info@jana.earth

Jana Eko Client — User Guide

Version: 0.2.0 **Last updated:** 2026-03-23

The `jana-eko-client` is a Python SDK for the Jana Earth Unified Environmental Data API. It provides typed, paginated access to six global environmental datasets — OpenAQ (air quality), Climate TRACE (greenhouse gas emissions), EDGAR (emissions inventories), GLEIF (legal entity identifiers), GCP (Global Carbon Project), and NOAA Storm Events — plus a unified analytics layer.

Table of Contents

Installation

Prerequisites

- **Python 3.10 or 3.11** (recommended). Python 3.8+ is supported but 3.10+ avoids SSL/LibreSSL warnings on macOS.
- **pip** (comes with Python)
- **Git** (for installing from GitHub)

Install from GitHub

```
pip install git+https://github.com/Jana-Earth-Data/jana-eko-client.git@development
```

To pin to a specific version or commit:

```
# Pin to a tag
pip install git+https://github.com/Jana-Earth-Data/jana-eko-client.git@v0.2.0

# Pin to a commit SHA
pip install git+https://github.com/Jana-Earth-Data/jana-eko-client.git@abc1234
```

Dependencies (installed automatically)

Package	Version	Purpose
`httpx`	>= 0.24.0	Async/sync HTTP client
`pydantic`	>= 2.0.0	Response model validation
`typing-extensions`	>= 4.5.0	Backported type hints
`nest-asyncio`	>= 1.5.0	Jupyter event loop compatibility

Environment Setup

Virtual environment (recommended)

Always use a dedicated virtual environment. Never install into system Python.

```
cd /path/to/data-analyst

# Create venv with Python 3.11
python3.11 -m venv .venv
source .venv/bin/activate   # macOS/Linux

# Install dependencies
pip install --upgrade pip
pip install -r requirements.txt
pip install git+https://github.com/Jana-Earth-Data/jana-eko-client.git@development
```

Jupyter kernel registration

```
python -m ipykernel install --user --name=data-analyst --display-name "Python (data-analys
```

Then select **"Python (data-analyst)"** in Jupyter or VS Code's kernel picker.

Verify installation

```
from eko_client import EkoUserClient
print("eko_client imported successfully")
```

Authentication

The Eko client supports three authentication methods. **Device code flow is recommended** for interactive use.

Method 1: Device Code Flow (recommended)

No password handling. Opens a browser for approval.

```
from eko_client import EkoUserClient

client = EkoUserClient(
    base_url="https://auth-dev.jana.earth",
    api_base_url="https://api-test.jana.earth",
)
client.login_device()
# A URL is printed to the console. Open it in your browser.
# Approve the login on the Jana web app.
# The client polls until approved and stores JWT tokens automatically.
```

Method 2: Email + Password

```
client = EkoUserClient(
    base_url="https://auth-dev.jana.earth",
    api_base_url="https://api-test.jana.earth",
)
client.login_password("your-email@example.com", "your-password")
```

Method 3: Environment variables (for notebooks)

Set credentials before launching Jupyter:

```
export JANA_AUTH_URL="https://auth-dev.jana.earth"
export JANA_API_URL="https://api-test.jana.earth"
export JANA_EMAIL="your-email@example.com"
export JANA_PASSWORD="your-password"
```

Then in a notebook:

```
import os
from eko_client import EkoUserClient

client = EkoUserClient(
    base_url=os.environ["JANA_AUTH_URL"],
```

```
    api_base_url=os.environ["JANA_API_URL"],
)
client.login_password(os.environ["JANA_EMAIL"], os.environ["JANA_PASSWORD"])
```

Dual URL architecture

The client uses two URLs:

URL	Purpose	Service
`https://auth-dev.jana.earth`	Authentication (`/api/auth/*`)	jana-user
`https://api-test.jana.earth`	Data endpoints (`/api/v1/*`)	jana-api-internal

Both point to the same ALB but route to different ECS services. Pass `base_url` for auth and `api_base_url` for data.

