Kneo Agent Client

User Guide

Version 0.1.0

2026-05-25

Quickstart, profiles and auth, idempotency and retries, pagination, error handling, and the compatibility matrix — one printable reading path for the kneo-client user documentation.

Table of contents

Quickstart

Source: [`docs/user/quickstart.md`]

This guide walks you from a fresh shell to your first authenticated call against a running Kneo Agent Platform instance, then to creating a run and inspecting its trace.

What "quickstart" means in kneo-client

`kneo-client` is a typed Python SDK and adapter toolkit for the Kneo Agent Platform's `/v1` HTTP API. Two products use it as their shared client layer — Kneo Agent Dashboard for operations and Kneo Agent Studio for development — but you can use it directly from any Python 3.12+ codebase that needs to talk to a platform instance.

A working setup needs three things:

1. The `kneo-client` package installed.
2. A **profile** — a `(url, api_key, auth_scheme, timeout)` tuple resolved from a TOML config file, environment variables, or explicit kwargs. See Profiles and auth for the full resolution model.
3. A reachable platform instance.

The rest of this page assumes you have all three.

Install

```
python -m pip install kneo-client
```

Requires Python ≥ 3.12. The runtime closure is small: `httpx`, `pydantic`, `anyio`, `platformdirs`, `attrs`. No CLI is installed — `kneo-client` is a library, not a service.

Verify the install:

```
python -c "from kneo_client import KneoClient, __version__; print(__version__)"
# → 0.1.0
```

Configure a profile

The client reads connection details in this order (later sources override earlier):

1. A TOML config file at `~/.config/kneo/client.toml` (XDG-style location via `platformdirs`).
2. Environment variables: `KNEO_PROFILE`, `KNEO_URL`, `KNEO_API_KEY`, `KNEO_AUTH_SCHEME`, `KNEO_TIMEOUT`.
3. Explicit keyword arguments to `KneoClient.from_profile()` (or the underlying `load_profile()`).

The minimum-friction setup is env vars — useful for one-off scripts and CI:

```
export KNEO_URL=https://kneo.example.com
export KNEO_API_KEY=your-api-key
```

For named multi-profile setups (e.g. `default` for prod, `staging` for non-prod), use the TOML file:

```
# ~/.config/kneo/client.toml
[default]
url = "https://kneo.example.com"
api_key = "prod-key"
auth_scheme = "bearer"
timeout = 30.0

[staging]
url = "https://staging-kneo.example.com"
api_key = "staging-key"
```

Switch profiles by name (`KneoClient.from_profile("staging")`) or by env (`KNEO_PROFILE=staging`).

See Profiles and auth for the full resolution semantics, both auth schemes, and the multi-profile workflow.

Ping the platform

The smallest possible script — verifies that auth flows correctly and the platform is reachable:

```
import asyncio
from kneo_client import KneoClient


async def main():
    async with KneoClient.from_profile() as client:
        ready = await client.platform.health.readyz()
        print(f"platform ready: ok={ready.ok} version={ready.service!r}")
```

```
asyncio.run(main())
```

What this exercises:

- Profile resolution (TOML / env / kwargs).
- API-key injection on the outgoing request (`Authorization: Bearer …` or `X-Kneo-Api-Key: …` depending on scheme).
- The transport's retry loop (will retry on transient network failures).
- Error mapping (a missing key surfaces as `KneoAuthError`, an unreachable URL as `KneoNetworkError`).

If you get a `KneoAuthError`, your API key isn't reaching the platform — re-check the profile resolution chain. If you get a `KneoNetworkError`, the URL is unreachable. See Error handling for the full hierarchy.

Create your first run

A run is the unit of work the platform executes — it instantiates a spec (an agent definition) and tracks its lifecycle through `queued → running → terminal {completed, failed, cancelled}`. To create one:

```
async with KneoClient.from_profile() as client:
    created = await client.platform.runs.create({"spec_id": "your-spec-id"})
    print(f"run_id={created.run_id} status={created.status}")

    terminal = await client.platform.runs.wait_for_completion(
        created.run_id, poll_interval=2.0, timeout=600
    )
    print(f"final status: {terminal.status}")

    trace = await client.platform.runs.trace(created.run_id, limit=20)
    for event in trace.events:
        print(event)
```

The interesting parts:

- **`runs.create(...)`** auto-injects an `Idempotency-Key` header (UUID4). Re-running the same payload with the same key is safe — the platform replays the original response. See Idempotency and retries.
- **`runs.wait_for_completion(...)`** polls `runs.get(run_id)` until the run reaches a terminal status. Default terminal set is `{"completed", "failed", "cancelled"}`; pass `terminal_statuses={"paused_human_review", …}` to treat additional states as terminal.

- `runs.trace(...)` returns events the platform recorded during the run (tool calls, model calls, middleware decisions, policy checks, etc.).

This is the operational core of the platform adapter. Every other endpoint follows the same shape: `client.platform.<resource>.<method>(...)` returning a typed model.

Sync facade

If your caller can't run an event loop (a script, a notebook, a sync framework), wrap the transport with `SyncTransport`:

```
 from kneo_client.core.profiles import load_profile
from kneo_client.core.transport import SyncTransport

with SyncTransport(load_profile()) as transport:
    response = transport.request("GET", "/v1/healthz")
    print(response.json())
```

Under the hood `SyncTransport` runs `Transport` inside an `anyio.from_thread.start_blocking_portal()` — same retry / idempotency / error flows, called synchronously. The async surface is the recommended path; the sync facade is for ergonomics.

