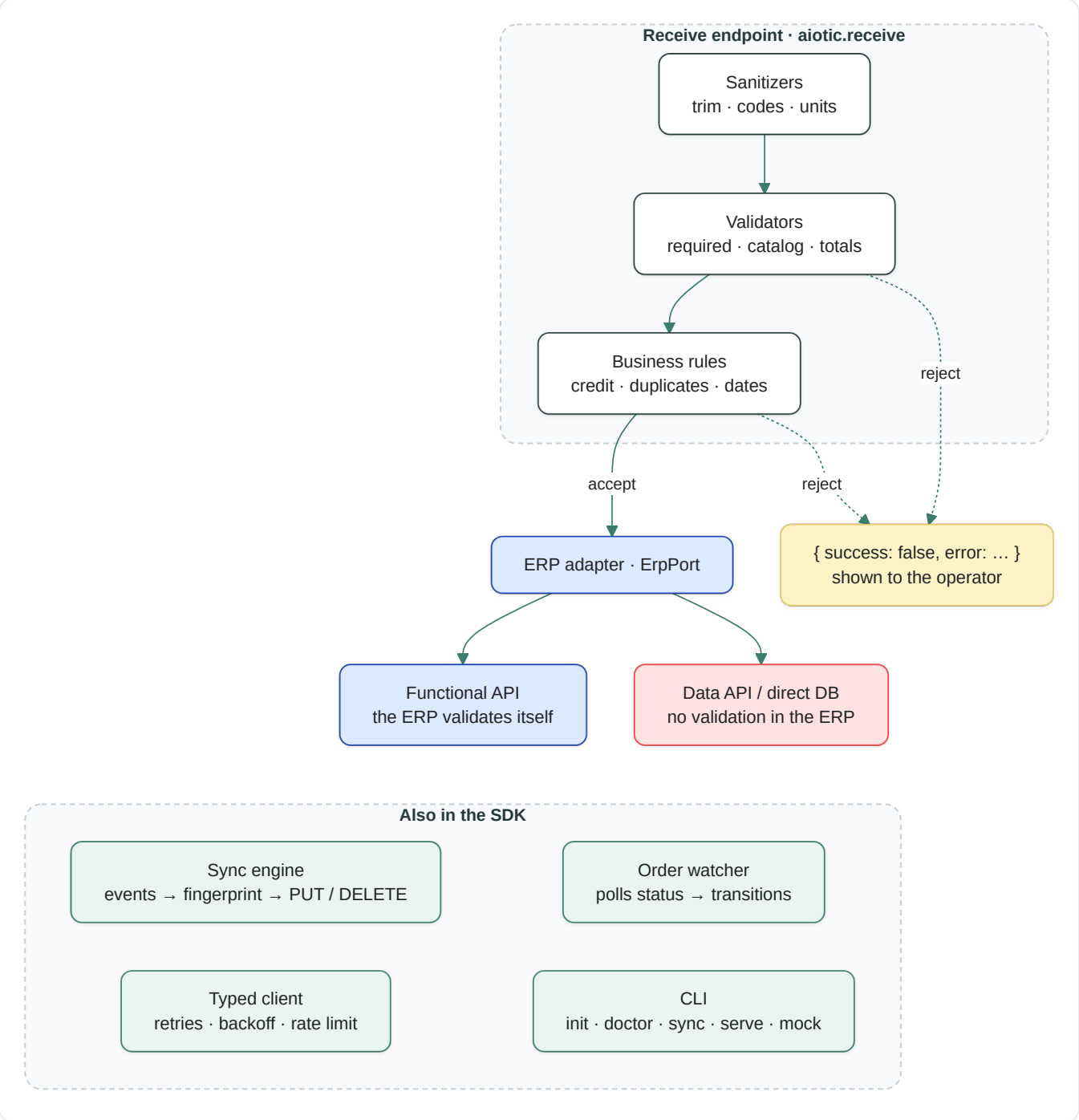
If you need more

Tell your AIOTIC contact which item above blocks you. The proposal appendix exists so we can prioritise with real integrator needs.

PYTHON SDK

Python SDK — overview & install

`aiotic-sdk` is the reference implementation of everything in this guide. Use it as a library, as a ready-made service, or as a source of patterns for another language.



Modules and how they connect.

Install

```
pip install "aiotic-sdk[all]"          # client + FastAPI service + CLI
pip install aiotic-sdk                 # client and pipeline only (httpx, pydantic)
```

Python ≥ 3.11. Extras: `server` (FastAPI, uvicorn), `cli` (typer, rich).

Modules

Module	What it gives you	Guide page
`aiotic.AioticClient` / `AsyncAioticClient`	Every public endpoint as a typed method; retries with backoff; token-bucket rate limit; automatic sync-key routing; `orders.wait()`	Client reference
`aiotic.models`	Pydantic v2 models pinned to the public OpenAPI (`PurchaseOrder`, `OrderStatus`, `ErpReceiveRequest`, …)	Purchase-order model
`aiotic.receive`	`ErpReceiver` (framework-free) + `create_receive_router` (FastAPI): key check, idempotency, pipeline, correct response	ERP receive endpoint
`aiotic.pipeline`	Sanitizers → validators → business rules with built-ins	Pipeline
`aiotic.erp`	`ErpPort`, `CatalogPort`, `CustomerPort`; templates `FunctionalApiAdapter`, `DataApiAdapter`; `InMemoryErp`	ERP adapters
`aiotic.sync`	`SyncEngine`, `ChangeEvent`, `HashStateStore`, `PollingChangeSource`, `reconcile()`	Sync engine
`aiotic.watch`	`OrderWatcher` — status transitions from polling	Order watcher
`aiotic.webhooks`	Processing-webhook receiver, ERP change-event endpoint, HMAC verifier	Webhooks & receivers
`aiotic.service`	`build_app()` — one call, complete service	Bootstrapping
`aiotic.cli`	`aiotic init · doctor · orders · sync · serve · mock`	CLI
`aiotic.mock`	A mock AIOTIC tenant (`aiotic mock`)	Mock server

Configuration

`Settings.from_env()` reads `AIOTIC_*` environment variables (and a `.env` file in the working directory):

Variable	Purpose
`AIOTIC_BASE_URL`	`https://acme.aiotic.ai`
`AIOTIC_API_KEY`	integration key
`AIOTIC_SYNC_API_KEY`	optional sync key (used automatically for master-data calls)
`AIOTIC_ERP_RECEIVE_KEY`	key AIOTIC sends to **your** receive endpoint
`AIOTIC_WEBHOOK_KEY`	key AIOTIC sends to your processing webhook
`AIOTIC_TIMEOUT`, `AIOTIC_MAX_RETRIES`, `AIOTIC_RATE_LIMIT`	30 s · 3 · 10 req/s

Design principles

- **The API is the truth.** Models carry the exact field names and nullability of the OpenAPI document; unknown fields are preserved, so a newer tenant never breaks an older SDK.
- **The integration layer validates.** Because some ERPs cannot. Every check is a small class you can add, remove or reorder.
- **Never a full re-upload.** The sync engine sends a record only when its fingerprint changed.
- **Framework-free core.** `ErpReceiver.handle()`, `Pipeline.run()`, `SyncEngine.apply()` are plain Python. FastAPI routers are thin wrappers; use Flask, Django or a queue consumer if you prefer.
- **Tested against the mock.** The SDK's own test-suite runs the mock tenant in-process; your tests can too.

Versioning

The SDK's `aiotic.API_VERSION` names the API version it was verified against; the guide's changelog lists breaking changes (there have been none: the API evolves additively).

PYTHON SDK

Bootstrapping a service

From zero to a running integration service — the receive endpoint, the webhooks, and the sync engine — in a few commands. Then swap the demo ERP for yours.

1. `aiotic init`

```
mkdir acme-aiotic && cd acme-aiotic
python -m venv .venv && source .venv/bin/activate && pip install "aiotic-sdk[all]"
aiotic init
```

Writes:

- `.env` — base URL, integration key, an empty sync key, a **generated** `AIOTIC_ERP_RECEIVE_KEY` (give this value to AIOTIC together with your endpoint URL), an empty webhook key.
- `service.py` — three lines that build the app with the in-memory demo ERP.

2. `aiotic doctor`

Checks the tenant is reachable, the keys work, and master data is present. Run it after every configuration change and in your deployment health checks.

3. `aiotic serve`

Runs `service.py` with uvicorn on `:9000`:

Route	Purpose
`POST /aiotic/orders`	ERP receive endpoint
`POST /aiotic/processing`	processing webhook (when `AIOTIC_WEBHOOK_KEY` is set)
`POST /erp/events`	ERP change events → sync engine
`GET /healthz`	liveness
`GET /docs`	OpenAPI UI of *your* service

4. Replace the ERP

`service.py`, grown up:

```
import pyodbc
from aiotic import AioticClient
from aiotic.erp.data_api import DataApiAdapter
from aiotic.pipeline import Pipeline, sanitizers as S, validators as V, rules as R
from aiotic.receive import SqliteStore
from aiotic.service import build_app
from aiotic.sync import HashStateStore, SyncEngine

erp = DataApiAdapter(lambda: pyodbc.connect(DSN), paramstyle="qmark")      # or FunctionalApiAdapter(...)
store = SqliteStore("integration.db")
pipeline = Pipeline(
    sanitizers=[S.StripWhitespace(), S.NormalizeCountryCodes(), S.NormalizeCurrency(),
S.MapUnits({"doos": "BOX"}), S.FillShippingFromCustomer()],
    validators=[V.RequiredFields(), V.CustomerResolved(erp), V.ArticlesInCatalog(erp),
V.PositiveQuantities(), V.CurrencyAllowed(["EUR"])],
    rules=[R.CustomerNotBlocked(erp), R.NoDuplicateOrder(store), R.ShipToAddressComplete()],
)
sync_engine = SyncEngine(AioticClient(), state=HashStateStore("sync-state.db"))

app = build_app(erp=erp, pipeline=pipeline, store=store, sync_engine=sync_engine)
```

`build_app` accepts any `ErpPort`; when the adapter also implements `CatalogPort` / `CustomerPort` (both templates do), the default pipeline picks them up automatically, so the explicit `pipeline=` above is optional.

5. Add the sync side

- Events: point your ERP's webhooks / outbox worker at `POST /erp/events` (format), or call `sync_engine.apply(...)` from your own code.
- Polling: a `PollingChangeSource` in a background thread or a separate process.
- Reconciliation: a scheduled `aiotic sync reconcile …` or `sync_engine.reconcile(...)`.

6. Deploy

- Run behind TLS (a reverse proxy or your platform's ingress). The receive endpoint must be reachable by the tenant.
- One instance is enough for most volumes; if you scale out, replace `SqliteStore` / `HashStateStore` / `SqliteWatchState` with your database (each is a ~20-line protocol).
- Secrets from the environment; never bake `.env` into an image.
- Health: `GET /healthz` for liveness; `aiotic doctor` for readiness in CI.

A `Dockerfile` for the service:

```
FROM python:3.12-slim
WORKDIR /app
RUN pip install --no-cache-dir "aiotic-sdk[all]" pyodbc
COPY service.py .
CMD ["uvicorn", "service:app", "--host", "0.0.0.0", "--port", "9000"]
```

PYTHON SDK

Client reference

```
from aiotic import AioticClient, AsyncAioticClient

client = AioticClient(base_url="https://acme.aiotic.ai", api_key="…", sync_api_key="…")   # or
AioticClient() from AIOTIC_* env
```

Both clients expose the same resources; the async one is `await`-able and iterators are `async for`.

Behaviour

Feature	Detail
Retries	on connection errors and `408 425 429 502 503 504`; backoff 0.5 s·2^n + jitter, capped 30 s; `Retry-After` honoured; uploads are not retried after the first attempt
Rate limit	token bucket, `rate_limit` req/s (default 10; `0` = off); safe across threads
Keys	master-data paths use `sync_api_key` when set, everything else the integration key
Errors	typed exceptions with `.status`, `.detail` (string or object), `.request_id`; see Errors
Unknown fields	preserved on every model

`client.orders`

Method	Endpoint
`upload(files, *, request_id=None, metadata=None) → OrderUploadResponse`	`POST /order/upload` — `files`: paths, `(name, bytes)` or `(name, bytes, content_type)`
`upload_raw_email(eml, *, request_id=None) → OrderUploadResponse`	`POST /order/raw/upload` — raises `AioticValidationError` with structured `detail` for non-orders
`classify_raw_email(eml) → EmailClassification`	`POST /order/raw/classify`
`get(request_id) → OrderStatus`	`GET /order_status/{id}`
`list(*, page=1, size=100) → OrderListResponse` · `iter_all(size=200, max_pages=None)`	`GET /order_status/list`
`group(email_group_id) → OrderGroup`	`GET /order/group/{id}`
`download_file(request_id, filename, *, preview=False) → bytes`	`GET /order/{id}/{filename}[/preview]`
`retry(request_id) → OrderUploadResponse`	`POST /order/retry/{id}`
`wait(request_id, *, until=LANDED_STATUSES, timeout=600, initial_interval=2, max_interval=15) → OrderStatus`	polling helper

`OrderUploadResponse.request_ids` returns the ids to poll (children when `split`).

`client.erp`

Method	Endpoint
`send(request_id) → ErpSendResponse`	`POST /erp/send/{id}` — raises `AioticErpRejectedError(.erp_error)` on `success: false`, `AioticConflictError` when not sendable

`ErpSendResponse.erp_order_number` — your ERP's reference from the response body.

`client.rejected`

`list(*, page, size, status="pending")`, `get(request_id)`, `reprocess(request_id)`.

`client.customers` (sync key ok)

`list(*, page, size)`, `iter_all(size=500)`, `search(query, *, top_k=10)`, `get(number)`, `upsert(number, CustomerUpsert | dict)`, `delete(number)`.

`client.products` (sync key ok)

`list(*, page, size, language_code=None)`, `iter_all(...)`, `get(item_number, language_code)`, `upsert(item_number, language_code, ProductUpsert | dict)`, `delete(item_number, language_code)`.

`client.customer_products` (sync key ok)

`list(*, page, size, customer_number=None, customer_item_number=None, item_number=None, language_code=None)`, `iter_all(...)`, `get(customer_number, customer_item_number)`, `upsert(customer_number, customer_item_number, CustomerProductUpsert | dict)`, `delete(...)`.

Misc

`client.health() → HealthCheck`, `client.system_status() → SystemStatus`, `client.mailbox.fetch_all()`.

Exceptions

```
AioticError
├── AioticAuthError            401
├── AioticNotFoundError        404
├── AioticConflictError        409
├── AioticValidationError      400 / 415 / 422   (.detail may be an object)
├── AioticErpRejectedError     422 on /erp/send  (.erp_error)
├── AioticUnavailableError     503
├── AioticServerError          5xx
└── AioticTransportError       network, after retries
```

Async example

```python
import asyncio
from aiotic import AsyncAioticClient

async def main():
    async with AsyncAioticClient() as client:
        up = await client.orders.upload(["PO.pdf"])
        status = await client.orders.wait(up.request_id)
        async for c in client.customers.iter_all():
            ...

asyncio.run(main())
```

Testing your code

`AioticClient(transport=httpx.MockTransport(handler))` injects a fake transport; or run the mock tenant in-process as the SDK's own tests do.

PYTHON SDK

Validation & business-rule pipeline

`aiotic.pipeline` is the integration layer's answer to ERPs that cannot validate an order themselves. It runs three kinds of steps in order and produces a `Verdict`:

```
ErpPurchaseOrder ⟶ Sanitizers ⟶ Validators ⟶ Business rules ⟶ Verdict(ok | errors, warnings, cleaned order)
```

- **Sanitizer** — `apply(order, ctx) -> order`: returns a cleaned *copy*.
- **Validator** — `check(order, ctx) -> Iterable[Issue]`: data shape and consistency.
- **BusinessRule** — same contract; policy. Kept in a separate list for readability.