Token lifecycle

- **Access token:** 15 minutes. The client automatically refreshes when expired.
- **Refresh token:** 24 hours. If both expire, `EkoSessionExpiredError` is raised — re-login required.
- **Auto-refresh:** Transparent. If a request gets a 401, the client refreshes and retries once.

```
# Manual refresh (rarely needed)
client.refresh_jwt()

# Check if authenticated
client.is_authenticated()  # True/False

# Get current user info
client.get_user_info()
# {'id': 1, 'email': 'admin@jana.earth', 'first_name': 'Admin', ...}
```

Quick Start

```
from eko_client import EkoUserClient

# Connect
client = EkoUserClient(
    base_url="https://auth-dev.jana.earth",
    api_base_url="https://api-test.jana.earth",
)
client.login_device()

# Check platform health
health = client.get_health()
print(health)
# {'service': 'esg', 'status': 'healthy', 'version': 'v1', 'timestamp': None}
```

```
# Get platform summary (2.1 billion records across 3 data sources)
summary = client.get_summary()
print(f"Total records: {summary['platform_overview']['total_records']:,}")
# Total records: 2,157,471,050

# Get Nepal's CO2 emissions from EDGAR
nepal_co2 = client.get_edgar_country_totals(country_code="NPL", gas="CO2")
for row in nepal_co2["results"][:3]:
    print(f"  {row['year']}: {float(row['value']):,.0f} kt CO2")
# 2024: 15,892 kt CO2
# 2023: 15,432 kt CO2
# 2022: 15,045 kt CO2

# Get air quality monitoring stations in Nepal
stations = client.get_openaq_locations(country_codes="NP")
print(f"Nepal has {stations['count']} air quality monitoring stations")
# Nepal has 121 air quality monitoring stations

# Clean up
client.close()
```

Client Architecture

Sync and async

Every method has both a synchronous and asynchronous version. The sync version is auto-generated from the async version at import time.

```
# Synchronous (default for scripts and notebooks)
data = client.get_edgar_country_totals(country_code="NPL")

# Asynchronous (for async frameworks)
data = await client.get_edgar_country_totals_async(country_code="NPL")
```

The client auto-detects Jupyter environments and handles event loop compatibility via `nest_asyncio`.

Context manager

```
with EkoUserClient(
    base_url="https://auth-dev.jana.earth",
    api_base_url="https://api-test.jana.earth",
) as client:
    client.login_device()
    data = client.get_health()
# Connection automatically closed
```

Thread safety

The client uses per-instance locks for both async and sync operations. Safe for multi-threaded use.

OpenAQ — Air Quality Data

OpenAQ aggregates air quality measurements from government agencies, research institutions, and low-cost sensor networks worldwide. It provides real-time and historical observations of criteria air pollutants (PM2.5, PM10, NO2, O3, SO2, CO) and other atmospheric parameters.

The Jana database contains **~1.49 billion measurements** from **50,514 monitoring stations** (hosting 311,602 sensors) across **159 countries**, spanning 2016-01-30 to the present.

Key concepts:

- **Location** — A physical monitoring station (e.g., "Embassy Kathmandu") with fixed geographic coordinates.
- **Sensor** — An individual measurement instrument at a location. One location may have multiple sensors measuring different parameters.
- **Parameter** — The air quality variable being measured (e.g., `pm25` = PM2.5 fine particulate matter in ug/m3).

Locations

Air quality monitoring stations worldwide.

```
# Get monitoring stations in Nepal
locations = client.get_openaq_locations(country_codes="NP", limit=2)
print(f"Total stations in Nepal: {locations['count']}")
# Total stations in Nepal: 121

# First result
loc = locations["results"][0]
print(f"Name: {loc['name']}")
print(f"Country: {loc['country_code']}")
print(f"Lat/Lon: {loc['coordinates']['latitude']}, {loc['coordinates']['longitude']}")
```

Output:

```
Name: Nepal Military Academy-333819
Country: NP
Lat/Lon: 27.68604, 85.45054
```

Parameters:

Parameter	Type	Description
`country_codes`	str or list	ISO 2-letter codes (e.g. "NP", ["NP", "IN"])
`location_bbox`	list[float]	Bounding box [min_lon, min_lat, max_lon, max_lat]