`SyncTransport` does **not** mount `.platform` / `.agent` adapters — those are async-only. Sync consumers wanting wrapped endpoints either glue async-to-sync at the call site or drop to `transport.request(method, path, ...)` directly.

Where to go next

By topic:

- **Profiles and auth** — multi-profile workflows, the two header schemes the platform supports, environment-variable precedence.
- **Idempotency and retries** — when keys are auto-injected, how 409 mismatches surface, customizing the retry policy.
- **Pagination** — walking large result sets across list endpoints.
- **Error handling** — the full exception hierarchy and what to catch.
- **Compatibility matrix** — which `kneo-client` releases support which `kneo_serv` versions.

For the comprehensive API reference (every class, method, exception), see the API Reference HTML or the PDF version.

For runnable end-to-end scripts: `examples/`.

Profiles and auth

Source: *docs/user/profiles_and_auth.md*

This guide explains how `kneo-client` resolves connection details (URL, API key, auth scheme, timeout) into a profile, what the two API-key header schemes mean, and how to set up multi-profile workflows for dev / staging / prod.

What "profile" means in kneo-client

A **profile** is a frozen dataclass bundling `(name, url, api_key, auth_scheme, timeout)`:

```
@dataclass(frozen=True)
class Profile:
    name: str
    url: str
    api_key: str
    auth_scheme: AuthScheme = AuthScheme.BEARER
    timeout: float = 30.0
```

Everything `Transport` needs to talk to one Kneo Agent Platform instance fits in those five fields. A `KneoClient` is built around exactly one profile; you construct one per platform instance you talk to.

Profiles are typically named — `default`, `staging`, `prod`, `ci`, etc. The name is informational (it appears in log lines) but doubles as the section name in the TOML config file.

Resolution order

`load_profile()` and `KneoClient.from_profile()` merge values from three sources, with later sources overriding earlier ones:

1. **TOML config file**, by default `~/.config/kneo/client.toml` (XDG-style via `platformdirs`). Pass `config_file=Path(...)` to point at a different file. If the file doesn't exist, it's skipped silently — env vars and explicit kwargs are still consulted.
2. **Environment variables**: `KNEO_URL`, `KNEO_API_KEY`, `KNEO_AUTH_SCHEME`, `KNEO_TIMEOUT`. (`KNEO_PROFILE` selects which TOML section to load — it does *not* override field values.)
3. **Explicit keyword arguments** to the function call.

If `url` or `api_key` cannot be resolved from any source, a `ProfileError` is raised with details on which sources were checked. Bad TOML, an unknown `auth_scheme`, or a non-numeric `timeout` also surface as `ProfileError`.

```
from kneo_client.core.profiles import load_profile

p = load_profile()                          # 'default' from TOML + env
p = load_profile("staging")                 # explicit profile name
p = load_profile(url="https://ad-hoc", api_key=token)  # explicit kwargs win
```

TOML format

Each top-level table is one profile:

```
# ~/.config/kneo/client.toml
[default]
url = "https://kneo.example.com"
api_key = "prod-key"
auth_scheme = "bearer"        # or "kneo_api_key"
timeout = 30.0                # seconds

[staging]
url = "https://staging-kneo.example.com"
api_key = "staging-key"

[local]
url = "http://127.0.0.1:8000"
api_key = "dev-token"
auth_scheme = "kneo_api_key"
```

`auth_scheme` and `timeout` are optional; their defaults are `"bearer"` and `30.0`.

Picking a profile at call time:

```
client = KneoClient.from_profile()          # 'default' (or $KNEO_PROFILE)
client = KneoClient.from_profile("staging")  # explicit
client = KneoClient.from_profile("local")    # explicit
```

Environment variables

Variable	Purpose
`KNEO_PROFILE`	Profile name to load. Falls back to `"default"` if unset.
`KNEO_URL`	Override the profile's URL.

Variable	Purpose
`KNEO_API_KEY`	Override the profile's API key.
`KNEO_AUTH_SCHEME`	Override the scheme. Accepts `"bearer"` or `"kneo_api_key"`.
`KNEO_TIMEOUT`	Override the per-request timeout (float seconds).

CI environments typically set just `KNEO_URL` and `KNEO_API_KEY` and skip the TOML file entirely. The lack of a config file is *not* an error — env vars + kwargs can satisfy resolution on their own.

A bad value for `KNEO_TIMEOUT` (non-numeric) raises `ProfileError` with the variable name in the message — easier to debug than a silent fallback.

Auth schemes — what the platform accepts

The platform accepts the API key in either of two header schemes. They are *semantically* equivalent — both end up at the same platform code path — but operationally they have different trade-offs:

Scheme	Header sent	When to choose
`bearer` (default)	`Authorization: Bearer <key>`	Works with most reverse proxies. Easy to revoke at the gateway layer. Standard HTTP semantics.
`kneo_api_key`	`X-Kneo-Api-Key: <key>`	Useful when your edge stack already uses the `Authorization` header for something else (mutual TLS auth, an upstream OAuth flow, etc.). Avoids the collision.

When in doubt, start with `bearer`. You can switch schemes per-profile without code changes — just update the TOML or the env var.

The two schemes are implemented by `kneo_client.core.auth.ApiKeyAuth`, an `httpx.Auth` subclass that injects whichever header the active profile selects. Internally the auth flow runs *inside* httpx's request flow (after `Transport` has added its other headers), so the API key reaches every redirect / retry attempt at the right layer.