An `Issue` has a `code`, a human `message`, a `severity` (`ERROR` rejects, `WARNING` is recorded) and an optional `path` (`items[2].article_number`).

```python
from aiotic.pipeline import Pipeline, sanitizers as S, validators as V, rules as R

pipeline = Pipeline(
    sanitizers=[S.StripWhitespace(), S.NormalizeCountryCodes(), S.NormalizeCurrency(),
S.NormalizePostalCodes(), S.MapUnits(), S.FillShippingFromCustomer(), S.DropEmptyLines()],
    validators=[V.RequiredFields(), V.CustomerResolved(erp), V.ArticlesInCatalog(erp),
V.PositiveQuantities(), V.LineTotalsConsistent(), V.OrderTotalConsistent(), V.CurrencyAllowed(["EUR"]),
V.DeliveryDateSane()],
    rules=[R.NoDuplicateOrder(store), R.CustomerNotBlocked(erp), R.OrderValueWithin(max_total=50_000),
R.ShipToAddressComplete()],
)

verdict = pipeline.run(order, {"request_id": request_id})
if not verdict.ok:
    return ErpReceiveResponse.rejected(verdict.error_message())     # shown to the operator
erp.create_sales_order(request_id, verdict.order)                  # the cleaned order
```

`aiotic.service.default_pipeline(erp, store)` builds this configuration (adapting to what the adapter implements).

Built-in sanitizers

Class	Effect
`StripWhitespace()`	trim + collapse whitespace on every string; empty → `null`
`NormalizeCountryCodes(extra=None, default=None)`	`Nederland` / `NLD` → `NL`; extend with your own aliases
`NormalizeCurrency(default="EUR")`	`€` → `EUR`, uppercase; fills missing line currencies from the header
`NormalizePostalCodes()`	`7327aa` → `7327 AA` (NL); trims others
`NormalizeOrderNumber(pattern, replacement="-", max_length=35)`	strip characters your reference field cannot hold; truncate
`MapUnits(mapping=None, default_unit="ST", keep_unknown=True)`	`stuks` / `Stk` / `pcs` → `ST`, `mtr` → `M`, … plus your map
`DropEmptyLines()`	remove lines with neither quantity, article nor price
`FillShippingFromCustomer()`	empty ship-to → copy of the customer block
`Custom(name, fn)`	any `(order, ctx) -> order`

Built-in validators

Class	Rejects when
`RequiredFields(require_customer_id=True, require_delivery_date=False)`	no order number / date / lines / customer id
`CustomerResolved(customers: CustomerPort)`	`customer_id` unknown in the ERP (stores the ERP record in `ctx["erp_customer"]`)
`ArticlesInCatalog(catalog: CatalogPort)`	a line has no article or an unknown one
`PositiveQuantities(max_quantity=100000)`	quantity missing / ≤ 0; warns above the max
`LineTotalsConsistent(tolerance=0.05, severity=WARNING)`	`quantity × price ≠ line_total`
`OrderTotalConsistent(tolerance_ratio=0.25, severity=WARNING)`	lines sum far from the document total
`CurrencyAllowed(["EUR"])`	header or line currency not in the set
`DeliveryDateSane(max_days_in_past=30, max_days_ahead=365)`	not ISO; warns when implausible
`Custom(name, fn)`	any `(order, ctx) -> Iterable[Issue]`

Built-in business rules

Class	Rejects when
`NoDuplicateOrder(store)`	same `(customer_id, order_number)` was booked before (remembered only after a successful booking)
`CustomerNotBlocked(customers: CustomerPort)`	`is_blocked(customer_id)`
`OrderValueWithin(min_total=None, max_total=None, severity=WARNING)`	value outside the band
`ShipToAddressComplete()`	street / postal code / city missing
`@rule("name")` decorator	turn a generator function into a rule

Writing your own

```
from aiotic.pipeline import Issue, Severity
from aiotic.pipeline.rules import rule

@rule("credit_limit")
def credit_limit(order, ctx):
    cust = ctx.get("erp_customer")                  # set by CustomerResolved
    total = order.total_price or sum(i.line_total or 0 for i in order.items)
    if cust and cust.get("credit_limit") is not None and total > cust["credit_limit"]:
        yield Issue("credit_limit", f"Order value {total:.2f} exceeds credit limit
{cust['credit_limit']:.2f}",
                    path="total_price", severity=Severity.ERROR)
```

Guidelines:

- Yield `Issue`s; do not raise. One issue per problem, with a `path`.
- Messages are read by operators — say what is wrong and where.
- Keep steps pure; the pipeline deep-copies the order once and hands each step the current version.
- Use `ctx` to pass lookups between steps instead of querying twice.
- Set `stop_on_first_error=True` on the pipeline if later checks are expensive and pointless after a rejection.

Verdict

Member	Meaning
`ok`	no `ERROR` issues
`errors`, `warnings`	filtered issue lists
`order`	the sanitized order — book **this**, not the input
`steps`	which steps ran (for logs)
`error_message(limit=5)`	one line for the operator

Warnings are worth logging and, for model C automation, worth turning into "needs a human" even though the order is technically bookable.

PYTHON SDK

ERP adapters (functional vs data API)

The receive endpoint talks to your ERP through one small interface. Implement it once; everything else (key check, idempotency, pipeline, response) is shared.

The ports

```
class ErpPort(Protocol):
    def find_order_by_request_id(self, request_id: str) -> ErpCreateResult | None: ...   # idempotency
    def create_sales_order(self, request_id: str, order: ErpPurchaseOrder) -> ErpCreateResult: ...

class CatalogPort(Protocol):                      # used by validators.ArticlesInCatalog
    def product_exists(self, article_number: str) -> bool: ...

class CustomerPort(Protocol):                     # used by validators.CustomerResolved,
rules.CustomerNotBlocked
    def get_customer(self, customer_id: str) -> dict | None: ...
    def is_blocked(self, customer_id: str) -> bool: ...
```

`ErpCreateResult(order_number, created, details)` carries your ERP's reference. Raise `ErpRejected("text")` for **business** refusals — the text goes to the operator; let anything else propagate — it becomes a generic failure and AIOTIC rolls the order back.

Template 1: functional API

Your ERP has an API that validates and creates orders (Business Central, Exact, Odoo, SAP B1 Service Layer, most SaaS ERPs). Copy `aiotic/erp/functional_api.py`, rename, and replace the three `_map_*` methods and the two HTTP calls:

```
class BusinessCentralAdapter(FunctionalApiAdapter):
    def _map_header(self, request_id, o):
        return {
            "customerNumber": o.customer.customer_id,
            "externalDocumentNumber": o.order_number[:35],
            "requestedDeliveryDate": o.delivery_date,
            "currencyCode": o.currency,
            "shipToName": o.shipping_details.recipient.company,
            "shipToAddressLine1": o.shipping_details.recipient.address.street,
            "shipToPostCode": o.shipping_details.recipient.address.postal_code,
            "shipToCity": o.shipping_details.recipient.address.city,
            "shipToCountry": o.shipping_details.recipient.address.country,
            "salesOrderLines": [{"lineType": "Item", "lineObjectNumber": i.article_number, "quantity":
i.quantity, "unitOfMeasureCode": i.unit} for i in o.items],
        }
```

The ERP's own error text (e.g. *"The Item does not exist. No.='PROD-999'"*) is raised as `ErpRejected` and reaches the operator unchanged. A light pipeline (sanitizers, `RequiredFields`, `NoDuplicateOrder`) is still recommended.

Template 2: data API / direct database

Your ERP exposes tables (or a generic data API that maps to tables). `aiotic/erp/data_api.py` shows the pattern with DB-API 2.0:

- all SQL in one dictionary (`SQL[...]`) so table/column names are changed in one place;
- header and lines in **one transaction**, keyed on `external_ref = request_id` ;

- `product_exists`, `get_customer`, `is_blocked` implemented against the same tables, so the adapter doubles as `CatalogPort` and `CustomerPort` and the pipeline can enforce what the ERP will not.

```
import pyodbc
from aiotic.erp.data_api import DataApiAdapter

class MyErp(DataApiAdapter):
    SQL = {**DataApiAdapter.SQL,
           "item_exists": "SELECT 1 FROM dbo.Item WHERE No_ = ? AND Blocked = 0",
           "customer": "SELECT No_, Name, Blocked, [Credit Limit (LCY)] FROM dbo.Customer WHERE No_ = ?",
           ...}

erp = MyErp(lambda: pyodbc.connect(DSN), paramstyle="qmark", number_prefix="SO")
```

For the quick start, `sqlite_demo()` returns a connect factory to a seeded SQLite database with the three example tables.

> **With a data API the pipeline is mandatory** Nothing else validates. Run at least `CustomerResolved`, `ArticlesInCatalog`, `PositiveQuantities`, `CustomerNotBlocked` and `NoDuplicateOrder` before `create_sales_order`. See Validation when your ERP has no functional API.

Template 0: in-memory (tests, demos)

`InMemoryErp(products=set(), customers={...}, blocked=set())` implements all three ports and behaves like a functional ERP (rejects unknown items). Used by the quick start and the SDK tests.

Idempotency inside the adapter

`find_order_by_request_id` is the second line of defence after the receiver's `IdempotencyStore`. Implement it against the external-reference column; it saves you when the store was wiped or when several service instances share an ERP but not a store.

Asynchronous ERPs

If order creation is slow (minutes), accept synchronously — validate, write a queued record with a provisional number, return `success: true` — and let a worker finish. `success: true` must mean "this will be booked"; if it later cannot be, that is an exception path for humans, not something AIOTIC will retry.

PYTHON SDK

Sync engine

`aiotic.sync` sends master data to AIOTIC **only when something changed**, in the right order, in parallel, and reports what happened.

Concepts

Type	Role
`ChangeEvent`	one change in AIOTIC terms: `kind` (customer / product / mapping), `op` (upsert / delete), `key`, `data`. Constructors: `customer_upsert(number, **fields)`, `customer_delete(number)`, `product_upsert(item_number, language_code, description=…, remark=None)`, `product_delete(…)`, `mapping_upsert(customer_number, customer_item_number, item_number=…, language_code=…)`, `mapping_delete(…)`
`StateStore`	remembers the fingerprint last sent per key (+ watermarks). `InMemoryStateStore`, `HashStateStore(path)` (SQLite), or your own
`SyncEngine(client, state, concurrency=4, dry_run=False)`	applies events; skips unchanged; FK ordering; parallelism; report
`SyncReport`	`sent`, `deleted`, `skipped_unchanged`, `failed`, `errors[]`, `duration`
`PollingChangeSource(name, fetch_changed_since, engine, interval)`	watermark-based poller for ERPs without events

API

```
engine = SyncEngine(client, state=HashStateStore("sync-state.db"), concurrency=6)

engine.apply(event, force=False) -> bool            # one event; True when a request was made
engine.apply_many(events, force=False) -> SyncReport # batch: upserts customers→products→mappings, deletes reversed
engine.reconcile(customers=…, products=…, mappings=…, delete_missing=True) -> SyncReport
engine.bootstrap_state_from_aiotic() -> int         # seed fingerprints from what AIOTIC holds
```

- Unchanged fingerprint → skipped, no request.
- `404` on an upsert → reported as "referenced customer/product missing (upsert those first)".
- `404` on a delete → counted as deleted (already gone).
- Other API errors → `failed` with the message; the fingerprint is **not** updated, so the next run retries.
- `dry_run=True` logs what would be sent.

Mapping ERP records

Keep the mapping in one function per kind so events and reconciliation share it:

```
def customer_event(c) -> ChangeEvent:
    if c.blocked:
        return ChangeEvent.customer_delete(c.no)
    return ChangeEvent.customer_upsert(c.no, name=c.name, address=c.address, postal_code=c.post_code,
city=c.city,
                                       vat_number=c.vat_registration_no, email=c.email,
phone_number=c.phone,
                                       contact_person=c.contact, coc_number=c.registration_no,
home_page=c.home_page)
```

Fields you do not have are simply omitted (`null`). Empty strings become `null` in the fingerprint, so a value that flips between `""` and `None` does not cause churn.

Event sources

```
# webhook / outbox → generic JSON → engine
from aiotic.webhooks import parse_change_events
engine.apply_many(parse_change_events(payload))

# polling
src = PollingChangeSource("customers", fetch_changed_since, engine, interval=300)
threading.Thread(target=src.run_forever, daemon=True).start()
```

The watermark advances only after a batch with zero failures.

Reconciliation

`reconcile()` expects the **complete** current data set per kind you pass (generators are fine — 200 000 products stream through). It sends differences and, with `delete_missing=True`, deletes keys it sent earlier that are no longer present. Pass only the kinds you have complete data for.

Before the first reconcile on a tenant that already has data: `engine.bootstrap_state_from_aiotic()` (or `aiotic sync bootstrap-state`).

Concurrency and keys

Use 4–8 workers. Configure `AIOTIC_SYNC_API_KEY`; the client routes master-data calls to it automatically. The client's rate limiter is shared by all workers.

Observability

Log the `SyncReport` per run; alert on `failed > 0` for two consecutive runs. Keep `errors[]` — they name the record and the reason.

PYTHON SDK

Webhooks & receivers

Three inbound HTTP surfaces live in an integration service. `aiotic.receive` and `aiotic.webhooks` implement them framework-free, with FastAPI routers on top.

Surface	Direction	SDK
ERP receive endpoint	AIOTIC → you	`ErpReceiver`, `create_receive_router`
Processing webhook	AIOTIC → you	`ProcessingWebhookReceiver`, `create_webhook_routers(processing=…)`
ERP change events	your ERP → you	`parse_change_events`, `create_webhook_routers(on_change_events=…)`

ERP receive endpoint

```
from aiotic.receive import ErpReceiver, SqliteStore, create_receive_router

receiver = ErpReceiver(erp, api_key=RECEIVE_KEY, pipeline=pipeline, store=SqliteStore("integration.db"),
                       on_accepted=lambda req, result, verdict: metrics.inc("booked"),
                       on_rejected=lambda req, resp, verdict: metrics.inc("rejected", reason=resp.error))
app.include_router(create_receive_router(receiver, path="/aiotic/orders"))
```

`handle(body, api_key_header=…) -> ReceiveOutcome` does everything in order: key check (constant time) → parse → idempotency (store, then adapter lookup) → pipeline → `create_sales_order` → remember → respond. Business problems never raise; unexpected exceptions become `success: false` with a generic message and are logged with the traceback.

Using another framework:

```
# Flask
@app.post("/aiotic/orders")
def receive():
    out = receiver.handle(request.get_data(), api_key_header=request.headers.get("X-API-KEY"))
    return jsonify(out.response.model_dump(exclude_none=True)), out.http_status
```

Processing webhook

```
from aiotic.webhooks import ProcessingWebhookReceiver, create_webhook_routers

hook = ProcessingWebhookReceiver(WEBHOOK_KEY, handler=lambda req: drafts.create(req.request_id,
req.purchase_order))
app.include_router(create_webhook_routers(processing=hook))          # POST /aiotic/processing
```

The handler's exceptions are logged and **acknowledged anyway** — AIOTIC does not retry, so failing loudly would only lose the signal.

ERP change events

```
app.include_router(create_webhook_routers(on_change_events=engine.apply_many, erp_events_key=EVENTS_KEY))
# POST /erp/events
```

Accepts the generic format described in Event-driven sync. Protect it with its own key (`X-API-KEY`) — it is *your* ERP calling, not AIOTIC.

Signed webhooks (future)

`verify_hmac_signature(secret, body, header)` implements the scheme in the proposal (`X-AIOTIC-Signature: t=<unix>,v1=<hmac-sha256>`), with a 5-minute tolerance. Nothing in AIOTIC sends it yet; it is here so your endpoint can be ready.

`build_app()` wires all three

`aiotic.service.build_app(erp=…, pipeline=…, store=…, sync_engine=…)` mounts the receive endpoint, the processing webhook (when `AIOTIC_WEBHOOK_KEY` is set), the change-event endpoint (protected with `AIOTIC_ERP_RECEIVE_KEY` by default — give it its own key in production) and `/healthz` .