Parameter	Type	Description
`page`	int	Page number (1-based)
`page_size`	int	Results per page (default varies)
`limit`	int	Alias for page_size
`offset`	int	Offset for pagination

Get a specific location

```
location = client.get_openaq_location(118068)
print(f"{location['name']} ({location['country_code']})")
# Nepal Military Academy-333819 (NP)
```

Sensors

Individual measurement sensors at monitoring stations.

```
sensors = client.get_openaq_sensors(country_code="NP", limit=2)
print(f"Total sensors: {sensors['count']}")
# Total sensors: 320,783

sensor = sensors["results"][0]
print(f"Parameter: {sensor['parameter_name']} ({sensor['parameter_units']})")
```

Additional filter parameters:

Parameter	Type	Description
`location_id`	int	Filter by specific location
`parameter`	str	Filter by parameter name (e.g. "pm25", "no2")
`location_bbox`	list[float]	Spatial bounding box
`coordinates`	list[float]	Point [lon, lat] for radius search
`radius_km`	float	Radius in km (used with coordinates)

Measurements

Individual air quality readings.

```
# Get PM2.5 measurements for a specific location
measurements = client.get_openaq_measurements(
    location_id=118068,
    parameter="pm25",
    limit=5,
)
for m in measurements["results"]:
    print(f"  {m['datetime']}: {m['value']} {m['parameter_units']}")
```

Parameters:

Parameter	Type	Description
`location_id`	int	Filter by location
`parameter`	str	Filter by parameter name
`date_from`	str/datetime	Start date (ISO 8601)
`date_to`	str/datetime	End date (ISO 8601)
`country_code`	str	ISO 2-letter country code
`ordering`	str	Sort field (e.g. "-datetime" for newest first)

Measurement aggregates

```
# Date range of available measurements
date_range = client.get_openaq_measurements_date_range()
print(f"From: {date_range['earliest_date']}")
print(f"To:   {date_range['latest_date']}")
print(f"Total: {date_range['total_records']:,}")
# From: 2016-01-30T00:00:00+00:00
# To:   2026-01-01T10:00:00+00:00
# Total: 1,489,718,052

# Global totals
totals = client.get_openaq_measurements_totals()
print(f"Records: {totals['record_count']:,}")
# Records: 1,489,718,016
```

Totals by parameter

```
param_totals = client.get_openaq_measurements_parameter_totals()
for p in param_totals[:5]:
    print(f"  {p['parameter_name']:12s} ({p['unit']:10s}) "
          f"records={p['record_count']:>12,}  avg={p['avg_value']:.2f}")
```

Output:

```
  pm25         (ug/m3     ) records= 301,939,503  avg=28.44
  pm10         (ug/m3     ) records= 198,224,211  avg=92.97
  pm1          (ug/m3     ) records= 111,379,130  avg=14.82
  no2          (ug/m3     ) records=  78,725,120  avg=1256.85
  o3           (ug/m3     ) records=  55,791,037  avg=49.35
```

Totals by country

```
country_totals = client.get_openaq_measurements_country_totals()
for c in country_totals[:5]:
    print(f"  {c['country_code']:5s} records={c['record_count']:>12,}  avg={c['avg_value']
```

Output:

```
  US    records= 498,590,916  avg=68.39
  BG    records= 240,991,372  avg=267891.28
  JP    records= 114,892,631  avg=2.51
  IN    records=  99,225,392  avg=66.55
  ES    records=  54,440,254  avg=62.95
```

> **Note:** The high average for Bulgaria (BG) reflects mixed unit scales across parameters (e.g., ppb vs ug/m3). Filter by a specific parameter for meaningful cross-country comparisons.

Parameters catalog

```
params = client.get_openaq_parameters()
print(f"Available parameters: {params['count']}")
# Available parameters: 30

for p in params["results"][:5]:
    print(f"  {p['name']:12s} {p['display_name']:10s} ({p['units']})")
```

Output:

```
  bc           BC         (ug/m3)
  bc_370       BC 370nm   (ng/m3)
  bc_375       BC 375nm   (ng/m3)
  bc_470       BC 470nm   (ng/m3)
  bc_528       BC 528nm   (ng/m3)
```

Common parameters: `pm25` (PM2.5), `pm10` (PM10), `no2` (NO2), `o3` (O3), `so2` (SO2), `co` (CO).