Multi-profile workflows

A common pattern in CI / local dev:

```
import os
from kneo_client import KneoClient

profile_name = "ci" if os.getenv("CI") else "default"
async with KneoClient.from_profile(profile_name) as client:
    ...
```

Or override explicitly when the situation calls for an ad-hoc connection:

```
async with KneoClient.from_profile(url="https://ad-hoc.example.com", api_key=tok) as client:
    ...
```

Explicit kwargs always win, so you can keep the TOML file as a baseline and override per-call.

Programmatic profile construction (when secrets come from a vault / secrets manager) skips `load_profile()` entirely:

```
from kneo_client import KneoClient
from kneo_client.core.auth import AuthScheme
from kneo_client.core.profiles import Profile

def profile_from_vault() -> Profile:
    secret = vault.get("kneo/prod")
    return Profile(
        name="prod",
        url=secret["url"],
        api_key=secret["api_key"],
        auth_scheme=AuthScheme.BEARER,
        timeout=30.0,
    )

async with KneoClient(profile_from_vault()) as client:
    ...
```

`Profile` is a frozen dataclass — pass it directly to `KneoClient(profile)`.

Inspecting a resolved profile

`KneoClient.profile` returns the `Profile` actually in use:

```
async with KneoClient.from_profile() as client:
    print(f"connected to {client.profile.url} as profile {client.profile.name!r}")
```

The `api_key` field is on the dataclass — handle it like any other secret. The redaction-aware logger in `kneo_client.core.logging` masks the key whenever it logs request / response headers, but **the dataclass itself is not redacted** when you print it directly.

If you do want to log a profile safely:

```
print(f"profile name={p.name!r} url={p.url!r} scheme={p.auth_scheme.value!r} timeout=
{p.timeout}")
```

…just exclude `p.api_key` from anything that goes to a log sink.

Profile errors

`ProfileError` covers four failure modes:

Trigger	Message pattern
Missing `url` after all sources	`"profile 'X': 'url' is not set …"`
Missing `api_key` after all sources	`"profile 'X': 'api_key' is not set …"`
Malformed TOML	`"failed to parse <path>: …"`
Unknown auth scheme	`"unknown auth_scheme '…'; expected one of: bearer, kneo_api_key"`
Non-numeric `KNEO_TIMEOUT`	`"$KNEO_TIMEOUT must be a float, got '…'"`

All five are explicit and name the offending source. Catch `ProfileError` (or `Exception` if you don't care which) at process startup to fail fast with a clear message rather than blowing up on the first call.

Default config path

`kneo_client.core.profiles.default_config_path()` returns the XDG-style default — `~/.config/kneo/client.toml` on Linux, `~/Library/Application Support/kneo/client.toml` on macOS, the appropriate `%APPDATA%\kneo\client.toml` on Windows. Resolved via `platformdirs.user_config_dir("kneo")`.

If you want a project-local TOML (committed to a repo, picked up by CI without needing a user-config), pass it explicitly:

```
from pathlib import Path
```

```
client = KneoClient.from_profile(config_file=Path(".kneo.toml"))
```

Idempotency and retries

Source: `docs/user/idempotency_and_retries.md`

This guide explains when `kneo-client` injects idempotency keys, how 409 mismatches surface, what the retry policy is, and how to customize either piece for non-default deployments.

What "idempotency and retries" mean in kneo-client

The Kneo Agent Platform's operational endpoints are designed for *safe retry*. The platform short-circuits a duplicate `POST` with the same `Idempotency-Key` header and identical payload — the second request replays the original response rather than re-executing the side effect.

`kneo-client` makes that safety automatic by:

1. Auto-injecting a fresh UUID4 `Idempotency-Key` on every `POST` (unless the caller supplies their own).
2. Retrying transient transport errors and the fixed set `{429, 502, 503, 504}` within a configurable `RetryPolicy`.
3. Honoring `Retry-After` on 429 responses (server's hint overrides the policy's computed delay).
4. Surfacing the platform's payload-mismatch behavior as `KneoIdempotencyMismatchError` so retry-with-wrong-payload bugs are loud, not silent.

Both pieces work together: idempotency keys make it *safe* to retry a `POST`; the retry policy *decides* when to retry. The default settings work for most callers; you can override either independently.

Idempotency keys, by default

On every `POST` the transport sends, it adds:

- `Idempotency-Key: <fresh UUID4>` — unless the caller passes `idempotency_key=<string>` on the method call.

The platform's contract:

- **Same key + identical payload** → server short-circuits and returns the original response. The retry is effectively a replay.
- **Same key + different payload** → server returns HTTP 409. The client surfaces this as `KneoIdempotencyMismatchError`.
- **Different key** → server treats this as a new request and executes the side effect.

The auto-generated key is a fresh UUID4 per request. Collisions are not a real concern (UUID4 has 122 bits of randomness), so by default each call is independent — retries beyond the transport's own loop won't be deduplicated by the server.

When to supply your own key

The auto-generated key is fine for one-shot calls. Supply your own when:

- **You're retrying outside the transport's loop.** The transport retries within `RetryPolicy.max_attempts` for transient failures, but if your *application* catches an error and retries (e.g. a job runner re-invoking after a process restart), pass the same key on both attempts so the platform dedupes. The transport's retries already share the key.
- **You want cross-process correlation.** Two services submitting the same logical request can dedupe by agreeing on the key (e.g. derive it from a hash of the request payload + a request ID from your application).
- **You're testing.** A stable key makes test fixtures deterministic.

```
from kneo_client.core.idempotency import new_idempotency_key

key = new_idempotency_key()
body = {"spec_id": "my-spec"}

# First attempt — succeeds normally
run = await client.platform.runs.create(body, idempotency_key=key)

# … later, retrying after a transient outage outside the transport's loop:
run = await client.platform.runs.create(body, idempotency_key=key)
# → returns the first run (same response, no new side effect)
```

Constraints on caller-supplied keys (validated by the client before sending):

- **Non-empty** — an empty string raises `ValueError`.
- **At most 256 characters** (`MAX_KEY_LENGTH`). The platform enforces this; the client validates locally to fail faster.