PYTHON SDK

Order watcher

`aiotic.watch.OrderWatcher` turns polling into events. It reads the first page(s) of `GET /order_status/list`, compares each order's status with the last one it stored, and calls your callback with a `Transition` — exactly once per change.

```
from aiotic.watch import OrderWatcher, SqliteWatchState, Transition

def on_transition(t: Transition) -> None:
    # t.request_id, t.from_status (None on first sight), t.to, t.order (full OrderStatus)
    match t.to:
        case "PROCESSED": queue.put(("book", t.request_id))
        case "ATTENTION": tickets.create(t.order)
        case "SENT":      erp.mark_exported(t.order.erp_ref)

watcher = OrderWatcher(client, on_transition=on_transition, state=SqliteWatchState("watch.db"),
                       interval=20, page_size=200, pages=2, only=None, seed_silently=True)
watcher.run_forever()          # or watcher.run_once() from your own scheduler
```

Option	Meaning
`state`	where last statuses live: `InMemoryWatchState` (lost on restart → replays history) or `SqliteWatchState(path)`
`interval`	seconds between polls
`pages` × `page_size`	how deep to look each poll; deeper than your busiest 2 hours is enough
`only`	restrict callbacks to certain target statuses
`seed_silently`	on the very first run, learn current statuses without firing (avoid a storm on a fresh state file)

Callback exceptions are logged and do not stop the loop; the transition is still recorded (it will not fire again), so make callbacks idempotent or push to a queue.

`aiotic orders watch` runs the watcher and prints one JSON line per transition:

```
{"request_id": "7f0c…", "from": "PROCESSING", "to": "PROCESSED", "order_number": "PO-4711", "erp_ref":
null, "timestamp": "2026-03-14T10:31:02"}
```

Cost

Two small requests every 20 s ≈ 8 600 requests/day, independent of order volume. Fine for any tenant.

Limits

Orders that scroll past the pages you look at before a transition happens are missed; raise `pages` or run a nightly full walk (`client.orders.iter_all()`) for reconciliation. Signed status webhooks would remove this — see the proposal.

PYTHON SDK

CLI

Installed with `pip install "aiotic-sdk[cli]"`. Reads `AIOTIC_*` from the environment or `.env`.

Command	What it does
`aiotic init`	Write `.env` (with a generated receive-endpoint key) and `service.py`
`aiotic doctor`	Check base URL, keys, `/healthcheck`, `/system-status`, and that customers/products/mappings exist; exit 1 on problems
`aiotic orders list [--size 20] [--page 1]`	Newest orders with status, PO number, customer, `erp_ref`
`aiotic orders get <request_id>`	Full JSON of one order
`aiotic orders upload <files…> [--request-id] [--no-wait]`	Upload one order (all files = one order) and wait for it to land
`aiotic orders send <request_id>`	`POST /erp/send`; prints the ERP reference or the ERP's error
`aiotic orders watch [--interval 15] [--state watch.db]`	Print status transitions as JSON lines
`aiotic sync customers|products|mappings --from file.csv|.json [--dry-run] [--force] [--concurrency 4]`	Upsert from a file; only changed records are sent
`aiotic sync reconcile [--customers a.csv] [--products b.csv] [--mappings c.csv] [--no-delete-missing]`	Diff-based safety net
`aiotic sync bootstrap-state`	Seed fingerprints from what AIOTIC already holds
`aiotic serve [--port 9000] [--reload]`	Run `service.py` (or the built-in demo service)
`aiotic mock [--port 8080] [--erp-url … --erp-key …] [--processing-speed 3]`	Run a mock AIOTIC tenant
`aiotic version`	SDK version

CSV columns follow the API field names (`number,name,address,postal_code,city,vat_number,email,phone_number,contact_person,coc_number,home_page`; `item_number,language_code,description,remark`; `customer_number,customer_item_number,item_number,language_code`). Comma or semicolon separated, UTF-8.

Exit codes

`0` success · `1` a check or a sync record failed · `2` configuration missing.

Automation examples

```
# nightly reconciliation from ERP exports (cron)
0 2 * * * cd /srv/aiotic && ./export.sh && aiotic sync reconcile --customers out/customers.csv --products
out/products.csv --mappings out/mappings.csv >> sync.log 2>&1

# readiness check in a deployment pipeline
aiotic doctor || exit 1
```

PYTHON SDK

Mock AIOTIC server

A local stand-in for a tenant, for development and CI. It implements the public API with realistic behaviour and **really calls your ERP receive endpoint** on `POST /erp/send`.

```
aiotic mock                     # http://localhost:8080
python -m aiotic.mock           # same, without the CLI extra
docker compose -f mock/docker-compose.yml up     # mock + demo integration service
```

Setting	Env / flag	Default
Keys	fixed	`mock-integration-key` (all endpoints), `mock-sync-key` (master data only)
ERP receive URL / key	`MOCK_ERP_URL`, `MOCK_ERP_KEY` or `--erp-url/--erp-key` or `POST /_mock/config`	none → `/erp/send` answers `503` like a tenant without ERP config
Processing webhook	`MOCK_WEBHOOK_URL`, `MOCK_WEBHOOK_KEY`	off
Processing time	`MOCK_PROCESSING_SECONDS` / `--processing-speed`	3 s

Behaviour

- Uploads move `QUEUED` → `PROCESSING` → `PROCESSED` with a realistic extracted order (`result`), PO number taken from the file name (`PO-4711.pdf` → `PO-4711`).
- File name contains `attention` → `ATTENTION` with one unresolved line (`article_number: null`, `customer_item_number: "LT-ART-999"`).
- File name contains `fail` → `FAILED` with `last_error`; `POST /order/retry` creates a new order that succeeds.
- Raw `.eml` containing "quotation"/"offerte" → `400 not_a_purchase_order` + entry in `/rejected/list`; containing two `Subject:` lines or `SPLIT` → split into two child orders with an `email_group_id`.
- Master data: seeded with customer `58931`, products `PROD-001`, `PROD-002`, `620206_01` (`nl`) and one mapping; a mapping needs an existing customer and product (`404` otherwise); the sync key is rejected on non-master-data paths (`401`).
- `/erp/send`: status gate (`409`), `SENDING` lock, real `POST` to the ERP URL with `X-API-KEY`, `success` interpretation, `422` / `500` and rollback, `erp_ref` on success.

Mock-only helpers

Endpoint	Purpose
`POST /_mock/config` `{erp_url, erp_key, webhook_url, webhook_key}`	set targets at runtime
`POST /_mock/orders/{id}/status` `{"status": "MODIFIED"}`	force a status the API cannot set (e.g. `MODIFIED`, `CANCELED`) to test your handling

In tests

A pytest fixture that starts the mock in a background thread and hands you a client:

```
import asyncio, threading, pytest, uvicorn
from aiotic import AioticClient
from aiotic.mock import create_mock_app

@pytest.fixture(scope="session")
def client():
    server = uvicorn.Server(uvicorn.Config(create_mock_app(processing_seconds=0.3), host="127.0.0.1",
port=0, log_level="warning"))
    server.install_signal_handlers = lambda: None
    threading.Thread(target=server.run, daemon=True).start()
    while not server.started:
        asyncio.run(asyncio.sleep(0.05))
    port = server.servers[0].sockets[0].getsockname()[1]
    with AioticClient(base_url=f"http://127.0.0.1:{port}", api_key="mock-integration-key", rate_limit=0)
as c:
        yield c
    server.should_exit = True
```

For pure unit tests of the receive endpoint you do not need the mock at all — call `ErpReceiver.handle()` directly.

Differences from a real tenant

No AI: the extraction is canned. No mailbox. No management endpoints. No operator app. Timing is instant-ish. Everything about the *contracts* — paths, headers, status codes, JSON shapes, the ERP call — matches the public API.

API REFERENCE

API reference

Every public endpoint of an AIOTIC tenant, generated from the OpenAPI document. Base URL: `https://<tenant>.aiotic.ai`. Authentication: `X-API-Key` header (details). Errors: `{"detail": …}` (details).

Which key do you have? A standard integration runs entirely on the **sync key**: the master-data endpoints under `/customer/*`, `/product/*` and `/customer-product/*`, plus the receive endpoint and webhook that AIOTIC calls on your side. The endpoints marked *integration key* exist for headless integrations and custom review UIs; that key is issued per project by the AIOTIC team.

Two calls go the **other way** — from AIOTIC to you — and are documented under Outbound: webhooks.

Method	Path	Summary	Auth
POST	/order/upload	Upload document(s) as one order	integration key
POST	/order/raw/upload	Upload a raw e-mail (.eml)	integration key
POST	/order/raw/classify	Classify a raw e-mail without processing it	integration key
GET	/order/group/{email_group_id}	Get all orders from one source e-mail	integration key
GET	/order/{request_id}/{filename}	Download an order file	integration key
GET	/order/{request_id}/{filename}/preview	Preview an order file	integration key
POST	/order/retry/{request_id}	Retry a failed order	integration key
GET	/order_status/{request_id}	Get order status and extracted data	integration key
GET	/order_status/list	List orders (paginated, newest first)	integration key
POST	/erp/send/{request_id}	Send an order to the ERP receive endpoint	integration key
GET	/rejected/list	List rejected e-mails	integration key
GET	/rejected/{request_id}	Get one rejected e-mail	integration key
POST	/rejected/{request_id}/reprocess	Reprocess a rejected e-mail as a purchase order	integration key
GET	/customer/list	List customers	integration or sync key
GET	/customer/search/{query}	Search customers (semantic)	integration or sync key
GET	/customer/{customer_number}	Get a customer	integration or sync key
PUT	/customer/{customer_number}	Create or update a customer (upsert)	integration or sync key
DELETE	/customer/{customer_number}	Delete a customer	integration or sync key
GET	/product/list	List products	integration or sync key
GET	/product/{product_number}/{language_code}	Get a product	integration or sync key

Method	Path	Summary	Auth
PUT	/product/{product_number}/{language_code}	Create or update a product (upsert)	integration or sync key
DELETE	/product/{product_number}/{language_code}	Delete a product	integration or sync key
GET	/customer-product/list	List customer item mappings	integration or sync key
GET	/customer-product/{customer_number}/{customer_item_number}	Get a customer item mapping	integration or sync key
PUT	/customer-product/{customer_number}/{customer_item_number}	Create or update a customer item mapping (upsert)	integration or sync key
DELETE	/customer-product/{customer_number}/{customer_item_number}	Delete a customer item mapping	integration or sync key
POST	/email-watcher/fetch-all	Fetch all unread e-mails now	integration key
POST	*your URL* (erpReceiveOrder)	ERP receive endpoint — implemented by you	your key
POST	*your URL* (processingCompleted)	Processing webhook (optional) — implemented by you	your key

Machine-readable Download the spec: `/openapi.yaml`. Point your AI assistant at `/llms.txt` or use the MCP server.

API REFERENCE

Health & status

Unauthenticated endpoints for liveness and platform status.

API REFERENCE

Orders — intake

Submit documents or raw e-mails for extraction. All uploads return a `request_id` immediately; processing runs in the background — continue with the status endpoints. Guide: Submitting documents.

Upload document(s) as one order

POST `/order/upload` — auth: integration key

Upload one or more files (PDF, JPG, PNG, TXT, MD) that together form **one** purchase order. Files are stored and queued; extraction runs in the background.

- Provide your own `request_id` (UUID v4) to make the call idempotent and to correlate the order in your own systems.
- Any additional multipart form field is stored as `metadata` on the order and echoed back by the status endpoints — use it for your own references.
- Multiple files are never split into multiple orders on this endpoint. Use `/order/raw/upload` for e-mails that may contain several orders.

Request body — `multipart/form-data`

Field	Type	Required	Description
`files`	array of string	yes	
`request_id`	string | null		

Responses

Status	Description	Body
`200`	Files accepted and queued	OrderUploadResponse
`400`	Invalid request (malformed UUID, unsupported file, …)	ErrorResponse
`401`	Missing or invalid API key	ErrorResponse
`404`	Resource not found	ErrorResponse
`415`	Unsupported media type	ErrorResponse
`422`	Validation Error	HTTPValidationError
`503`	Tenant not initialised yet (no configuration deployed)	ErrorResponse

```
<span class="tab">curl</span>
curl -X POST "https://acme.aiotic.ai/order/upload" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
up = client.orders.upload(["PO-4711.pdf"], request_id=uuid.uuid4(), metadata={"my_ref": "TICKET-1"})
status = client.orders.wait(up.request_id)
```

```
<span class="tab">Response</span>
{
  "request_id": "550e8400-e29b-41d4-a716-446655440000",
  "split": false
}
```

Upload a raw e-mail (.eml)

POST `/order/raw/upload` — auth: integration key

Submit a complete e-mail (`.eml`). AIOTIC classifies it; if it is a purchase order the attachments and body are extracted. If the tenant has order splitting enabled and the e-mail carries several orders, the response has `split: true` and one child order per detected order — **poll the children**, not the source `request_id` .

A non-purchase-order e-mail is rejected with `400` and a structured `detail` object (`error: not_a_purchase_order` , the detected `category` , …) and is recorded in the rejected-e-mails list so an operator can override the decision.

Request body — `multipart/form-data`

Field	Type	Required	Description
`file`	string	yes	
`request_id`	string | null		

Responses

Status	Description	Body
`200`	Accepted (single order or split children)	OrderUploadResponse
`400`	Invalid file, or the e-mail is not a purchase order (structured `detail`)	RawUploadRejection
`401`	Missing or invalid API key	ErrorResponse
`404`	Resource not found	ErrorResponse
`422`	A split was predicted but no child order could be prepared	ErrorResponse
`503`	Classification not configured on this tenant yet	ErrorResponse

```
<span class="tab">curl</span>
curl -X POST "https://acme.aiotic.ai/order/raw/upload" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
up = client.orders.upload_raw_email("mail.eml")       # AioticValidationError with structured detail for non-orders
for rid in up.request_ids:                             # children when split
    client.orders.wait(rid)
```

```
<span class="tab">Response</span>
{
  "request_id": "550e8400-e29b-41d4-a716-446655440000",
  "split": false
}
```

Classify a raw e-mail without processing it

POST `/order/raw/classify` — auth: integration key

Returns the category the classifier would assign (`purchase_order` , `invoice` , `quotation` , …). Nothing is stored or queued.

Request body — `multipart/form-data`

Field	Type	Required	Description
`file`	string	yes	

Responses

Status	Description	Body
`200`	Successful Response	EmailClassificationResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X POST "https://acme.aiotic.ai/order/raw/classify" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.orders.classify_raw_email("mail.eml").category   # 'purchase_order' | 'quotation' | …
```

Get all orders from one source e-mail

GET `/order/group/{email_group_id}` — auth: integration key

Aggregate view of a split e-mail — the correlation id, the message id and every child order's status.

Parameters

Name	In	Type	Required	Description
`email_group_id`	path	string	yes	Correlation id shared by all child orders of one split e-mail

Responses

Status	Description	Body
`200`	Group with child orders	OrderGroup
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/order/group/<email_group_id>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
group = client.orders.group(email_group_id)      # .orders: list[OrderStatus]
```

Download an order file

GET `/order/{request_id}/{filename}` — auth: integration key

Returns one file belonging to the order: an original upload (by its file name) or a generated artifact such as `latest_result.json` (the extracted purchase order). The `Content-Type` follows the file extension.

Parameters

Name	In	Type	Required	Description
`request_id`	path	string	yes	The order's id (UUID) as returned by the upload endpoints
`filename`	path	string	yes	File name as listed in the order's `attachments`, or `latest_result.json`

Responses

Status	Description	Body
`200`	Successful Response	object
`404`	Resource not found	ErrorResponse

Status	Description	Body
422	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/order/<request_id>/<filename>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
pdf = client.orders.download_file(request_id, "PO-4711.pdf")
result = client.orders.download_file(request_id, "latest_result.json")
```

Preview an order file

GET `/order/{request_id}/{filename}/preview` — auth: integration key

Same as the download endpoint but with an inline `Content-Disposition`, for showing a PDF or image inside your own review UI.

Parameters

Name	In	Type	Required	Description
request_id	path	string	yes	The order's id (UUID) as returned by the upload endpoints
filename	path	string	yes	File name as listed in the order's `attachments`, or `latest_result.json`

Responses

Status	Description	Body
200	Successful Response	object
404	Resource not found	ErrorResponse
422	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/order/<request_id>/<filename>/preview" -H "X-API-Key:
$AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
data = client.orders.download_file(request_id, "PO-4711.pdf", preview=True)
```

Retry a failed order

POST `/order/retry/{request_id}` — auth: integration key

Re-runs processing for an order in `FAILED` status by creating a **new** order with the same files. The original moves to `REPROCESSED`; the response carries the new `request_id` — track that one from now on.