Climate TRACE — Greenhouse Gas Emissions

Climate TRACE is a global coalition that uses satellite observation, remote sensing, and AI to independently track greenhouse gas (GHG) emissions at the facility level. Unlike government-reported inventories (which lag by years), Climate TRACE provides near-real-time emission estimates for individual power plants, factories, farms, and other assets worldwide.

The Jana database contains **53.5 million emission records** across 26 sectors, 251 countries, and 1.3 million tracked assets. Data spans from 2021 to the present, with monthly granularity.

Key concepts:

- **Asset** — A physical facility or geographic area (e.g., a power plant, a crop region, a road segment) that emits greenhouse gases.
- **Sector** — A category of emission source (e.g., "agriculture", "transportation", "buildings"). Climate TRACE defines 26 sectors.
- **CO2e (CO2 equivalent)** — A standardized unit that converts all greenhouse gases (CO2, CH4, N2O, etc.) to their equivalent warming impact in tonnes of CO2.

Sectors

```
sectors = client.get_climatetrace_sectors()
print(f"Total sectors: {sectors['count']}")
# Total sectors: 26

for s in sectors["results"][:5]:
    print(f"  {s['display_name']:20s} assets={s['assets_count']:,}")
```

Output:

```
  Agriculture          assets=1,002,010
  Aluminum             assets=1
  Buildings            assets=114,070
  Chemicals            assets=0
  Coal Mining          assets=0
```

Countries

```
countries = client.get_climatetrace_countries()
print(f"Countries with data: {countries['count']}")
# Countries with data: 251

for c in countries["results"][:3]:
    print(f"  {c['iso3']}: {c['assets_count']:,} assets")
```

Output:

```
  ABW: 20 assets
  AFG: 3,601 assets
  AGO: 2,184 assets
```

Assets

Physical facilities or geographic areas tracked by Climate TRACE.

```
assets = client.get_climatetrace_assets(country_code="NPL", limit=5)
for a in assets["results"][:3]:
    print(f"  {a['name']} ({a['sector_name']})")
```

Parameters:

Parameter	Type	Description
`sector_id`	int	Filter by sector
`country_code`	str	ISO 3-letter country code
`location_bbox`	list[float]	Bounding box [min_lon, min_lat, max_lon, max_lat]

Emissions

Individual emission records tied to assets, sectors, and time periods.

```
emissions = client.get_climatetrace_emissions(country_code="NPL", limit=2)
for e in emissions["results"][:2]:
    print(f"  {e['asset_name']} | {e['sector_name']} | "
          f"{e['start_time'][:7]} | {float(e['co2e_tonnes']):,.1f} t CO2e")
```

Output:

```
  Tilottama Urban Area | agriculture | 2025-10 | 18.5 t CO2e
  Tulsipur Urban Area | agriculture | 2025-10 | 6.3 t CO2e
```

Parameters:

Parameter	Type	Description
`asset_id`	int	Filter by asset
`sector_id`	int	Filter by sector
`sector_name`	str	Filter by sector name string
`country_code`	str	ISO 3-letter country code
`gas`	str	Gas type: "co2e", "co2", "ch4", "n2o"
`date_from`	str/datetime	Start date
`date_to`	str/datetime	End date

Emission aggregates

```
# Total emissions for Nepal
totals = client.get_climatetrace_emissions_totals(country_code="NPL")
print(f"Nepal total CO2e: {totals['total_co2e']:,.0f} tonnes")
print(f"Records: {totals['record_count']:,}")
# Nepal total CO2e: 143,283,329 tonnes
# Records: 17,512

# Emissions by sector for Nepal
sector_totals = client.get_climatetrace_emissions_sector_totals(country_code="NPL")
```

```
for s in sector_totals:
    print(f"  {s['sector_name']:20s} {s['total_co2e']:>15,.0f} t CO2e")
```