Catching 409 mismatch

A 409 with an idempotency key set means the *same key was reused with a different payload* — almost always a caller bug. Surface it loudly:

```
from kneo_client.core.errors import KneoIdempotencyMismatchError

try:
```

```
    await client.platform.runs.create(payload, idempotency_key=key)
except KneoIdempotencyMismatchError as exc:
    raise RuntimeError(
        f"Idempotency-Key {exc.idempotency_key!r} was reused with a different payload. "
        f"Either generate a new key for the new request or fix the payload drift."
    ) from exc
```

The exception carries the key as sent (`exc.idempotency_key`), the platform's error body (`exc.body`), the server-assigned request ID for log correlation (`exc.request_id`), and the HTTP status (`exc.status == 409`).

Note that `KneoIdempotencyMismatchError` is a *subclass* of `KneoConflictError`, so a generic `except KneoConflictError` will also catch it. If you only care about non-mismatch conflicts (resource state collisions, optimistic-lock failures, etc.), catch `KneoIdempotencyMismatchError` first.

The retry policy

`RetryPolicy` is a frozen dataclass describing when and how long to wait between attempts. The transport applies it; the policy itself does no I/O.

The default policy:

```
RetryPolicy(
    max_attempts=3,    # up to 3 total attempts
    base_delay=0.2,    # ~0.2s before attempt 2
    max_delay=30.0,    # cap on the computed delay
    jitter=0.1,        # 10% jitter applied to the computed delay
)
```

Delay sequence (without jitter): attempt 1 → no delay, attempt 2 → `base_delay`, attempt 3 → `2 × base_delay`, …, capped at `max_delay`. With `jitter=0.1`, a 1-second delay becomes uniformly distributed in `[0.9, 1.1]`.

`Retry-After` from a 429 response overrides the computed delay verbatim. The server's hint is authoritative; no jitter is applied on top.

When retries fire

The transport retries on:

- Transport-level errors from `httpx` — DNS resolution failures, connect failures, TLS handshakes, read timeouts.
- HTTP **429** (rate limited), **502** (bad gateway), **503** (service unavailable), **504** (gateway timeout). Other 4xx and 5xx status codes do *not* trigger a retry — those typically indicate

caller errors or non-transient server problems that retrying won't fix.

The retryable status set is `RETRYABLE_STATUS_CODES = frozenset({429, 502, 503, 504})` — a module constant, intentionally not configurable. If your platform deployment legitimately returns transient `500`s, fix the deployment; we'd rather diagnose the root cause than open the gate.

Retries fire only for:

- **Idempotent verbs**: `GET`, `HEAD`, `OPTIONS`, `PUT`, `DELETE`.
- **`POST` *with* an `Idempotency-Key`** — i.e. always, since the transport auto-injects one on every `POST`.

This is why auto-injection matters: it's what makes `POST` retries safe.

Customizing the policy

Pass a custom policy when constructing the client:

```
from kneo_client import KneoClient
from kneo_client.core.profiles import load_profile
from kneo_client.core.retries import RetryPolicy

profile = load_profile()

# Aggressive retries for a flaky network
policy = RetryPolicy(max_attempts=8, base_delay=0.5, max_delay=60.0)
client = KneoClient(profile, retry_policy=policy)

# Flat delay (no exponential growth)
policy = RetryPolicy(max_attempts=3, base_delay=2.0, max_delay=2.0, jitter=0)
client = KneoClient(profile, retry_policy=policy)
```

To disable retries entirely (useful in tests that want to see the first failure surface immediately):

```
client = KneoClient(profile, retry_policy=RetryPolicy(max_attempts=1))
```

Constraints on `RetryPolicy` parameters (validated at construction):

- `max_attempts ≥ 1`
- `base_delay ≥ 0`
- `max_delay ≥ base_delay`
- `0 ≤ jitter ≤ 1`

Invalid values raise `ValueError` immediately — you find out at startup, not on the first retry.

What surfaces when retries exhaust

If all retries fail, the client raises the appropriate typed exception based on the *final* attempt's outcome:

Final attempt failed because…	Exception raised
HTTP 429	`KneoRateLimited` (carries `.retry_after`)
HTTP 502 / 503 / 504	`KneoServerError`
Transport error (DNS, connect, TLS, read)	`KneoNetworkError`

Intermediate retry attempts are logged at `INFO` level under the `kneo_client.transport` logger:

```
INFO:kneo_client.transport:status 503 on attempt 1; sleeping 0.20s
INFO:kneo_client.transport:status 503 on attempt 2; sleeping 0.40s
```

If you want to see the retry behavior in your application logs, set the logger level:

```
import logging
logging.basicConfig(level=logging.INFO)
logging.getLogger("kneo_client.transport").setLevel(logging.INFO)
```

Bypassing the auto-injection

The transport auto-injects an `Idempotency-Key` on every `POST` *unconditionally*. If for some reason you need to send a `POST` without one (very unusual — the platform's contract assumes the header is present), drop to the transport directly and supply explicit headers:

```
# Standard call — transport adds Idempotency-Key automatically
await client.platform.runs.create(body)

# No-auto-inject path (you take responsibility for deduplication)
await client._transport.request("POST", "/v1/runs", json=body, headers={"Idempotency-Key":
"your-deterministic-key"})
```

You almost never want this. Auto-injection is the right default; this is documented mostly so you know the escape hatch exists.