Parameters

Name	In	Type	Required	Description
request_id	path	string	yes	The order's id (UUID) as returned by the upload endpoints

Request body — `application/x-www-form-urlencoded`

Field	Type	Required	Description
hil_prompt	string | null		

Responses

Status	Description	Body
`200`	New order queued	OrderUploadResponse
`400`	Order is not in `FAILED` status, or has no files to retry	ErrorResponse
`401`	Missing or invalid API key	ErrorResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X POST "https://acme.aiotic.ai/order/retry/<request_id>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
new = client.orders.retry(request_id)            # track new.request_id from now on
```

API REFERENCE

Orders — status & files

Read processing status and the extracted purchase order, list orders, download files. Guide: Polling & notifications · Order lifecycle.

Get order status and extracted data

GET `/order_status/{request_id}` — auth: integration key

The current lifecycle status of an order plus, once processing finished, the extracted purchase order in `result`. Also carries `erp_ref` after a successful ERP send, `last_error` / `retry_count` / `next_retry_at` for transient failures, and the `email_group_id` / `order_label` when the order came from a split e-mail.

Parameters

Name	In	Type	Required	Description
`request_id`	path	string	yes	The order's id (UUID) as returned by the upload endpoints

Responses

Status	Description	Body
`200`	Order status	OrderStatus
`400`	Invalid request (malformed UUID, unsupported file, …)	ErrorResponse
`401`	Missing or invalid API key	ErrorResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/order_status/<request_id>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
status = client.orders.get(request_id)
status.status, status.result, status.erp_ref
```

Response


```
{
  "request_id": "550e8400-e29b-41d4-a716-446655440000",
  "status": "PROCESSED",
  "timestamp": "2026-03-14T10:30:00",
  "attachments": {
    "PO-4711.pdf": {
      "size": 125840,
      "mime_type": "application/pdf"
    }
  },
  "metadata": {
    "source": "api",
    "my_reference": "TICKET-8812"
  },
  "result": {
    "order_number": "PO-4711",
    "order_date": "2026-03-14",
    "delivery_date": "2026-03-21",
    "currency": "EUR",
    "total_price": 123.4,
    "supplier": {
      "company": "Acme Supplies BV",
      "contact_person": null,
      "email": null,
      "address": {
        "street": "Industrieweg 5",
        "postal_code": "1234 AB",
        "city": "Amsterdam",
        "country": "NL"
      }
    },
    "customer": {
      "customer_id": "58931",
      "company": "LUMITECH INSTALLATIES",
      "contact_person": "J. de Boer",
      "email": "info@lumitech.example",
      "phone": null,
      "branch": null,
      "vat_id": "NL001234567B01",
      "iban": null,
      "bic": null,
      "address": {
        "street": "Ambachtsweg 12",
        "postal_code": "7327 AA",
        "city": "Apeldoorn",
        "country": "NL"
      }
    },
    "shipping_details": null,
    "items": [
      {
        "article_number": "PROD-001",
        "customer_item_number": "LT-ART-001",
        "description": "LED Driver 48V",
        "quantity": 10,
        "quantity_state": "Valid",
        "unit": "ST",
        "price": 12.34,
        "currency": "EUR",
        "line_total": 123.4
      }
    ],
    "additional_information": null
  },
```

```
  "state": null,
  "erp_ref": null,
  "email_group_id": null,
  "order_label": "PO-4711",
  "retry_count": 0,
  "next_retry_at": null,
  "last_error": null
}
```

List orders (paginated, newest first)

GET `/order_status/list` — auth: integration key

Paginated list of orders with their status. There is no server-side filter or `updated_since` parameter today — poll the first page(s) and track transitions locally (the Python SDK's `OrderWatcher` does exactly this).

Parameters

Name	In	Type	Required	Description
`page`	query	integer		Page number Default `1`. Min 1.
`size`	query	integer		Items per page Default `100`. Min 1. Max 1000.

Responses

Status	Description	Body
`200`	Successful Response	OrderListResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/order_status/list?page=1&size=100" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
page = client.orders.list(page=1, size=100)
for order in client.orders.iter_all(size=200, max_pages=5): ...
```

API REFERENCE

ERP hand-off

Trigger the send of a reviewed order to your ERP receive endpoint. The contract of that endpoint is under Outbound: webhooks. Guide: Sending to the ERP.

Send an order to the ERP receive endpoint

POST `/erp/send/{request_id}` — auth: integration key

Hands a reviewed order to your ERP by calling the receive endpoint configured for the tenant (see `webhooks` → `erpReceiveOrder`). The call is synchronous: AIOTIC waits for your response (timeout configurable, default 30 s) and returns its outcome.

- Only orders in `PROCESSED`, `MODIFIED` or `ATTENTION` can be sent. Sending an `ATTENTION` order is an explicit override of the review flag.
- The order is atomically locked (`SENDING`) — concurrent sends of the same order get `409`. On success the status becomes `SENT` and `erp_ref` is stored.
- On any failure the status rolls back to what it was, so the send can be retried.
- Operator corrections made in the AIOTIC app are merged into the payload before it leaves AIOTIC.

Parameters

Name	In	Type	Required	Description
`request_id`	path	string	yes	The order's id (UUID) as returned by the upload endpoints

Responses

Status	Description	Body
`200`	Your ERP accepted the order	ErpSendResponse
`400`	Invalid request (malformed UUID, unsupported file, …)	ErrorResponse
`401`	Missing or invalid API key	ErrorResponse
`404`	Resource not found	ErrorResponse
`409`	Order is not in a sendable status, or another send is in progress	ErrorResponse
`422`	Your ERP answered `success: false`; `detail` carries your `error` text	ErrorResponse
`500`	Your ERP returned a non-JSON body, timed out or was unreachable (status rolled back)	ErrorResponse
`503`	No ERP endpoint configured for this tenant	ErrorResponse

```
<span class="tab">curl</span>
curl -X POST "https://acme.aiotic.ai/erp/send/<request_id>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
from aiotic import AioticErpRejectedError
try:
    res = client.erp.send(request_id)            # res.erp_order_number
except AioticErpRejectedError as e:
    print(e.erp_error)                           # your endpoint's error text
```

```
<span class="tab">Response</span>
{
  "success": true,
  "request_id": "550e8400-e29b-41d4-a716-446655440000",
  "data": {
    "success": true,
    "order_number": "SO-2026-00981"
  }
}
```

API REFERENCE

Rejected e-mails

E-mails the classifier did not treat as purchase orders. Guide: Rejected e-mails.

List rejected e-mails

GET `/rejected/list` — auth: integration key

E-mails the classifier did not treat as purchase orders, newest first. `status=pending` (default) lists the ones still awaiting a decision; `status=overridden` those already reprocessed.

Parameters

Name	In	Type	Required	Description
`page`	query	integer		Page number Default `1`. Min 1.
`size`	query	integer		Items per page Default `100`. Min 1. Max 1000.
`status`	query	string		Rejection-triage status filter Default `pending`.

Responses

Status	Description	Body
`200`	Successful Response	RejectedEmailListResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/rejected/list?page=1&size=100&status=pending" -H "X-API-Key:
$AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.rejected.list(status="pending").items
```

Get one rejected e-mail

GET `/rejected/{request_id}` — auth: integration key

Get a single rejected (non-purchase-order) classified email by request ID.

This is the rejected-email detail endpoint, not a generic classified-email lookup: a `purchase_order` row is reported as 404 (it does not belong to this surface).

Parameters

Name	In	Type	Required	Description
`request_id`	path	string	yes	The order's id (UUID) as returned by the upload endpoints

Responses

Status	Description	Body
`200`	Successful Response	ClassifiedEmail
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/rejected/<request_id>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.rejected.get(request_id)
```

Reprocess a rejected e-mail as a purchase order

POST `/rejected/{request_id}/reprocess` — auth: integration key

Overrides the classifier for one e-mail and runs it through normal processing under the **same** `request_id`. The original classification is preserved for audit.

Parameters

Name	In	Type	Required	Description
`request_id`	path	string	yes	The order's id (UUID) as returned by the upload endpoints

Responses

Status	Description	Body
`200`	Successful Response	ReprocessResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X POST "https://acme.aiotic.ai/rejected/<request_id>/reprocess" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.rejected.reprocess(request_id)            # then poll client.orders.get(request_id)
```

API REFERENCE

Customers

Your debtors. Keyed on **your** customer number, which AIOTIC returns as `customer.customer_id` on orders. Accepts the sync key. Guide: Reference data · Event-driven sync.

List customers

GET `/customer/list` — auth: integration or sync key

List all customers with pagination.

Parameters

Name	In	Type	Required	Description
`page`	query	integer		Page number Default `1`. Min 1.
`size`	query	integer		Items per page Default `100`. Min 1. Max 1000.

Responses

Status	Description	Body
`200`	Successful Response	CustomerListResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/customer/list?page=1&size=100" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
for c in client.customers.iter_all(): ...
```

Search customers (semantic)

GET `/customer/search/{query}` — auth: integration or sync key

Fuzzy search over name, address and contact fields, comparable to how AIOTIC matches the sender of an order to your customer records. Handy to check how well your master data resolves.

Parameters

Name	In	Type	Required	Description
`query`	path	string	yes	Search query text
`top_k`	query	integer		Top items by similarity score Default `10`. Min 1. Max 1000.

Responses

Status	Description	Body
`200`	Successful Response	CustomerSearchResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/customer/search/<query>?top_k=10" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.customers.search("lumitech apeldoorn", top_k=3).items
```

Get a customer

GET `/customer/{customer_number}` — auth: integration or sync key

Get a specific customer by ID.

Parameters

Name	In	Type	Required	Description
`customer_number`	path	string	yes	Unique customer number

Responses

Status	Description	Body
`200`	Successful Response	Customer
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/customer/<customer_number>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.customers.get("58931")
```

Create or update a customer (upsert)

PUT `/customer/{customer_number}` — auth: integration or sync key

Idempotent upsert keyed on **your** customer (debtor) number. The number is what AIOTIC returns as `customer.customer_id` on every order it identifies for this customer. Populate as many fields as you have — name, address, VAT, e-mail and phone all improve identification. Response status is `202 Accepted`.

Parameters

Name	In	Type	Required	Description
`customer_number`	path	string	yes	Unique customer number

Request body — `application/json`

Fields accepted when creating or updating a customer. All optional; the more you fill, the better identification works.

Field	Type	Required	Description
`id`	string | null		Unique customer identifier (UUID, auto-generated if not provided)
`name`	string | null		Company name
`postal_code`	string | null		Postal code
`city`	string | null		City
`address`	string | null		Address
`contact_person`	string | null		Contact person
`phone_number`	string | null		Phone number
`vat_number`	string | null		BTW (VAT) number

Field	Type	Required	Description
`email`	string | null		Customer email
`coc_number`	string | null		Chamber of Commerce number
`home_page`	string | null		Company website URL
`similarity`	number | null		Similarity score

Responses

Status	Description	Body
`202`	Successful Response	Customer
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X PUT "https://acme.aiotic.ai/customer/<customer_number>" -H "X-API-Key: $AIOTIC_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"id":"550e8400-e29b-41d4-a716-
446655440000","name":"string","postal_code":"string","city":"string","address":"string","contact_person":
"string","phone_number":"string","vat_number":"string","email":"string","coc_number":"string","home_page"
:"string","similarity":1}'
```

```
<span class="tab">Python SDK</span>
client.customers.upsert("58931", {"name": "LUMITECH INSTALLATIES", "city": "Apeldoorn", "vat_number":
"NL001234567B01"})
# or, only-when-changed via the sync engine:
engine.apply(ChangeEvent.customer_upsert("58931", name="LUMITECH INSTALLATIES", city="Apeldoorn"))
```

Delete a customer

DELETE `/customer/{customer_number}` — auth: integration or sync key

Remove a customer. Prefer deleting over leaving inactive records — stale records can be matched by mistake. (Renaming a record with a "formerly / do not use" marker archives it instead.)

Parameters

Name	In	Type	Required	Description
`customer_number`	path	string	yes	Unique customer number

Responses

Status	Description	Body
`204`	Successful Response	—
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X DELETE "https://acme.aiotic.ai/customer/<customer_number>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.customers.delete("58931")
```

API REFERENCE

Products

Your article catalog — the list of valid article numbers. Composite key `item_number` + `language_code`. Accepts the sync key.

List products

GET `/product/list` — auth: integration or sync key

List all products with pagination.

Parameters

Name	In	Type	Required	Description
`page`	query	integer		Page number Default `1`. Min 1.
`size`	query	integer		Items per page Default `100`. Min 1. Max 1000.
`language_code`	query	string | null		Filter by language code

Responses

Status	Description	Body
`200`	Successful Response	ProductListResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/product/list?page=1&size=100&language_code=<language_code>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
for p in client.products.iter_all(language_code="nl"): ...
```

Get a product

GET `/product/{product_number}/{language_code}` — auth: integration or sync key

Get a specific product by its primary key.

Parameters

Name	In	Type	Required	Description
`product_number`	path	string	yes	Product item number
`language_code`	path	string	yes	Product language code

Responses

Status	Description	Body
`200`	Successful Response	Product
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/product/<product_number>/<language_code>" -H "X-API-Key:
$AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.products.get("PROD-001", "nl")
```

Create or update a product (upsert)

PUT `/product/{product_number}/{language_code}` — auth: integration or sync key

Idempotent upsert keyed on your article number **and** a language code (`nl`, `de`, `en`, …). Create one record per language if your catalog is multilingual. The product table is AIOTIC's list of *valid* article numbers: an order line whose article is not in it is flagged for review instead of being sent. Keep it complete.

Parameters

Name	In	Type	Required	Description
`product_number`	path	string	yes	Product item number
`language_code`	path	string	yes	Product language code

Request body — `application/json`

Fields accepted when creating or updating a product.

Field	Type	Required	Description
`description`	string	yes	Product description
`remark`	string | null		Additional remarks about the product
`created_at`	string		When this product was created in AIOTIC Format: `date-time`.

Responses

Status	Description	Body
`202`	Successful Response	Product
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X PUT "https://acme.aiotic.ai/product/<product_number>/<language_code>" -H "X-API-Key:
$AIOTIC_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"description":"string","remark":"string","created_at":"2026-03-14T10:30:00Z"}'
```

```
<span class="tab">Python SDK</span>
client.products.upsert("PROD-001", "nl", {"description": "LED Driver 48V 100W"})
engine.apply(ChangeEvent.product_upsert("PROD-001", "nl", description="LED Driver 48V 100W"))
```

Delete a product

DELETE `/product/{product_number}/{language_code}` — auth: integration or sync key

Delete a product.

Parameters

Name	In	Type	Required	Description
`product_number`	path	string	yes	Product item number
`language_code`	path	string	yes	Product language code

Responses

Status	Description	Body
`204`	Successful Response	—
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X DELETE "https://acme.aiotic.ai/product/<product_number>/<language_code>" -H "X-API-Key:
$AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.products.delete("PROD-001", "nl")
```

API REFERENCE

Customer item mappings

A customer's own article codes mapped to your article numbers. Composite key `customer_number` + `customer_item_number`; the referenced customer and product must exist. Accepts the sync key.

List customer item mappings

GET `/customer-product/list` — auth: integration or sync key

List customer product mappings with pagination and optional filters.

Parameters

Name	In	Type	Required	Description
`page`	query	integer		Page number Default `1`. Min 1.
`size`	query	integer		Items per page Default `100`. Min 1. Max 1000.
`customer_number`	query	string | null		Filter by customer number
`customer_item_number`	query	string | null		Filter by customer item number
`item_number`	query	string | null		Filter by supplier item number
`language_code`	query	string | null		Filter by language code

Responses

Status	Description	Body
`200`	Successful Response	CustomerProductListResponse
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/customer-product/list?page=1&size=100&customer_number=
<customer_number>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.customer_products.list(customer_number="58931").items
```

Get a customer item mapping

GET `/customer-product/{customer_number}/{customer_item_number}` — auth: integration or sync key

Get a specific customer product mapping by its composite key.