Output:

```
  agriculture            44,173,091 t CO2e
  buildings              40,406,447 t CO2e
  manufacturing          31,488,322 t CO2e
  transportation         23,674,169 t CO2e
  waste                   3,541,300 t CO2e
```

Emissions by country (top emitters)

Aggregate emissions grouped by country. Returns all countries ranked by total CO2e.

```
country_totals = client.get_climatetrace_emissions_country_totals()
for c in country_totals[:5]:
    print(f"  {c['country_iso3']}: {c['total_co2e']:>18,.0f} t CO2e  "
          f"({c['record_count']:,} records)")
```

Output:

```
  CHN:  55,427,299,074 t CO2e  (1,887,391 records)
  USA:  30,617,951,988 t CO2e  (9,695,595 records)
  IND:  14,558,323,895 t CO2e  (1,695,797 records)
  RUS:   8,394,756,507 t CO2e  (2,068,570 records)
  JPN:   6,756,139,865 t CO2e  (1,124,859 records)
```

Parameters:

Parameter	Type	Description
`gas`	str	Filter by gas type (default: all)
`date_from`	str/datetime	Start date
`date_to`	str/datetime	End date

Gas type distribution

Breakdown of emission records by greenhouse gas type.

```
gas_dist = client.get_climatetrace_emissions_gas_type_distribution(country_code="NPL")
print(f"Total records: {gas_dist['total_records']:,}")
for g in gas_dist["gas_types"]:
    print(f"  {g['gas']:15s} {g['record_count']:>12,} records  ({g['percentage']:.1f}%)")
```

Output:

```
Total records: 53,422,192
  co2e             53,422,191 records  (100.0%)
  co2e_100yr                1 records  (0.0%)
```

> **Note:** Nearly all Climate TRACE records report CO2 equivalent (co2e). The `co2e_100yr` variant uses 100-year global warming potential. Individual gas breakdowns (CO2, CH4, N2O) are available in the per-emission records, not in this distribution endpoint.

Parameters:

Parameter	Type	Description
`country_code`	str	ISO 3-letter country code (optional — omit for global)

Emissions date range

Get the earliest and latest emission timestamps, useful for determining data coverage before running larger queries.

```
date_range = client.get_climatetrace_emissions_date_range(country_code="NPL")
print(f"From: {date_range['earliest_date']}")
print(f"To:   {date_range['latest_date']}")
print(f"Total: {date_range['total_records']:,} records")
```

Output:

```
From: 2021-01-01T00:00:00+00:00
To:   2025-10-01T00:00:00+00:00
Total: 17,512 records
```

Parameters:

Parameter	Type	Description
`country_code`	str	ISO 3-letter code (optional)
`sector_name`	str	Filter by sector name (optional)
`gas`	str	Filter by gas type (optional)

Company matches

Links between Climate TRACE emission-producing assets and GLEIF legal entities (companies/organizations). Each match includes the asset details and, when linked, the GLEIF LEI and legal name.

```
# List all matches (paginated)
matches = client.get_climatetrace_company_matches(limit=5)
print(f"Total company-asset links: {matches['count']}")
for m in matches["results"]:
    print(f"  Asset {m['asset_id']} ({m['asset_name']})"
```

```
          f" → LEI {m.get('legal_entity_lei', 'N/A')}")

# Filter by GLEIF Legal Entity Identifier
lei_matches = client.get_climatetrace_company_matches(
    legal_entity_lei="HWUPKR0MPOU8FGXBT394", limit=10,
)

# Full-text search across asset name, owner name, legal entity name
search_matches = client.get_climatetrace_company_matches(
    search="Alphabet", ordering="-matching_confidence",
)
```

> **Note:** Company match data is populated via the `match_companies_to_lei` management command, which resolves Climate TRACE owner/parent company names to GLEIF legal entities. An empty result does not indicate an error.