Putting it together — a robust pattern

```
 from kneo_client import KneoClient
from kneo_client.core.errors import (
    KneoIdempotencyMismatchError, KneoNetworkError, KneoServerError,
)
from kneo_client.core.idempotency import new_idempotency_key

async def create_run_robust(client: KneoClient, body: dict) -> str:
    key = new_idempotency_key()  # one key, used across application-level retries

    for attempt in range(1, 4):
        try:
            run = await client.platform.runs.create(body, idempotency_key=key)
            return run.run_id

        except KneoIdempotencyMismatchError as exc:
            # Caller bug — body changed between attempts. Don't retry.
            raise RuntimeError(f"payload drift on key {exc.idempotency_key!r}") from exc

        except (KneoNetworkError, KneoServerError) as exc:
            # The transport already retried within its policy; we add an outer
            # guard for application-level recovery (e.g. across process restarts).
            if attempt == 3:
                raise
            await asyncio.sleep(2 ** attempt)

    raise AssertionError("unreachable")
```

In practice the transport's built-in retries are sufficient for most use cases; the outer loop above is for the rare case where you want application-level retry behavior (e.g. survive a process restart while the platform is temporarily unreachable).

Pagination

Source: *docs/user/pagination.md*

This guide explains the platform's `limit` / `offset` pagination protocol, how to walk one page or all pages, and how to use `iterate_all()` for streaming iteration across large result sets.

What "pagination" means in kneo-client

Every list endpoint the platform exposes — runs, audit events, human tasks, environment policies, traces, checkpoints — uses the same `limit` / `offset` pagination protocol. The client surfaces that protocol through:

- Uniform keyword arguments on every list method (`limit`, `offset`, `sort_by`, `sort_order`).
- A `PaginatedResult[T]` wrapper class in `kneo_client.core.pagination` for typed page-by-page handling.
- An `iterate_all()` async iterator that walks pages transparently given a `fetch_page(limit, offset)` callable.

The protocol

Platform list endpoints accept:

Query parameter	Type	Meaning
`limit`	int (1–1000)	Page size. Default 100.
`offset`	int	Skip this many items.
`sort_by`	str	Field to sort by. Defaults vary per endpoint (`updated_at` for runs, `timestamp` for audit, etc.).
`sort_order`	`"asc"` or `"desc"`	Sort direction. Defaults vary per endpoint.

Responses include:

Response field	Meaning
`count`	Items on this page (`len(items)`).
`total`	Total items across all pages, as reported by the server.
`limit`	The page size the server actually applied (may equal the requested value).
`offset`	The offset of the first item on this page.
`sort_by`, `sort_order`	Echo of the sort parameters in effect.
Items array	Endpoint-specific name (`runs`, `events`, `tasks`, `checkpoints`, …).

Concrete shapes vary per endpoint — the items array is named after the resource (`page.runs`, `page.events`, etc.). See the API Reference for each endpoint's exact response model.

Walking one page

Every platform list method exposes the same kwargs:

```
page = await client.platform.runs.list(
    status="running",
    limit=50,
    offset=0,
    sort_by="updated_at",
    sort_order="desc",
)
print(f"got {page.count} of {page.total} runs")
for run in page.runs:
    print(run)
```

`None` for any keyword argument means *omit* — the platform's default kicks in. The client passes through whatever the server returns; it doesn't second-guess the page size or impose its own defaults beyond passing through your input.

Walking all pages manually

The straightforward pattern works for any list endpoint:

```
async def all_runs(client, **filters):
    offset = 0
    page_size = 200
    while True:
        page = await client.platform.runs.list(
            limit=page_size, offset=offset, **filters
        )
        for run in page.runs:
            yield run
        if page.count < page_size:
            break
        offset += page.count
```

This pattern uses `count < page_size` as the end-of-iteration signal, which works whether or not the server returns a `total`. It's also robust to a server that returns fewer items than requested (e.g., quota / rate limiting on a per-page basis).

`examples/03_paginate_audit.py` does exactly this for audit events.

The `iterate_all()` helper

`kneo_client.core.pagination.iterate_all()` is an async iterator that walks pages given a `fetch_page(limit, offset)` callable returning a `PaginatedResult`:

```
from kneo_client.core.pagination import PaginatedResult, iterate_all

async for item in iterate_all(fetch_page, page_size=200):
    process(item)
```

The platform list methods don't yet return `PaginatedResult` directly — that integration is a known follow-up — so today you adapt the call site:

```
from kneo_client.core.pagination import PaginatedResult, iterate_all

async def all_audit_events(client, **filters):
    async def fetch_page(limit: int, offset: int) -> PaginatedResult:
        resp = await client.platform.audit.list(limit=limit, offset=offset, **filters)
        return PaginatedResult(
            items=resp.events,
            total=getattr(resp, "total", 0) or 0,
            limit=limit,
            offset=offset,
        )

    async for event in iterate_all(fetch_page, page_size=200):
```

```
        yield event

# Use it:
async for event in all_audit_events(client, event_type="run.created"):
    print(event)
```

`iterate_all()` does three things on your behalf:

1. **Clamps `page_size` to `MAX_PAGE_SIZE = 1000`** — the platform's hard upper bound. Asking for more than 1000 silently downsizes.
2. **Walks `offset` automatically** — each page's `offset` becomes the next page's starting position.
3. **Stops when `has_more` is false** — the `PaginatedResult.has_more` property is `offset + count < total`, which works whenever the response includes a `total`. For responses that don't, build the `PaginatedResult` with `total=count` and the iteration stops after one page.