Parameters

Name	In	Type	Required	Description
`customer_number`	path	string	yes	Customer number
`customer_item_number`	path	string	yes	Customer's own item number

Responses

Status	Description	Body
`200`	Successful Response	CustomerProduct

Status	Description	Body
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X GET "https://acme.aiotic.ai/customer-product/<customer_number>/<customer_item_number>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.customer_products.get("58931", "LT-ART-001")
```

Create or update a customer item mapping (upsert)

PUT `/customer-product/{customer_number}/{customer_item_number}` — auth: integration or sync key

Tells AIOTIC that when customer `customer_number` orders **their** article `customer_item_number`, it means **your** article `item_number` (`language_code`). Both the customer and the product must already exist (`404` otherwise).

Parameters

Name	In	Type	Required	Description
`customer_number`	path	string	yes	Customer number
`customer_item_number`	path	string	yes	Customer's own item number

Request body — `application/json`

Fields accepted when creating or updating a customer item mapping.

Field	Type	Required	Description
`item_number`	string	yes	Supplier product item number
`language_code`	string	yes	Product language code
`created_at`	string		When this mapping was created in AIOTIC Format: `date-time` .

Responses

Status	Description	Body
`202`	Successful Response	CustomerProduct
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X PUT "https://acme.aiotic.ai/customer-product/<customer_number>/<customer_item_number>" -H "X-API-Key: $AIOTIC_API_KEY" \
  -H "Content-Type: application/json" \
  -d '{"item_number":"string","language_code":"string","created_at":"2026-03-14T10:30:00Z"}'
```

```
<span class="tab">Python SDK</span>
client.customer_products.upsert("58931", "LT-ART-001", {"item_number": "PROD-001", "language_code": "nl"})
engine.apply(ChangeEvent.mapping_upsert("58931", "LT-ART-001", item_number="PROD-001", language_code="nl"))
```

Delete a customer item mapping

DELETE `/customer-product/{customer_number}/{customer_item_number}` — auth: integration or sync key

Delete a customer product mapping.

Parameters

Name	In	Type	Required	Description
`customer_number`	path	string	yes	Customer number
`customer_item_number`	path	string	yes	Customer's own item number

Responses

Status	Description	Body
`204`	Successful Response	—
`404`	Resource not found	ErrorResponse
`422`	Validation Error	HTTPValidationError

```
<span class="tab">curl</span>
curl -X DELETE "https://acme.aiotic.ai/customer-product/<customer_number>/<customer_item_number>" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.customer_products.delete("58931", "LT-ART-001")
```

API REFERENCE

Mailbox

Operational trigger for tenants that receive purchase orders by e-mail. Integration key only.

Fetch all unread e-mails now

POST `/email-watcher/fetch-all` — auth: integration key

Operational trigger — asks the tenant mailbox watcher to fetch and classify all unread mails immediately instead of at the next interval.

Responses

Status	Description	Body
`200`	Successful Response	FetchAllEmailsResponse
`404`	Resource not found	ErrorResponse

```
<span class="tab">curl</span>
curl -X POST "https://acme.aiotic.ai/email-watcher/fetch-all" -H "X-API-Key: $AIOTIC_API_KEY"
```

```
<span class="tab">Python SDK</span>
client.mailbox.fetch_all()
```

Tenant configuration

The tenant's ERP receive endpoint (URL, key, timeout) and the processing webhook are configured in the AIOTIC app by a tenant admin, or by the AIOTIC team during onboarding. They are not part of the integration API: hand the URL and key to your AIOTIC contact, or enter them in the app's settings screen.

API REFERENCE

Outbound: webhooks

Calls **AIOTIC makes to you**. You implement these endpoints; AIOTIC authenticates with the key you provided. Guide: The ERP receive endpoint · The processing webhook.

ERP receive endpoint — implemented by you

AIOTIC calls this URL (configured per tenant as the *ERP API URL*) when an operator — or an API caller via `POST /erp/send/{request_id}` — sends a reviewed order to the ERP.

- Method `POST`, `Content-Type: application/json`, header `X-API-KEY: <key you gave us>`.
- Respond within the tenant's timeout (default 30 s) with a JSON body carrying `success`.
- `request_id` is stable across retries: make the handler idempotent (return the same `order_number` if you already created the order for this id).
- Return `success: false` with a clear `error` for business rejections (unknown article, blocked customer, …); the text is shown to the operator in AIOTIC.

Header `X-API-KEY` — The key you provided during onboarding

Request body — `application/json`

Body AIOTIC POSTs to your ERP receive endpoint.

Field	Type	Required	Description
`request_id`	string	yes	AIOTIC's order id (UUID). Stable across retries of the same send — use it as the idempotency key and store it on your sales order as external reference. Format: `uuid`.
`purchase_order`	ErpPurchaseOrder	yes	The reviewed purchase order with operator corrections applied. Every key below is always present; unknown values are `null`.

Example:

```
{
  "request_id": "550e8400-e29b-41d4-a716-446655440000",
  "purchase_order": {
    "order_number": "EB2500011645",
    "order_date": "2026-01-14",
    "delivery_date": "2026-02-01",
    "currency": "EUR",
    "total_price": 1234.56,
    "additional_information": "Please deliver before noon.",
    "supplier": {
      "company": "Acme Supplies BV",
      "contact_person": "P. Jansen",
      "email": "orders@acme.example",
      "address": {
        "street": "Industrieweg 5",
        "postal_code": "1234 AB",
        "city": "Amsterdam",
        "country": "NL"
      }
    },
    "customer": {
      "customer_id": "58931",
      "company": "LUMITECH INSTALLATIES",
      "contact_person": "J. de Boer",
      "email": "info@lumitech.example",
      "phone": "+31 55 123 4567",
      "vat_id": "NL001234567B01",
      "iban": null,
      "bic": null,
      "address": {
        "street": "Ambachtsweg 12",
        "postal_code": "7327 AA",
        "city": "Apeldoorn",
        "country": "NL"
      }
    },
    "shipping_details": {
      "recipient": {
        "company": "LUMITECH INSTALLATIES",
        "department": null,
        "contact_person": "J. de Boer",
        "email": null,
        "phone": null,
        "address": {
          "street": "Ambachtsweg 12",
          "postal_code": "7327 AA",
          "city": "Apeldoorn",
          "country": "NL"
        }
      },
      "special_instructions": null
    },
    "items": [
      {
        "article_number": "PROD-001",
        "description": "LED Driver 48V",
        "quantity": 10,
        "unit": "ST",
        "price": 12.34,
        "currency": "EUR",
        "line_total": 123.4
      }
    ]
```

```
  }
}
```

Every field of the hand-off payload

Every key listed here is **always present**; "nullable" means the value may be `null`.

Path	Type	Nullable	Description
`request_id`	string		AIOTIC's order id (UUID). Stable across retries of the same send — use it as the idempotency key and store it on your sales order as external reference.
`purchase_order`	object		The reviewed purchase order with operator corrections applied. Every key below is always present; unknown values are `null`.
`purchase_order.order_number`	string		The customer's purchase-order number as printed, with `/` replaced by `-`. Map it to the "your reference" / external document number of the sales order.
`purchase_order.order_date`	string	yes	Order date as an ISO `YYYY-MM-DD` string where the document allowed it; otherwise as printed.
`purchase_order.delivery_date`	string	yes	Requested delivery date (`YYYY-MM-DD`) or `null`. When the document stated a window it is already collapsed to a single date.
`purchase_order.currency`	string	yes	Currency code as printed on the document (e.g. `EUR`) or `null`.
`purchase_order.total_price`	number	yes	The document's printed total, which may include VAT, or the recalculated net sum when e-mail instructions changed lines. Informational — let your ERP compute totals.
`purchase_order.additional_information`	string	yes	Free text from the document and e-mail plus notes AIOTIC appends (e.g. the original printed total after a recalculation).
`purchase_order.supplier`	object	yes	Your own company as configured for the tenant — identical on every order, never read from the document. Safe to ignore.

Path	Type	Nullable	Description
`purchase_order.supplier.company`	string		Your company name as configured for the tenant.
`purchase_order.supplier.contact_person`	string	yes	Contact person configured for the tenant, if any.
`purchase_order.supplier.email`	string	yes	Order intake e-mail address of the tenant.
`purchase_order.supplier.address`	object		Your company address as configured.
`purchase_order.supplier.address.street`	string		Street and house number.
`purchase_order.supplier.address.postal_code`	string		Postal code.
`purchase_order.supplier.address.city`	string		City.
`purchase_order.supplier.address.country`	string	yes	Country code as printed, usually ISO 3166-1 alpha-2.
`purchase_order.customer`	object		The identified customer (debtor). `customer_id` is your customer number; the address block is canonicalised from your master record on a confident match.
`purchase_order.customer.customer_id`	string	yes	Your customer (debtor) number as synced via `PUT /customer/{number}`. `null` only when an operator explicitly sent an unidentified order.
`purchase_order.customer.company`	string	yes	Customer company name.
`purchase_order.customer.contact_person`	string	yes	Person who placed this order, as printed on the document (document-authoritative, not the master record).
`purchase_order.customer.email`	string	yes	Customer e-mail address.
`purchase_order.customer.phone`	string	yes	Customer phone number.
`purchase_order.customer.iban`	string	yes	Bank account when printed on the order; usually `null`.
`purchase_order.customer.bic`	string	yes	Bank identifier when printed on the order; usually `null`.
`purchase_order.customer.vat_id`	string	yes	VAT registration number.
`purchase_order.customer.address`	object		Customer (bill-to) postal address.
`purchase_order.customer.address.street`	string	yes	Street and house number.
`purchase_order.customer.address.postal_code`	string	yes	Postal code as printed or as stored in your master record.
`purchase_order.customer.address.city`	string	yes	City.
`purchase_order.customer.address.country`	string	yes	Country code as printed, usually ISO 3166-1 alpha-2.

Path	Type	Nullable	Description
`purchase_order.shipping_details`	object		Ship-to recipient and delivery instructions. Filled from the customer block when the document has no explicit delivery address.
`purchase_order.shipping_details.recipient`	object		The ship-to block.
`purchase_order.shipping_details.recipient.company`	string	yes	Ship-to company name.
`purchase_order.shipping_details.recipient.department`	string	yes	Department or address line 2.
`purchase_order.shipping_details.recipient.contact_person`	string	yes	Ship-to contact for the carrier.
`purchase_order.shipping_details.recipient.email`	string	yes	Ship-to e-mail.
`purchase_order.shipping_details.recipient.phone`	string	yes	Ship-to phone.
`purchase_order.shipping_details.recipient.address`	object		Delivery address.
`purchase_order.shipping_details.recipient.address.street`	string	yes	Street and house number.
`purchase_order.shipping_details.recipient.address.postal_code`	string	yes	Postal code as printed or as stored in your master record.
`purchase_order.shipping_details.recipient.address.city`	string	yes	City.
`purchase_order.shipping_details.recipient.address.country`	string	yes	Country code as printed, usually ISO 3166-1 alpha-2.
`purchase_order.shipping_details.special_instructions`	string	yes	Free-text delivery instructions ("deliver before noon", dock number).
`purchase_order.items[]`	array of objects		One entry per order line, in document order. Lines without an ordered quantity (assortment listings) are already dropped.
`purchase_order.items[].article_number`	string	yes	Your article number (SKU) after resolution through your catalog and customer item mappings. `null` only on an explicit override send of an unresolved line.
`purchase_order.items[].description`	string	yes	Line description as printed by the customer.
`purchase_order.items[].quantity`	integer	yes	Ordered quantity in whole units. `null` only when the quantity was unreadable and an operator overrode the review.
`purchase_order.items[].unit`	string	yes	Unit of measure as printed (`ST` , `PCS` , `Stk` , `m` , `KG` , …) — map it to your ERP's codes.
`purchase_order.items[].price`	number	yes	Unit price as stated by the customer, or `null` .
`purchase_order.items[].currency`	string	yes	Line currency, normally equal to the header currency.

Path	Type	Nullable	Description
`purchase_order.items[].line_total`	number	yes	Line amount as printed (quantity × price), or `null`.

Expected response

Outcome (any HTTP status is accepted; the body decides)

Body your ERP receive endpoint must return. **`success` is authoritative** — AIOTIC ignores the HTTP status for the business outcome (a `500` with `success: false` and a `success: true` with `200` are both handled correctly; a non-JSON body is treated as a server error).

Field	Type	Required	Description
`success`	boolean	yes	
`order_number`	string		Your ERP's reference for the created order; stored as `erp_ref` and shown to operators
`error`	string		Human-readable reason when `success` is `false`; shown to operators

Accepted:

```
{
  "success": true,
  "order_number": "SO-2026-00981"
}
```

Business rejection:

```
{
  "success": false,
  "error": "Unknown article number: PROD-999"
}
```

Processing webhook (optional) — implemented by you

If the tenant has the *processing webhook* enabled, AIOTIC POSTs the extracted order to this URL as soon as extraction finishes — **before** any operator review, and regardless of whether the order landed in `PROCESSED` or `ATTENTION`.

Use it as an early signal ("an order arrived, here is the AI's reading"), not as the hand-off: the reviewed, corrected version reaches you through the ERP receive endpoint.

- Header `X-API-KEY: <configured value>`; respond `2xx` to acknowledge.
- Delivery is best-effort: no retries and no signature today (see the proposal appendix in the guide).

Request body — `application/json`

Body of the optional processing webhook, sent when extraction completes (before any human review).

Field	Type	Required	Description
`request_id`	string	yes	Format: `uuid`.
`purchase_order`	PurchaseOrder	yes	

Every field of the webhook payload

Every key listed here is **always present**; "nullable" means the value may be `null`.

Path	Type	Nullable	Description
`request_id`	string		

Path	Type	Nullable	Description
`purchase_order`	object		The extracted purchase order as returned in `OrderStatus.result`. Every field is always present; optional fields are `null`, never omitted.
`purchase_order.order_number`	string		Customer's PO number. `/` is replaced by `-`.
`purchase_order.order_date`	string		ISO 8601 date (`YYYY-MM-DD`) where the document allowed it
`purchase_order.delivery_date`	string	yes	Single requested delivery date (resolved from a window per tenant preference)
`purchase_order.delivery_date_from`	string	yes	Lower bound when the document states a delivery window
`purchase_order.delivery_date_to`	string	yes	Upper bound when the document states a delivery window
`purchase_order.supplier`	object		The receiving company — your company. Pinned from tenant configuration, not extracted per document.
`purchase_order.supplier.company`	string		Your company name as configured for the tenant.
`purchase_order.supplier.contact_person`	string	yes	Contact person configured for the tenant, if any.
`purchase_order.supplier.email`	string	yes	Order intake e-mail address of the tenant.
`purchase_order.supplier.address`	object		Your company address as configured.
`purchase_order.supplier.address.street`	string		Street and house number.
`purchase_order.supplier.address.postal_code`	string		Postal code.
`purchase_order.supplier.address.city`	string		City.
`purchase_order.supplier.address.country`	string	yes	Country code as printed, usually ISO 3166-1 alpha-2.
`purchase_order.customer`	object	yes	The identified customer (debtor). `customer_id` is your customer number and is set only when AIOTIC resolved the customer with confidence.
`purchase_order.customer.customer_id`	string	yes	
`purchase_order.customer.company`	string		
`purchase_order.customer.contact_person`	string	yes	
`purchase_order.customer.email`	string	yes	

Path	Type	Nullable	Description
`purchase_order.customer.phone`	string	yes	
`purchase_order.customer.branch`	string	yes	Issuing branch / location named on the document, when any
`purchase_order.customer.vat_id`	string	yes	
`purchase_order.customer.iban`	string	yes	
`purchase_order.customer.bic`	string	yes	
`purchase_order.customer.address`	object		
`purchase_order.customer.address.street`	string		Street and house number.
`purchase_order.customer.address.postal_code`	string		Postal code.
`purchase_order.customer.address.city`	string		City.
`purchase_order.customer.address.country`	string	yes	Country code as printed, usually ISO 3166-1 alpha-2.
`purchase_order.shipping_details`	object	yes	
`purchase_order.shipping_details.recipient`	object		The ship-to block.
`purchase_order.shipping_details.recipient.company`	string		Ship-to company name.
`purchase_order.shipping_details.recipient.contact_person`	string	yes	Ship-to contact for the carrier.
`purchase_order.shipping_details.recipient.department`	string	yes	Department or address line 2.
`purchase_order.shipping_details.recipient.email`	string	yes	Ship-to e-mail.
`purchase_order.shipping_details.recipient.phone`	string	yes	Ship-to phone.
`purchase_order.shipping_details.recipient.address`	object		Delivery address.
`purchase_order.shipping_details.recipient.address.street`	string		Street and house number.
`purchase_order.shipping_details.recipient.address.postal_code`	string		Postal code.
`purchase_order.shipping_details.recipient.address.city`	string		City.
`purchase_order.shipping_details.recipient.address.country`	string	yes	Country code as printed, usually ISO 3166-1 alpha-2.
`purchase_order.shipping_details.special_instructions`	string	yes	Free-text delivery instructions.
`purchase_order.items[]`	array of objects		One order line as stored on the order (`result.items[]`).
`purchase_order.items[].article_number`	string	yes	Your (supplier-side) article number, resolved through the catalog and customer mappings where possible
`purchase_order.items[].customer_item_number`	string	yes	The customer's own article number as printed, when present
`purchase_order.items[].description`	string	yes	
`purchase_order.items[].quantity`	integer	yes	Ordered quantity in whole units; `null` when unreadable (see `quantity_state`)

Path	Type	Nullable	Description
`purchase_order.items[].quantity_state`	string	yes	What the quantity column literally showed. `Unrecognised` rows keep an empty quantity and put the order in ATTENTION.
`purchase_order.items[].unit`	string	yes	Unit of measure as printed (e.g. `ST`, `PCS`, `KG`)
`purchase_order.items[].price`	number	yes	Unit price
`purchase_order.items[].currency`	string	yes	
`purchase_order.items[].line_total`	number	yes	
`purchase_order.total_price`	number	yes	
`purchase_order.currency`	string	yes	
`purchase_order.additional_information`	string	yes	Free text from the document / e-mail, plus audit notes AIOTIC appends (e.g. original document total)

Expected response

Acknowledged

API REFERENCE

Schemas

Every object in the public API. Fields typed `… | null` are always present in responses and may be `null`.

Address

Field	Type	Required	Description
`street`	string	yes	Street and house number.
`postal_code`	string	yes	Postal code.
`city`	string	yes	City.
`country`	string | null		Country code as printed, usually ISO 3166-1 alpha-2.

ClassifiedEmail

An e-mail that was classified as something other than a purchase order.

Field	Type	Required	Description
`request_id`	string	yes	Unique identifier for the request/email Format: `uuid`.
`email_type`	string	yes	The classified type of the email
`sender`	string	yes	Email address of the sender
`from_name`	string	yes	Display name of the sender
`subject`	string	yes	Email subject
`timestamp`	string	yes	When the email was received and classified Format: `date-time`.
`metadata`	object		Additional metadata about the email
`message_id`	string | null		RFC 5322 Message-ID (dedup key)
`classification_reason`	string | null		Why the classifier chose the type
`rejection_status`	string | null		Rejected-email triage: None (handled) | pending | overridden
`original_email_type`	string | null		Classifier's original type, preserved on operator override

Customer

A customer (debtor) record as stored in AIOTIC.

Field	Type	Required	Description
`number`	string	yes	Your customer (debtor) number — the key of this record.
`id`	string		Internal identifier, generated by AIOTIC. Format: `uuid`.
`name`	string | null		Company name
`postal_code`	string | null		Postal code
`city`	string | null		City
`address`	string | null		Address

Field	Type	Required	Description
`contact_person`	string | null		Contact person
`phone_number`	string | null		Phone number
`vat_number`	string | null		BTW (VAT) number
`email`	string | null		Customer email
`coc_number`	string | null		Chamber of Commerce number
`home_page`	string | null		Company website URL
`similarity`	number | null		Match score, only populated in search results.
`archived_at`	string | null		Set when the record is archived because its name carries a legacy or closure marker (e.g. "formerly", "do not use"). Archived customers are never chosen by identification but still resolve when explicitly referenced.
`superseded_by`	string | null		Customer number this record redirects to, derived from a "see customer N" marker in the name (e.g. `*** ZIE 12306 ***`). Orders matched to this record are filed under the successor.

CustomerListResponse

A page of customers.

Field	Type	Required	Description
`items`	array of Customer	yes	List of customer objects
`total`	integer	yes	Total number of customers in the system
`limit`	integer	yes	Maximum number of customers returned in this response
`offset`	integer	yes	Number of customers skipped in this response

CustomerProduct

A mapping from a customer's own article code to one of your articles.

Field	Type	Required	Description
`customer_number`	string	yes	Customer identifier referencing the customers table
`customer_item_number`	string	yes	Customer's own item number for this product
`item_number`	string	yes	Supplier product item number
`language_code`	string	yes	Product language code
`created_at`	string	yes	When this mapping was created in AIOTIC Format: `date-time`.

CustomerProductListResponse

A page of customer item mappings.

Field	Type	Required	Description
`items`	array of CustomerProduct	yes	

Field	Type	Required	Description
`total`	integer	yes	
`limit`	integer	yes	
`offset`	integer	yes	

CustomerProductUpsert

Fields accepted when creating or updating a customer item mapping.

Field	Type	Required	Description
`item_number`	string	yes	Supplier product item number
`language_code`	string	yes	Product language code
`created_at`	string		When this mapping was created in AIOTIC Format: `date-time`.

CustomerSearchResponse

Customers matching a search, best match first.

Field	Type	Required	Description
`items`	array of Customer	yes	List of customer objects with similarity scores
`total`	integer	yes	Total number of customers in the system
`limit`	integer	yes	Maximum number of customers returned in this response

CustomerUpsert

Fields accepted when creating or updating a customer. All optional; the more you fill, the better identification works.

Field	Type	Required	Description
`id`	string | null		Unique customer identifier (UUID, auto-generated if not provided)
`name`	string | null		Company name
`postal_code`	string | null		Postal code
`city`	string | null		City
`address`	string | null		Address
`contact_person`	string | null		Contact person
`phone_number`	string | null		Phone number
`vat_number`	string | null		BTW (VAT) number
`email`	string | null		Customer email
`coc_number`	string | null		Chamber of Commerce number
`home_page`	string | null		Company website URL
`similarity`	number | null		Similarity score

EmailClassificationResponse

The category an uploaded e-mail would be classified as.

Field	Type	Required	Description
`category`	enum	yes	The category of the email Values: `purchase_order`, `order_confirmation`, `invoice`, `delivery_note`, `quotation`, `receipt`, `credit_note`, `unknown`.

ErpAddress

Postal address in the ERP hand-off payload (all keys present, values may be `null`).

Field	Type	Required	Description
`street`	string | null	yes	Street and house number.
`postal_code`	string | null	yes	Postal code as printed or as stored in your master record.
`city`	string | null	yes	City.
`country`	string | null	yes	Country code as printed, usually ISO 3166-1 alpha-2.

ErpCustomer

Field	Type	Required	Description
`customer_id`	string | null	yes	Your customer (debtor) number as synced via `PUT /customer/{number}`. `null` only when an operator explicitly sent an unidentified order.
`company`	string | null	yes	Customer company name.
`contact_person`	string | null	yes	Person who placed this order, as printed on the document (document-authoritative, not the master record).
`email`	string | null	yes	Customer e-mail address.
`phone`	string | null	yes	Customer phone number.
`iban`	string | null	yes	Bank account when printed on the order; usually `null`.
`bic`	string | null	yes	Bank identifier when printed on the order; usually `null`.
`vat_id`	string | null	yes	VAT registration number.
`address`	ErpAddress	yes	Customer (bill-to) postal address.

ErpOrderItem

One order line in the ERP hand-off payload (a fixed subset of `OrderItem`).

Field	Type	Required	Description
`article_number`	string | null	yes	Your article number (SKU) after resolution through your catalog and customer item mappings. `null` only on an explicit override send of an unresolved line.
`description`	string | null	yes	Line description as printed by the customer.
`quantity`	integer | null	yes	Ordered quantity in whole units. `null` only when the quantity was unreadable and an operator overrode the review.
`unit`	string | null	yes	Unit of measure as printed (`ST`, `PCS`, `Stk`, `m`, `KG`, …) — map it to your ERP's codes.
`price`	number | null	yes	Unit price as stated by the customer, or `null`.
`currency`	string | null	yes	Line currency, normally equal to the header currency.

Field	Type	Required	Description
`line_total`	number | null	yes	Line amount as printed (quantity × price), or `null`.

ErpPurchaseOrder

The purchase order as delivered to your ERP receive endpoint — the reviewed values with operator corrections applied. Keys are always present (`null` when unknown).

Field	Type	Required	Description
`order_number`	string	yes	The customer's purchase-order number as printed, with `/` replaced by `-`. Map it to the "your reference" / external document number of the sales order.
`order_date`	string | null	yes	Order date as an ISO `YYYY-MM-DD` string where the document allowed it; otherwise as printed.
`delivery_date`	string | null	yes	Requested delivery date (`YYYY-MM-DD`) or `null`. When the document stated a window it is already collapsed to a single date.
`currency`	string | null	yes	Currency code as printed on the document (e.g. `EUR`) or `null`.
`total_price`	number | null	yes	The document's printed total, which may include VAT, or the recalculated net sum when e-mail instructions changed lines. Informational — let your ERP compute totals.
`additional_information`	string | null	yes	Free text from the document and e-mail plus notes AIOTIC appends (e.g. the original printed total after a recalculation).
`supplier`	Supplier | null	yes	Your own company as configured for the tenant — identical on every order, never read from the document. Safe to ignore.
`customer`	ErpCustomer	yes	The identified customer (debtor). `customer_id` is your customer number; the address block is canonicalised from your master record on a confident match.
`shipping_details`	ErpShippingDetails	yes	Ship-to recipient and delivery instructions. Filled from the customer block when the document has no explicit delivery address.
`items`	array of ErpOrderItem	yes	One entry per order line, in document order. Lines without an ordered quantity (assortment listings) are already dropped.

ErpReceiveRequest

Body AIOTIC POSTs to your ERP receive endpoint.

Field	Type	Required	Description
`request_id`	string	yes	AIOTIC's order id (UUID). Stable across retries of the same send — use it as the idempotency key and store it on your sales order as external reference. Format: `uuid`.
`purchase_order`	ErpPurchaseOrder	yes	The reviewed purchase order with operator corrections applied. Every key below is always present; unknown values are `null`.

ErpReceiveResponse

Body your ERP receive endpoint must return. **`success` is authoritative** — AIOTIC ignores the HTTP status for the business outcome (a `500` with `success: false` and a `success: true` with `200` are both handled correctly; a non-JSON body is treated as a server error).

Field	Type	Required	Description
`success`	boolean	yes	
`order_number`	string		Your ERP's reference for the created order; stored as `erp_ref` and shown to operators
`error`	string		Human-readable reason when `success` is `false`; shown to operators

ErpRecipient

Ship-to block in the ERP hand-off payload.

Field	Type	Required	Description
`company`	string | null	yes	Ship-to company name.
`department`	string | null	yes	Department or address line 2.
`contact_person`	string | null	yes	Ship-to contact for the carrier.
`email`	string | null	yes	Ship-to e-mail.
`phone`	string | null	yes	Ship-to phone.
`address`	ErpAddress	yes	Delivery address.

ErpSendResponse

Field	Type	Required	Description
`success`	boolean	yes	
`request_id`	string	yes	Format: `uuid`.
`data`	ErpReceiveResponse	yes	The JSON body your ERP receive endpoint returned, verbatim.

ErpShippingDetails

Shipping section of the ERP hand-off payload.

Field	Type	Required	Description
`recipient`	ErpRecipient	yes	The ship-to block.
`special_instructions`	string | null	yes	Free-text delivery instructions ("deliver before noon", dock number).

ErrorResponse

Standard error body. `detail` is usually a string; a few endpoints return a structured object (documented per endpoint).

Field	Type	Required	Description
`detail`	string | object	yes	

FetchAllEmailsResponse

Result of a manual mailbox fetch.

Field	Type	Required	Description
`status`	string	yes	
`emails_queued`	integer	yes	
`emails_total`	integer	yes	

Field	Type	Required	Description
`message`	string	yes	

HTTPValidationError

Request validation error (malformed parameters).

Field	Type	Required	Description
`detail`	array of ValidationError		

OrderCustomer

The identified customer (debtor). `customer_id` is your customer number and is set only when AIOTIC resolved the customer with confidence.

Field	Type	Required	Description
`customer_id`	string | null		
`company`	string	yes	
`contact_person`	string | null		
`email`	string | null		
`phone`	string | null		
`branch`	string | null		Issuing branch / location named on the document, when any
`vat_id`	string | null		
`iban`	string | null		
`bic`	string | null		
`address`	Address	yes	

OrderGroup

Field	Type	Required	Description
`email_group_id`	string	yes	Format: `uuid`.
`message_id`	string | null		
`order_count`	integer	yes	
`orders`	array of OrderStatus	yes	

OrderItem

One order line as stored on the order (`result.items[]`).

Field	Type	Required	Description
`article_number`	string | null		Your (supplier-side) article number, resolved through the catalog and customer mappings where possible
`customer_item_number`	string | null		The customer's own article number as printed, when present
`description`	string | null		

Field	Type	Required	Description
`quantity`	integer | null		Ordered quantity in whole units; `null` when unreadable (see `quantity_state`)
`quantity_state`	string | null		What the quantity column literally showed. `Unrecognised` rows keep an empty quantity and put the order in ATTENTION. Values: `Valid`, `Empty`, `Zero`, `Unrecognised`.
`unit`	string | null		Unit of measure as printed (e.g. `ST`, `PCS`, `KG`)
`price`	number | null		Unit price
`currency`	string | null		
`line_total`	number | null		

OrderListResponse

A page of orders, newest first.

Field	Type	Required	Description
`items`	array of OrderStatus	yes	List of order status objects
`total`	integer	yes	Total number of orders in the system
`limit`	integer	yes	Maximum number of orders returned in this response
`offset`	integer	yes	Number of orders skipped in this response

OrderRef

Reference to one child order of a split e-mail.

Field	Type	Required	Description
`request_id`	string	yes	Unique identifier for the child order Format: `uuid`.
`status`	OrderStatusValue	yes	
`order_label`	string | null		Splitter-assigned label for the order

OrderRetryBody

Field	Type	Required	Description
`hil_prompt`	string | null		

OrderStatus

The processing status of one order, with the extracted purchase order once available.

Field	Type	Required	Description
`request_id`	string	yes	Unique identifier for the request Format: `uuid`.
`attachments`	object		File names of the order with size and MIME type.
`metadata`	object		Extra form fields supplied at upload, plus `source`.
`timestamp`	string	yes	When this status was recorded Format: `date-time`.

Field	Type	Required	Description
`status`	OrderStatusValue	yes	
`result`	PurchaseOrder | null		The extracted purchase order once processing finished; `null` before that and for failed orders.
`state`	object | null		Free-form processing details used by the AIOTIC app (validation notes, flags). Informational.
`erp_ref`	string | null		ERP-assigned order reference returned on successful send
`email_group_id`	string | null		Correlates orders from the same source email
`order_label`	string | null		Human-readable label, e.g. 'PO-12345' or 'Order 1 of 2'
`retry_count`	integer		Number of retry attempts for transient failures Default: `0`.
`next_retry_at`	string | null		When to attempt next retry
`last_error`	string | null		Last error message from transient failure

OrderStatusValue

Order lifecycle status.

Values: `QUEUED`, `PROCESSING`, `ATTENTION`, `PROCESSED`, `FAILED`, `RETRY_PENDING`, `MODIFIED`, `REPROCESSED`, `SENDING`, `SENT`, `CANCELED`

OrderUploadBody

Field	Type	Required	Description
`files`	array of string	yes	
`request_id`	string | null		

OrderUploadResponse

Result of an upload. When `split` is true, poll the child `orders[]` instead of `request_id`.