`PaginatedResult[T]` — the typed wrapper

```
@dataclass(frozen=True)
class PaginatedResult(Generic[T]):
    items: list[T]
    total: int
    limit: int
    offset: int
    sort_by: str | None = None
    sort_order: str | None = None

    @property
    def count(self) -> int:
        return len(self.items)

    @property
    def has_more(self) -> bool:
        return self.offset + self.count < self.total
```

Use it when building your own page-walker (as above) or when you want a uniform shape across endpoints regardless of what the underlying response model is named.

Choosing a page size

A few rules of thumb:

- **100–200** for most interactive flows. Small enough to keep latency snappy; large enough to amortize request overhead.
- **500–1000** for back-end export jobs that walk the whole list and don't care about first-byte latency. Larger pages reduce request count.
- **< 50** if downstream processing per item is slow and you want to start producing output sooner.

Larger pages reduce per-request overhead but increase the cost of a failed page — everything in flight has to be re-fetched. If the platform deployment you're talking to is on a flaky network, prefer smaller pages.

The maximum is `MAX_PAGE_SIZE = 1000` (the platform's enforced upper bound). Anything larger is silently clamped by `iterate_all()` and by the platform itself.

Pagination + filters

All list methods accept resource-specific filter kwargs alongside the pagination args. Common patterns:

```
 # Just the failures
failed_runs = await client.platform.runs.list(status="failed")

# Just audit events for one run
audit_for_run = await client.platform.audit.list(run_id="r1")

# Just pending human tasks
pending = await client.platform.human_tasks.list(status="pending")
```

Filters compose with pagination — `runs.list(status="failed", limit=50)` filters then paginates the filtered result.

What's *not* on the roadmap

Auto-pagination at the adapter layer (e.g., `client.platform.runs.list_all()` returning an iterator) is *not* planned. The current shape — list methods return one page; `iterate_all()` walks pages explicitly — keeps the per-call cost transparent and lets callers decide when to stop. Auto-walking can mask runaway iteration if a filter accidentally matches a huge result set.

If you want a one-liner for the common case, write a small helper in your application like `all_runs()` above. The pattern is identical for every endpoint.

Error handling

Source: `docs/user/errors.md`

This guide explains the typed exception hierarchy `kneo-client` raises, what each exception carries, and how to write robust catch blocks for the common operational shapes.

What "errors" mean in kneo-client

Every failure — at any layer — surfaces as a typed exception derived from `KneoError`. There is no `(ok, err)` tuple return, no `Response[T]` wrapper, no errno field. The standard Python `try` / `except` flow is the *only* error-handling shape.

Each exception carries enough context to:

- **Log the failure with traceability** — every exception has `.request_id` (the server-assigned correlation ID) and, for `POST` failures, `.idempotency_key`.
- **Branch on the operational meaning** — the exception class encodes "what went wrong" (auth vs. permission vs. server outage vs. network).
- **Read the server's reason** — `.body` is the parsed JSON the platform returned (or raw text when the response wasn't JSON).
- **Decide whether to retry, escalate, or surface** — combined with `.status` and the exception type, you can route the failure programmatically.

Hierarchy

```
KneoError
├── KneoNetworkError              # DNS / connect / TLS / read timeout — wrapped from
httpx.HTTPError
├── KneoAuthError                 # HTTP 401 — missing or invalid API key
├── KneoPermissionError           # HTTP 403 — key valid but lacks the required scope
├── KneoNotFoundError             # HTTP 404 — resource does not exist
├── KneoConflictError             # HTTP 409 (generic)
│   └── KneoIdempotencyMismatchError   # HTTP 409 with payload mismatch on a replayed
Idempotency-Key
├── KneoRateLimited               # HTTP 429 (carries .retry_after)
└── KneoServerError               # HTTP 5xx — server-side failure
```

`KneoIdempotencyMismatchError` is a *subclass* of `KneoConflictError` (catching `KneoConflictError` also catches the mismatch case). Everything else is parallel.

What every exception carries

```
class KneoError(Exception):
    status: int | None          # HTTP status, or None for transport-level failures
    body: Any                   # Parsed JSON dict, raw text, or None
    request_id: str | None      # X-Request-ID echoed by the server
    idempotency_key: str | None # Idempotency-Key sent on the failing request (POSTs)
```

`KneoRateLimited` adds one field:

```
class KneoRateLimited(KneoError):
    retry_after: float | None   # Seconds parsed from the Retry-After header
```

All other subclasses inherit from `KneoError` without adding fields.

Status code → exception mapping

HTTP status	Exception	When
401	`KneoAuthError`	Missing or invalid API key. Re-check the profile resolution chain.
403	`KneoPermissionError`	API key is valid but the platform won't authorize this operation. Check the key's scopes / role.
404	`KneoNotFoundError`	The resource (run, spec, environment, etc.) doesn't exist. Often a stale ID.
409	`KneoConflictError`	Generic conflict — resource state, optimistic-lock failure, etc.
409 with idempotency key	`KneoIdempotencyMismatchError`	Same key reused with a different payload — see idempotency.
429	`KneoRateLimited`	Rate limit hit. `.retry_after` carries the server's hint.
5xx	`KneoServerError`	Server-side failure. The transport already retried within

HTTP status	Exception	When
		its policy (for 502/503/504); a `KneoServerError` reaching your code means retries exhausted.
Other (1xx / 3xx / 4xx that aren't above)	`KneoError` (base)	Catch-all for unmodeled statuses.
Connection / DNS / TLS / read timeout	`KneoNetworkError`	Transport-level failure. The transport already retried for transient errors; a `KneoNetworkError` reaching your code means retries exhausted.