Field	Type	Required	Description
`request_id`	string	yes	Unique identifier for the uploaded document Format: `uuid`.
`split`	boolean		True when the upload was split into multiple orders; poll orders[] rather than request_id Default: `false`.
`email_group_id`	string | null		Correlation id for child orders when the email was split into multiple orders
`orders`	array of OrderRef | null		Child order references when the email was split into multiple orders

ProcessingWebhookRequest

Body of the optional processing webhook, sent when extraction completes (before any human review).

Field	Type	Required	Description
`request_id`	string	yes	Format: `uuid`.
`purchase_order`	PurchaseOrder	yes	

Product

An article record (one per article number and language).

Field	Type	Required	Description
`item_number`	string	yes	Unique identifier for the product
`language_code`	string	yes	Language code for the product description
`description`	string | null		Product description
`remark`	string | null		Additional remarks about the product
`created_at`	string	yes	When this product was created in AIOTIC Format: `date-time`.

ProductListResponse

A page of products.

Field	Type	Required	Description
`items`	array of Product	yes	
`total`	integer	yes	
`limit`	integer	yes	
`offset`	integer	yes	

ProductUpsert

Fields accepted when creating or updating a product.

Field	Type	Required	Description
`description`	string	yes	Product description
`remark`	string | null		Additional remarks about the product
`created_at`	string		When this product was created in AIOTIC Format: `date-time`.

PurchaseOrder

The extracted purchase order as returned in `OrderStatus.result`. Every field is always present; optional fields are `null`, never omitted.

Field	Type	Required	Description
`order_number`	string	yes	Customer's PO number. `/` is replaced by `-`.
`order_date`	string	yes	ISO 8601 date (`YYYY-MM-DD`) where the document allowed it
`delivery_date`	string | null		Single requested delivery date (resolved from a window per tenant preference)
`delivery_date_from`	string | null		Lower bound when the document states a delivery window
`delivery_date_to`	string | null		Upper bound when the document states a delivery window
`supplier`	Supplier	yes	
`customer`	OrderCustomer | null		
`shipping_details`	ShippingDetails | null		
`items`	array of OrderItem	yes	
`total_price`	number | null		
`currency`	string | null		

Field	Type	Required	Description
`additional_information`	string | null		Free text from the document / e-mail, plus audit notes AIOTIC appends (e.g. original document total)

RawEmailClassifyBody

Field	Type	Required	Description
`file`	string	yes	

RawEmailUploadBody

Field	Type	Required	Description
`file`	string	yes	
`request_id`	string | null		

RawUploadRejection

Field	Type	Required	Description
`detail`	string | object	yes	

RejectedEmailListResponse

A page of rejected e-mails.

Field	Type	Required	Description
`items`	array of ClassifiedEmail		Rejected emails on this page
`total`	integer	yes	Total rejected emails matching the rejection_status filter
`limit`	integer	yes	Page size
`offset`	integer	yes	Pagination offset

ReprocessResponse

Result of forcing a rejected e-mail through order processing.

Field	Type	Required	Description
`request_id`	string	yes	The request being reprocessed (reuses the original id) Format: `uuid`.
`status`	string	yes	Outcome, e.g. 'reprocessing'

ShippingDetails

Field	Type	Required	Description
`recipient`	ShippingRecipient	yes	The ship-to block.
`special_instructions`	string | null		Free-text delivery instructions.

ShippingRecipient

Field	Type	Required	Description
`company`	string	yes	Ship-to company name.
`contact_person`	string | null		Ship-to contact for the carrier.
`department`	string | null		Department or address line 2.
`email`	string | null		Ship-to e-mail.
`phone`	string | null		Ship-to phone.
`address`	Address	yes	Delivery address.

Supplier

The receiving company — your company. Pinned from tenant configuration, not extracted per document.

Field	Type	Required	Description
`company`	string	yes	Your company name as configured for the tenant.
`contact_person`	string | null		Contact person configured for the tenant, if any.
`email`	string | null		Order intake e-mail address of the tenant.
`address`	Address	yes	Your company address as configured.

SystemStatus

Field	Type	Required	Description
`status`	enum	yes	Values: `operational`, `degraded`, `unknown`.
`message`	string | null		
`updated_at`	string | null		Format: `date-time`.

ValidationError

One request validation problem.

Field	Type	Required	Description
`loc`	array of string | integer	yes	
`msg`	string	yes	
`type`	string	yes	
`input`	object		
`ctx`	object		

SECURITY & OPERATIONS

Security checklist

Transport

- [ ] All calls to AIOTIC over HTTPS (the tenant URL is HTTPS-only).
- [ ] Your receive endpoint and webhook URLs are HTTPS with a valid certificate. Self-signed certificates are not accepted.
- [ ] If you allow-list source IPs, ask your AIOTIC contact for the tenant's egress addresses and re-check when they change.

Keys

- [ ] Integration key lives in exactly one place: your order-processing service. Not in browsers, mobile apps, spreadsheets, ERP client scripts.
- [ ] Master-data jobs use the **sync key**, which cannot read or send orders.
- [ ] The key you gave AIOTIC for *your* endpoints is long and random (≥ 24 bytes; `aiotic init` generates one), stored as a secret, compared in constant time.
- [ ] Separate keys per environment (test tenant vs production tenant vs your ERP test/prod).
- [ ] Rotation procedure written down: who asks AIOTIC for a new integration key, how the sync key is rotated by a tenant admin, how you rotate your endpoint key (configure the new one in AIOTIC first, accept both briefly, drop the old one).
- [ ] Logs never contain key values. Log "auth failed" and the source IP.

Inbound calls (your receive endpoint / webhooks)

- [ ] Reject missing or wrong `X-API-KEY` with `401` before parsing the body.
- [ ] Parse JSON with a schema (the SDK models); reject unknown shapes with `400` and `success: false`.
- [ ] Idempotent on `request_id`; body size limited (an order is kilobytes, not megabytes).
- [ ] Never execute or interpret text from the payload: descriptions, notes and addresses are customer-written free text. Treat them as data in SQL (parameters, never string concatenation) and in any downstream template.
- [ ] Rate-limit and alert on bursts of `401`s (probing).

Outbound calls (to AIOTIC)

- [ ] Client-side rate limit (SDK default 10 req/s) and bounded concurrency in sync jobs.
- [ ] Timeouts and retries with backoff; no infinite retry loops on `4xx`.
- [ ] Only the fields you intend to sync leave your ERP (no prices, margins, credit data in `remark`).

Data handling

- [ ] Purchase orders contain personal data (contact names, e-mails, phone numbers). Apply your retention policy to what you store from payloads and downloaded artifacts.
- [ ] Do not copy AIOTIC artifacts (original PDFs) into systems that are not part of your order process.
- [ ] Support tickets: share `request_id`s, not payloads, unless the channel is approved for personal data.

AI-assisted development

- [ ] Use the **docs-mode** MCP server (public docs only) freely. Enable **tenant mode** only on a developer machine, only against the mock or a test tenant, with write tools disabled unless needed. See MCP server.
- [ ] Never paste keys into prompts or chat tools. The SDK reads them from the environment for a reason.

Dependencies

- [ ] Pin the SDK version; review the changelog before upgrading.
- [ ] `pip audit` / `npm audit` in CI for the integration service.

SECURITY & OPERATIONS

Data, privacy & retention

What AIOTIC stores per order

Data	Where you can see it	Notes
Original files (PDF, images, `.eml`)	`GET /order/{id}/{filename}`, artifacts zip, the app	Needed for review and support.
Extracted purchase order (`result`)	status endpoints, the app	Includes customer contact details as printed.
Processing state and validation notes (`state`)	status endpoints, the app	Which fields were flagged and why.
Operator corrections	the app; merged into the hand-off payload	Original values are kept next to corrections (audit).
ERP reference (`erp_ref`)	status endpoints, the app	Your order number, after a successful send.
Classification records for rejected mails	`/rejected/*`, the app	Sender, subject, reason.

Master data you sync (customers, products, mappings) is stored in full; customer records are additionally indexed for identification.

Personal data

Purchase orders routinely carry names, e-mail addresses and phone numbers of contact persons, and sometimes bank details. In GDPR terms your company is the controller; AIOTIC processes on your behalf under the agreement you have with DevOps Company. Practical consequences for the integration:

- Sync only the customer fields that help identification (name, address, VAT, e-mail, phone, contact). Do not sync notes, credit information or private remarks.
- Your receive endpoint should store what your ERP needs; log `request_id` and outcomes, not entire payloads, unless your log store is in scope of your retention policy.
- Downloaded artifacts (PDFs) are copies — keep them where the originals of orders are kept, not on developer machines.

Retention

Retention of orders and artifacts in AIOTIC is a tenant setting agreed during onboarding. Deleting a customer or product record via the API removes it from AIOTIC's master data immediately; it does not alter historical orders.

Where data is processed

Tenants run in the EU. The AI models used for extraction are configured per platform; ask your AIOTIC contact for the current processing overview and DPA if you need it for your records.

Your integration service

The SDK's default stores are local SQLite files (`integration.db`, `sync-state.db`, `watch.db`). They hold `request_id`s, order numbers, fingerprints and statuses — no payloads. Include them in your backup and retention policies like any operational database.

SECURITY & OPERATIONS

Monitoring

What to watch on the AIOTIC side

Check	How	Alert when
Tenant reachable	`GET /healthcheck` every minute (no key)	2 consecutive failures
Platform status	`GET /system-status` every 5 min	`status != "operational"` — pause uploads, expect slower processing
Orders stuck	watcher or list walk: orders in `PROCESSING` > 15 min, `RETRY_PENDING` with `retry_count ≥ 3`	any
`ATTENTION` backlog	count of `ATTENTION` orders older than N hours	above your SLA
`FAILED` orders	transitions to `FAILED`	any
Rejected mails pending	`GET /rejected/list?status=pending` total	growing for a day
Mailbox stalled (mail intake tenants)	orders stop arriving while mail is sent; trigger `POST /email-watcher/fetch-all` (integration key) or ask a tenant admin to check the mailbox status in the app	no new orders during business hours

What to watch on your side

Check	Alert when
Receive endpoint: `success: false` rate by error text	a spike in "unknown article" or "unknown customer" (your sync is behind)
Receive endpoint: p95 latency	> 10 s (AIOTIC times out at 30 s)
Receive endpoint: `401` count	bursts (probing)
Sync: `SyncReport.failed`	> 0 in two consecutive runs
Sync: watermark age (polling sources)	not advanced for > 2 intervals
Watcher: last successful poll	older than 3 intervals

Correlation

Log the `request_id` on every line that concerns an order, on both sides. Your ERP order carries it as external reference. Support requests to AIOTIC should quote it.

Health endpoints in the SDK service

- `GET /healthz` — process is up.
- `aiotic doctor` — tenant reachable, keys valid, master data present; use it as a readiness probe in CI/CD, not on every request.

AI-ASSISTED INTEGRATION

LLM-native docs (llms.txt)

This guide is published in forms an AI coding assistant can consume directly, so "build me the AIOTIC receive endpoint" produces code that matches the real contracts instead of a plausible guess.

File	Content	Use
`/llms.txt`	Index: one line per page with a summary, plus links to the spec and samples	Paste the URL into your assistant; it fetches what it needs
`/llms-full.txt`	The **whole guide** as one Markdown file, in reading order	Add to a project's context / knowledge files
`/openapi.yaml`	The public OpenAPI 3.1 document, including the outbound webhooks	Code generation, request validation, schema-aware assistants
`/samples/*.json`	Real payload samples (ERP receive request, order status)	Fixtures for tests
`<page URL>.md`	Every page is also served as raw Markdown at `…/path.md`	Selective context

How to use it

```
<span class="tab">Claude Code / Cursor / Copilot chat</span>
Read https://developers.aiotic.ai/llms.txt and then implement an ASP.NET minimal-API endpoint
that receives AIOTIC orders according to the "ERP receive endpoint" contract, including
X-API-KEY verification and idempotency on request_id.
```

```
<span class="tab">Project knowledge</span>
Add https://developers.aiotic.ai/llms-full.txt to the project's documentation sources (or
download it into docs/vendor/aiotic.md) so every prompt in this repo has the contracts available.
```

```
<span class="tab">CLAUDE.md / AGENTS.md</span>
# In the repository of your integration service:
echo "AIOTIC integration contracts: https://developers.aiotic.ai/llms-full.txt (OpenAPI: /openapi.yaml)"
>> CLAUDE.md
```

For richer interaction — searching the guide, pulling one endpoint's schema, fetching examples, or (opt-in) inspecting your own test tenant — use the MCP server.

What the assistant should know

The most common mistakes we see in generated integrations, all covered in the docs the assistant will read:

- treating the HTTP status of the receive endpoint as the outcome (only `success` counts),
- forgetting idempotency on `request_id`,
- re-uploading the entire catalog on a schedule instead of syncing changes,
- assuming a corrections/cancel endpoint exists (it does not yet — see Headless limits),
- polling a single order in a tight loop instead of with backoff.

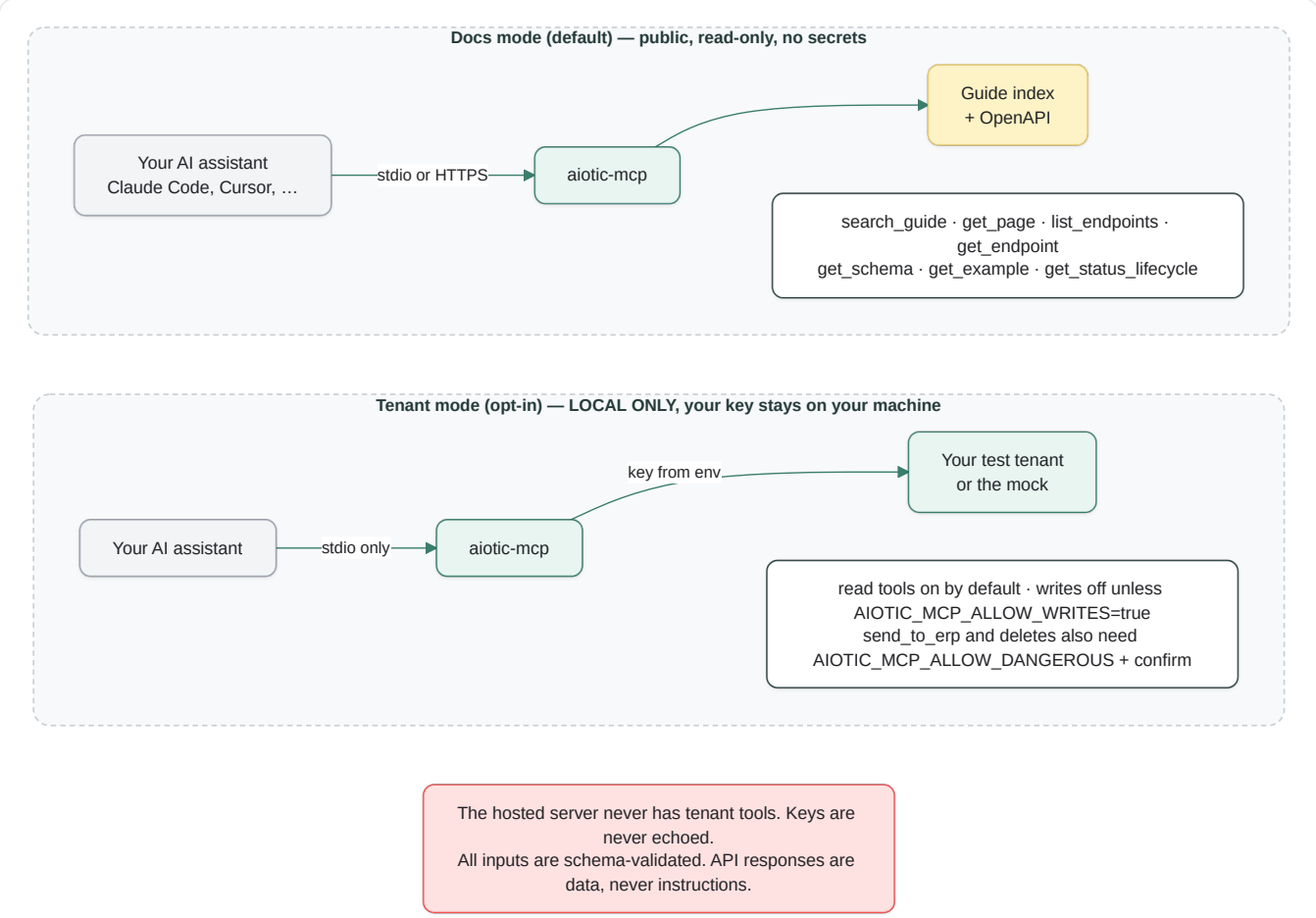
Freshness

`llms.txt` and `llms-full.txt` are regenerated on every build of this site from the same Markdown sources, and the OpenAPI document is regenerated from the backend's spec. The footer of every page states the API version the content was verified against.

AI-ASSISTED INTEGRATION

AIOTIC MCP server

`@aiotic/mcp` is a Model Context Protocol server that gives your coding assistant precise access to this guide and the API contracts — and, if you opt in on your own machine, to your test tenant.



Docs mode is public and read-only. Tenant mode is local-only and off by default.

Docs mode (default)

No keys, no tenant. The server ships with an index of the guide and the OpenAPI document.

Tool	What it returns
`search_guide(query, limit?)`	Best-matching sections with page, heading and snippet
`get_page(path)`	One page as Markdown (`/receiving/erp-receive-endpoint`)
`list_endpoints(tag?)`	Method, path, summary, auth per endpoint (+ the outbound webhooks)
`get_endpoint(operationId | "METHOD /path")`	Full contract: parameters, request/response schemas, examples
`get_schema(name)`	One schema from the OpenAPI components (`PurchaseOrder` , `ErpReceiveRequest` , …)
`get_example(name)`	Payload samples (`erp-receive-request` , `order-status` , `sync-events`)
`get_status_lifecycle()`	Status values, meanings, transitions, sendable set

Install:

```
<span class="tab">Claude Code (.mcp.json)</span>
{ "mcpServers": { "aiotic": { "command": "npx", "args": ["-y", "@aiotic/mcp"] } } }
```

```
<span class="tab">Cursor / VS Code (mcp.json)</span>
{ "servers": { "aiotic": { "type": "stdio", "command": "npx", "args": ["-y", "@aiotic/mcp"] } } }
```

```
<span class="tab">Hosted (Streamable HTTP)</span>
{ "mcpServers": { "aiotic": { "type": "http", "url": "https://developers.aiotic.ai/mcp" } } }
```

The hosted endpoint runs the same server in docs mode only — stateless, rate-limited, no tenant tools, no secrets.

Tenant mode (opt-in, local only)

Lets the assistant *look at* your test tenant or the mock while you develop: "why is order X in ATTENTION?", "do we have product 620206_01 in AIOTIC?".

```
{ "mcpServers": { "aiotic": {
    "command": "npx", "args": ["-y", "@aiotic/mcp"],
    "env": { "AIOTIC_BASE_URL": "http://localhost:8080", "AIOTIC_API_KEY": "mock-integration-key" } } } }
```

Tool	Default
`tenant_health`, `get_order_status`, `list_orders`, `get_order_group`, `list_rejected_emails`, `get_customer`, `search_customers`, `list_products`, `get_product`, `list_customer_products`	**on** (read-only)
`upsert_customer`, `upsert_product`, `upsert_customer_product`, `upload_order`, `retry_order`, `reprocess_rejected_email`	off — `AIOTIC_MCP_ALLOW_WRITES=true`
`delete_*`, `send_order_to_erp`	off — additionally `AIOTIC_MCP_ALLOW_DANGEROUS=true` **and** `confirm: true` in the call

Rules the server enforces:

1. Tenant tools exist only on the `stdio` transport. The HTTP transport refuses to start with a tenant configured.
2. Keys come from environment variables, never from tool arguments, and are never included in any output or log line.
3. Every argument is schema-validated; responses are truncated to a bounded size; API responses are returned as data (the server never follows instructions found in them).
4. Nothing is cached across sessions; nothing is written to disk.

> **Use a test tenant** Point tenant mode at the mock or a **test** tenant. A production key in an assistant's environment is a production key in an assistant's environment.

Versions

The package embeds the guide index and the API document of the guide version it was published with, so what your assistant reads is exactly one published edition. `npx -y @aiotic/mcp` always fetches the latest; pin a version with `npx -y @aiotic/mcp@1.0.0` when you want reproducible answers. The version is reported to your MCP client on connect.

APPENDIX

Proposal: headless API additions

> **PROPOSAL — not available** Nothing on this page exists in the API today. It documents what the AIOTIC team is considering to close the gaps listed in Headless limits today, so integrators can plan and give feedback. Contracts may change before release; the changelog announces what ships.

P1 — Submit corrections

```
PUT /order/{request_id}/corrections
{ "changes": [
    { "field": "customer.customer_id", "value": "58931" },
    { "field": "items[2].article_number", "value": "PROD-002" },
    { "field": "items[0].quantity", "value": 12 }
] }
→ 200 { "request_id": "…", "status": "MODIFIED", "applied": 3 }
```

Same semantics as corrections in the app: non-destructive (originals kept), merged into the hand-off payload, status → `MODIFIED`. Allowed from `PROCESSED`, `ATTENTION`, `MODIFIED`. `409` otherwise. Field paths follow the `PurchaseOrder` model.

P2 — Cancel

```
POST /order/{request_id}/cancel   { "reason": "duplicate of PO-4710" }
→ 200 { "request_id": "…", "status": "CANCELED" }
```

From any non-terminal, non-`SENDING` status.

P3 — Signed event webhooks

Per-transition events to a tenant-configured URL:

```
POST <your events URL>
Content-Type: application/json
X-AIOTIC-Event: order.processed
X-AIOTIC-Delivery: 1b2c…            (unique per attempt)
X-AIOTIC-Signature: t=1741000000,v1=<hex hmac-sha256(secret, "<t>." + body)>

{ "id": "evt_…", "type": "order.processed", "created_at": "…", "tenant": "acme",
  "data": { "request_id": "…", "status": "PROCESSED", "previous_status": "PROCESSING", "order_number":
"PO-4711", "erp_ref": null } }
```

Types: `order.queued`, `order.processed`, `order.attention`, `order.failed`, `order.retry_pending`, `order.modified`, `order.sent`, `order.send_failed`, `order.canceled`, `email.rejected`. Retries with exponential backoff for 24 h on non-2xx; `GET /events?since=` for replay. The SDK's `verify_hmac_signature` already implements the signature check.

P4 — Batch upserts

```
POST /customer/batch          { "items": [ { "number": "58931", ...fields } ] }        ≤ 1000
POST /product/batch           { "items": [ { "item_number": "…", "language_code": "nl", "description":
"…" } ] }
POST /customer-product/batch  { "items": [ { "customer_number": "…", "customer_item_number": "…",
"item_number": "…", "language_code": "nl" } ] }
→ 200 { "results": [ { "key": "58931", "status": "upserted" | "error", "error": "…" } ] }
```

Per-item results, no all-or-nothing. Same keys as the single-record endpoints.

P5 — Change detection on reads

- `updated_at` on customers, products, mappings and orders; `?updated_since=<ISO>` on every list endpoint.
- `ETag` / `If-None-Match` on single-record reads.
- `?status=` and `?since=` filters on `GET /order_status/list`.

P6 — Scoped keys and rate limits

Keys with scopes (`orders:read` , `orders:send` , `masterdata:write` , …) manageable by tenant admins; documented per-tenant rate limit with `429` + `Retry-After` .

Feedback

Which of these unblocks your integration? Tell your AIOTIC contact. Ordering will follow real demand.

APPENDIX

Status reference

The complete `status` enumeration of an order, as returned by the status endpoints. Kept identical to the backend's definition (`OrderStatusValue` in the OpenAPI schemas).

Value	Badge	Landed?	Sendable?	Final?	Set by
`QUEUED`	`QUEUED`	no	no	no	intake
`PROCESSING`	`PROCESSING`	no	no	no	processing
`PROCESSED`	`PROCESSED`	yes	**yes**	no	processing
`ATTENTION`	`ATTENTION`	yes	yes (explicit override)	no	processing
`FAILED`	`FAILED`	yes (no result)	no	no (retry creates a new order)	processing
`RETRY_PENDING`	`RETRY_PENDING`	no	no	no	retry scheduler
`MODIFIED`	`MODIFIED`	yes	**yes**	no	operator (app)
`SENDING`	`SENDING`	yes	no (locked)	no	`POST /erp/send`
`SENT`	`SENT`	yes	no	**yes**	`POST /erp/send` success
`REPROCESSED`	`REPROCESSED`	—	no	**yes**	`POST /order/retry` (on the old order)
`CANCELED`	`CANCELED`	—	no	**yes**	operator (app)

Rejected e-mails have their own, separate `rejection_status`: `pending` → `overridden`.

Transitions

```
QUEUED → PROCESSING → PROCESSED | ATTENTION | FAILED | RETRY_PENDING
RETRY_PENDING → PROCESSING
PROCESSED | ATTENTION → MODIFIED            (operator edit)
PROCESSED | MODIFIED | ATTENTION → SENDING → SENT
SENDING → (previous status)                 (send failed)
FAILED → REPROCESSED  (+ new order QUEUED)  (retry)
any non-final → CANCELED                    (operator)
```

Forward compatibility

New values may be added. Treat an unknown value as "not mine to act on", log it, and keep polling.

APPENDIX

Glossary

Term	Meaning
AIOTIC app	The operator web application (review, correct, send, manage users and settings). Talks to the same tenant as the API.
Article number	Your SKU / item number. `article_number` on order lines.
ATTENTION	Order status meaning "a person must decide". Not an error.
Customer	Debtor: the company that sent the purchase order. `customer_id` = your customer number.
Customer item number	The customer's own code for one of your articles, as printed on their orders.
Customer item mapping	Record linking `(customer_number, customer_item_number)` → `(item_number, language_code)`.
ERP receive endpoint	The HTTPS endpoint you implement; AIOTIC `POST`s reviewed orders to it.
erp_ref	Your ERP's order reference, stored in AIOTIC after a successful send.
Extraction	Turning a document + e-mail into a structured `PurchaseOrder` with AI.
Hand-off	The act of sending an order to the ERP receive endpoint (`POST /erp/send`).
Integration key	API key accepted on every endpoint.
Landed	An order that finished processing: `PROCESSED` , `ATTENTION` , `MODIFIED` (with result) or `FAILED` .
Master data / reference data	Customers, products, customer item mappings.
MCP	Model Context Protocol — how AI assistants call tools; AIOTIC ships an MCP server for the guide.
Mock tenant	Local stand-in for an AIOTIC tenant (`aiotic mock`).
Order splitting	Tenant feature that turns one e-mail with several orders into several child orders sharing an `email_group_id` .
Pipeline	The SDK's sanitizers → validators → business rules chain in the receive endpoint.
Processing webhook	Optional call AIOTIC makes when extraction finishes, before review.
Purchase order (PO)	The document the buyer (your customer) sends. What AIOTIC reads; the `purchase_order` field in every payload.
Sales order (SO)	The record your ERP creates from a purchase order. Your receive endpoint returns its number as `order_number` , stored in AIOTIC as `erp_ref` .
Rejected e-mail	A mail the classifier decided is not a purchase order; can be overridden.
request_id	UUID identifying one order in AIOTIC; stable across retries; your idempotency key.
Sendable	`PROCESSED` , `MODIFIED` , `ATTENTION` .
Supplier	You — the receiving company. Pinned from configuration.
Sync key	API key accepted only on master-data endpoints.
Tenant	One AIOTIC deployment for one company (`https://<tenant>.aiotic.ai`).
Unit (of measure)	As printed on the document (`ST` , `PCS` , `Stk` , `m` …); map it on your side.

APPENDIX

FAQ

Contract

Does AIOTIC retry a send if my endpoint is down? No. A failed send rolls the order back to its previous status; an operator (or your code) sends again. Your idempotency on `request_id` makes that safe.

Can I return `200` with `success: false`? Yes. The body decides. Many integrators do exactly that so their monitoring separates transport errors from business rejections.

What if `customer_id` is `null` in the payload? An operator explicitly sent an unidentified order (override of `ATTENTION`). Decide your policy: reject with a clear message, or book to a "walk-in" debtor and flag it.

Why does `total_price` not match the sum of lines? It is the printed document total, which often includes VAT, or was recalculated after e-mail instructions changed lines (then `additional_information` notes the original). Do not book it; let your ERP compute totals.

What is `supplier` for? It is you. It makes the payload self-describing (and useful for multi-company setups). Ignore it otherwise.

Master data

Why does AIOTIC sync my data instead of reading it live from my ERP? Performance and consistency: matching runs against normalised, enriched data that sits next to the processing, independent of your ERP's speed, availability or API style; and the synchronised data is what AIOTIC benchmarks and improves against for your tenant. See Reference data.

Do I really need to sync products? The article numbers are on the document. Yes. The product table is the definition of a *valid* article number. Without it AIOTIC cannot tell a mis-read `PR0D-001` from a real code, and every order would need a human.

How often should I sync? On change. Use events or an `updated_at` poll; reconcile nightly/weekly. Never schedule full dumps.

Can I sync from a script inside the ERP? Yes — that is what the sync key is for. Use one `PUT` per changed record.

Can two customers share a VAT number? Yes (branches, group companies). Keep `city` and, if applicable, the branch in the name accurate; AIOTIC uses the branch printed on the document to pick the right one.

Orders

How do I get notified when an order is ready? Poll — see Polling & notifications. The processing webhook fires once when extraction finishes; signed status webhooks are proposed.

Can I correct a field through the API? Not yet (Headless limits). Corrections happen in the AIOTIC app or on your side after receiving the payload.

A customer sent the same PO twice. What happens? Two orders with different `request_id`s. Detect duplicates in your receive endpoint by `(customer_id, order_number)` and reject the second with a clear message (the SDK's `NoDuplicateOrder`).

An e-mail with three POs arrived. What do I get? If the tenant has order splitting enabled: three orders sharing an `email_group_id`. Otherwise one merged order — ask for splitting to be enabled.

What does `Unrecognised` mean on a line? A quantity was there but unreadable (handwritten, smudged). `quantity` is `null` and the order is in `ATTENTION` so someone types it in.

Keys & security

Can I use one key for everything? The integration key works everywhere, but keep it in one service and give sync jobs the sync key. See Security checklist.

Is there IP allow-listing? On your side, yes — ask for the tenant's egress addresses. On AIOTIC's side, keys are the control.

SDK

I do not use Python. The contracts are language-neutral; this guide shows curl for everything and C# for the receive endpoint. The Python SDK doubles as an executable specification — read `aiotic/receive.py` and `aiotic/sync/engine.py` for the logic worth porting. Other languages are planned.

Can I use the SDK with Flask/Django? Yes. `ErpReceiver.handle()`, `Pipeline.run()` and `SyncEngine.apply()` are plain Python; only the routers are FastAPI.

APPENDIX

Changelog

The guide follows the AIOTIC API. Entries name the API version a change appeared in and what integrators need to do.

Guide 1.0.0 — verified against API 1.0.0 (2026-09)

First public edition.

- Full contract for the **ERP receive endpoint** and the **processing webhook** (also as OpenAPI `webhooks`).
- Public OpenAPI 3.1 document covering intake, status, ERP hand-off, rejected e-mails, master data and health.
- Sync guidance: event-driven first, hash-based reconciliation as safety net, initial load order and throughput.
- Python SDK 0.1.0: client, receive endpoint, pipeline, ERP adapter templates, sync engine, watcher, webhooks, CLI, mock tenant.
- LLM-native docs (`llms.txt`, `llms-full.txt`) and the MCP server (docs mode + opt-in tenant mode).
- Appendix with the headless-API proposal (corrections, cancel, signed events, batch upserts, change detection, scoped keys).

API changes tracked

This section lists changes to the public API as they are released.

2026-09-08

- The processing-artifacts download (zip of everything for a request) is no longer part of the public API. Use the per-file download (`GET /order/{request_id}/{filename}`) instead.