Catching patterns

Catch broadly, log richly

The most common pattern — log the full context, then decide whether to re-raise:

```
 from kneo_client.core.errors import KneoError

try:
    run = await client.platform.runs.create(payload)
except KneoError as exc:
    log.error(
        "create_run failed status=%s request_id=%s idempotency_key=%s body=%r",
        exc.status,
        exc.request_id,
        exc.idempotency_key,
        exc.body,
    )
    raise
```

The `request_id` is the link to the platform's audit events — pass it along when reporting a problem to the platform operators.

Branch on specific status

When the operational meaning matters (auth flow, retry decision, user-visible error message):

```
from kneo_client.core.errors import (
    KneoAuthError,
    KneoNotFoundError,
    KneoRateLimited,
    KneoServerError,
)

try:
    run = await client.platform.runs.get(run_id)
except KneoAuthError:
    print("API key is missing, invalid, or revoked.")
    raise
except KneoNotFoundError:
    print(f"run {run_id!r} does not exist.")
    return None
except KneoRateLimited as exc:
    print(f"rate-limited; server suggests waiting {exc.retry_after}s")
    await asyncio.sleep(exc.retry_after or 10)
    raise
except KneoServerError as exc:
    log.error("platform 5xx (after retries): %s", exc.body)
    raise
```

Order matters: catch more specific exceptions first (`KneoNotFoundError` before `KneoError`).

Handle idempotency-key mismatches loudly

A 409 with an idempotency key set means the *same key was reused with a different payload*. This is almost always a caller bug — surface it explicitly:

```
from kneo_client.core.errors import KneoIdempotencyMismatchError

try:
    await client.platform.runs.create(payload, idempotency_key=key)
except KneoIdempotencyMismatchError as exc:
    raise RuntimeError(
        f"Idempotency-Key {exc.idempotency_key!r} was reused with a different payload. "
        f"Either generate a new key for the new request or fix the payload drift."
    ) from exc
```

See Idempotency and retries for the full story on how / when this happens.

Don't catch successful-status branches

Methods on the platform / agent clients return parsed response models on success and *raise* on failure. There is no "ok / err" branching at the call site. Wrap the *call* in `try` / `except`, not the return value:

```
# Right
try:
    run = await client.platform.runs.create(payload)
    process(run)
except KneoError:
    ...

# Wrong — runs.create never returns None / False on failure; it raises
result = await client.platform.runs.create(payload)
if result is None:  # never happens
    ...
```

Transport errors specifically

`KneoNetworkError` covers everything below the HTTP layer: DNS resolution, TCP connect failures, TLS handshakes, read timeouts. It wraps the underlying `httpx.HTTPError` as the cause — `exc.__cause__` is the original httpx exception if you need to inspect it.

The transport retries these automatically within `RetryPolicy.max_attempts` for transport errors and for HTTP 429 / 502 / 503 / 504. So a `KneoNetworkError` reaching your code means **all retries exhausted**:

```
from kneo_client.core.errors import KneoNetworkError

try:
    health = await client.platform.health.readyz()
except KneoNetworkError as exc:
    print(f"could not reach the platform: {exc}")
    # Treat as a hard dependency outage; don't pretend the call succeeded.
    raise
```

If you want to see the retry behavior in your logs, set the `kneo_client.transport` logger to `INFO`:

```
import logging
logging.getLogger("kneo_client.transport").setLevel(logging.INFO)
# → INFO kneo_client.transport: transport error on attempt 1; sleeping 0.20s: ...
```

Building error messages for users

The exceptions are designed for *internal* error handling, not for user-facing messages. If you're surfacing platform errors to end users (in a dashboard UI, a CLI prompt, etc.), build a friendly message from the exception's attributes rather than printing `str(exc)`:

```
def user_message(exc: KneoError) -> str:
    if isinstance(exc, KneoAuthError):
        return "Your API key is invalid. Please check your credentials."
    if isinstance(exc, KneoPermissionError):
        return "You don't have permission to perform this action."
    if isinstance(exc, KneoNotFoundError):
        return "The requested item could not be found."
    if isinstance(exc, KneoRateLimited):
        wait = exc.retry_after or 60
        return f"Too many requests. Please try again in {int(wait)}s."
    if isinstance(exc, KneoServerError):
        return f"Server error (request {exc.request_id}). Please report this."
    if isinstance(exc, KneoNetworkError):
        return "Could not reach the server. Check your network connection."
    return f"Unexpected error: {exc}"
```

Why a typed hierarchy

A single `KneoError` would force callers to inspect `.status` everywhere they want to branch. A flat enum of error codes would lose the natural `isinstance` ergonomics. The hierarchy lets you catch broadly (`except KneoError`) for logging and narrowly (`except KneoAuthError`) for recovery — without losing the underlying response context, which stays attached to the exception instance.

Subclasses are added when the platform introduces a new HTTP status that warrants its own catch site, and not before. The set above is sufficient for `/v1` as it stands at `kneo_serv` 0.4.0.

Reference

Helper	Where	What it does
`from_response(response, *, idempotency_key=None)`	`kneo_client.core.errors`	Maps an `httpx.Response` to the appropriate `KneoError` subclass. Used internally by `Transport`; rarely called directly.
`KneoError(message, *, status, body, request_id, idempotency_key)`	`kneo_client.core.errors`	The base. All subclasses share this constructor signature (except `KneoRateLimited`, which adds `retry_after`).

Compatibility matrix

Source: *docs/user/compatibility.md*

This guide tells you which `kneo-client` release supports which `kneo_serv` platform version, what forward and backward compatibility mean in practice, and how to use the drop-to-transport escape hatch when you need access to an endpoint the current `kneo-client` release doesn't wrap yet.

What "compatibility" means in kneo-client

The Kneo Agent Platform's `/v1` HTTP API is a stability boundary — a `kneo-client` release pinned to one `kneo_serv` minor works against any patch-level `kneo_serv` release on the same minor line, and against newer minors that don't break `/v1`.

The pinning is *explicit* and *committed*: `schemas/openapi.json` is a `/v1`-filtered copy of one specific `kneo_serv` release's published OpenAPI spec. Bumping the pin is a deliberate PR (via `scripts/bump_schemas.py`), reviewed in isolation, and never happens automatically. See ADR-004 for the rationale.

Current matrix

`kneo-client`	Pinned to `kneo_serv`	Tested against	Python	Status
`0.1.0`	`v0.4.0` (`info.version` `0.4.0`)	`kneo_serv` `0.4.x` line	`>=3.12`	First release

The pinned `kneo_serv` version is recorded in `schemas/SOURCE.md` — that's the source of truth for which platform version generated the committed `_generated/` tree.

How pinning works in practice

The pin is the input to the generated layer. When you bump it:

1. `scripts/bump_schemas.py` fetches the new `openapi.json` from the target `kneo_serv` ref (or a local checkout).
2. The spec is filtered to `/v1` paths only — `kneo_serv` mounts every route at both `/v1/…` and `/…`; we drop the unprefixed mounts.
3. The filtered spec replaces `schemas/openapi.json`, `schemas/SOURCE.md` is updated, and `_generated/` is regenerated.

4. The hand-rolled adapter layer (`platform/`, `agent/`) may need updates if endpoints were added, renamed, or had their shapes change. The contract test `tests/contract/test_path_coverage.py` catches both sides of that drift.

A `kneo-client` minor release ships exactly one `kneo_serv` pin. Patches don't change pins. A pin bump can land in any minor / patch — versioning is independent.

Forward compatibility — newer `kneo_serv` than the pin

`kneo-client` ships an explicit list of every `(method, path)` it wraps. A newer `kneo_serv` that adds endpoints will still work for everything `kneo-client` already wraps — you just won't have wrappers for the new endpoints until the next `kneo-client` release.

If you need access to a new endpoint before that release, drop to the raw transport:

```
async with KneoClient.from_profile() as client:
    # Call an endpoint that doesn't have a wrapper yet:
    resp = await client._transport.request("GET", "/v1/some/new/endpoint")
    payload = resp.json()
```

The `_transport` attribute is informally accessible (single-underscore prefix). It's not part of the documented stability surface, but the API has been stable since `0.1.0` and is unlikely to churn. The transport handles auth, retries, idempotency, request-ID injection, and error mapping just like the wrapped methods do — you just lose the typed response model.

Backward compatibility — older `kneo_serv` than the pin

`kneo-client X.Y.Z` is **not** guaranteed against `kneo_serv` releases older than its pin. Wrappers may rely on response fields that older `kneo_serv` versions don't emit, and the client's error mapping assumes the current platform error shape. A `KeyError` on `from_dict()` is the typical symptom — the wrapper expects a field that wasn't in the older platform's response.

If you have to talk to an older `kneo_serv`, pin to a matching `kneo-client` minor:

Need to talk to `kneo_serv` …	Use `kneo-client`
`0.4.x`	`0.1.x`

(More rows added as the project ships.)

Breaking changes

A breaking change in either direction triggers a major bump:

- **`kneo-client` major** — the public Python API (`kneo_client.*` namespace) changes incompatibly. Very rare.
- **`kneo_serv` major** — i.e., introduction of `/v2`. `kneo-client` may need a major bump if `/v1` is sunset; otherwise it ships a new minor that supports both.

Within a major, deprecated surfaces keep aliases for at least one minor. See `docs/dev/contributing.md` for the deprecation policy.

Verifying compatibility yourself

The simplest smoke is the bundled examples:

```
 # Install a specific kneo-client version
python -m pip install "kneo-client==X.Y.Z"

# Point it at your kneo_serv instance
export KNEO_URL=https://your-kneo-serv.example.com
export KNEO_API_KEY=...

# Run the smoke (touches health, runs, audit, agent specs, and human tasks)
python -m kneo_client.examples.01_basic_run YOUR_SPEC_ID
```

The five `examples/` scripts collectively touch the platform health, runs, audit, agent spec, and human-task surfaces — enough to catch obvious incompatibilities in seconds.

For deeper validation, the integration test suite is env-gated; set `KNEO_TEST_URL` and `KNEO_TEST_API_KEY` and run `python -m pytest tests/integration -v`.

What the pin does *not* guarantee

The pin guarantees the *wire format* and the *path set* of `/v1`. It does not guarantee:

- **Provider availability** — whether your `kneo_serv` deployment has GPT-4 configured, an MCP server reachable, a specific runtime registered. The pin says nothing about deployment state.
- **Spec compatibility** — a spec that works on one `kneo_serv` may fail on another if the spec relies on a specific provider, tool, or platform version. Validate specs against the target environment with `client.agent.specs.validate(...)`.
- **Policy outcomes** — `policies.environment_*` queries return whatever policy is configured on the target deployment.

These are platform-deployment concerns, not client-library concerns. The client gives you a clean wire to the platform; what the platform allows is a separate dimension.