Parameters:

Parameter	Type	Description
`legal_entity_lei`	str	Filter by GLEIF LEI (20-char identifier) (optional)
`company_id`	str	Filter by internal company identifier (optional)
`matching_method`	str	Filter by match method, e.g. `'exact_name'`, `'fuzzy_name'` (optional)
`relationship_type`	str	Filter by relationship, e.g. `'owner'`, `'operator'` (optional)
`verified`	bool	Filter by verification status (optional)
`search`	str	Full-text search across asset name, owner name, legal entity name (optional)
`ordering`	str	Sort field: `'matching_confidence'`, `'-matching_confidence'`, `'created_at'`, `'-created_at'` (optional)
`limit`	int	Results per page
`offset`	int	Pagination offset

Violations

Regulatory emission violations associated with Climate TRACE assets.

```
violations = client.get_climatetrace_violations(limit=5)
print(f"Total violations: {violations['count']}")
for v in violations["results"]:
    print(f"  Asset {v['asset_id']}: {v['violation_type']} — {v['description']}")
```

Output:

```
Total violations: 0
```

> **Note:** Violation data is populated as regulatory compliance data is integrated. An empty result means no violations have been ingested yet.

Parameters:

Parameter	Type	Description
`asset_id`	int	Filter by specific asset (optional)
`limit`	int	Results per page
`offset`	int	Pagination offset

EDGAR — Emissions Inventories

EDGAR (Emissions Database for Global Atmospheric Research) is produced by the European Commission's Joint Research Centre (JRC). It provides comprehensive, peer-reviewed emission inventories covering all countries, all greenhouse gases, and all anthropogenic sectors, with data from 1970 to the present.

The Jana database contains **612 million emission records** including 145 million grid cells at 0.1-degree resolution (approximately 11 km at the equator).

Key concepts:

- **Country totals** — National emission totals aggregated by year, gas, and sector. Derived from bottom-up activity data (energy statistics, industrial output, agricultural data).
- **Grid emissions** — Spatially distributed emissions on a 0.1-degree latitude/longitude grid. Each cell contains the emission value for a specific gas, sector, and year. Useful for mapping emission hotspots.
- **Temporal profiles** — Monthly and hourly distribution factors that describe how annual emissions vary over the course of a year or day. For example, heating emissions peak in winter months.
- **Fast-track data** — Preliminary emission estimates published ahead of the full EDGAR release cycle. Fast-track data uses simplified methodologies and early activity data to provide more timely (but less precise) estimates, typically covering the most recent 1-2 years. It is gradually replaced by final data as the full inventory is completed.
- **Source version** — EDGAR releases versioned datasets (e.g., `EDGAR_2024_GHG`). Newer versions may revise historical estimates as methodologies improve.

Country totals

National emission totals by year, gas type, and sector.

```
nepal = client.get_edgar_country_totals(country_code="NPL", gas="CO2")
for row in nepal["results"][:5]:
    print(f"  {row['year']}: {float(row['value']):>10,.0f} kt CO2  "
          f"(sector: {row['sector']}, version: {row['source_version']})")
```

Output:

```
  2024:     15,892 kt CO2  (sector: India +, version: EDGAR_2024_GHG)
  2023:     15,432 kt CO2  (sector: India +, version: EDGAR_2024_GHG)
  2022:     15,045 kt CO2  (sector: India +, version: EDGAR_2024_GHG)
  2021:     17,383 kt CO2  (sector: India +, version: EDGAR_2024_GHG)
  2020:     16,456 kt CO2  (sector: India +, version: EDGAR_2024_GHG)
```

Parameters:

Parameter	Type	Description
`country_code`	str	ISO 3-letter code (e.g. "NPL", "USA")
`year`	int	Filter by year
`gas`	str	Gas type: "CO2", "CH4", "N2O", "GWP_100_AR5_GHG", "F-gases"
`sector`	str	Sector filter
`provisional`	bool	Include provisional data

Grid emissions

0.1-degree latitude/longitude gridded emissions. Supports spatial queries.

```
# Get grid emissions around Kathmandu (bbox: 84E-86E, 27N-29N)
grid = client.get_edgar_grid_emissions(
    bbox="84,27,86,29",
    gas="CO2",
    year=2022,
    limit=5,
)
for cell in grid["results"][:3]:
    print(f"  ({cell['lat']}, {cell['lon']}): "
          f"{float(cell['value']):.4f} kt CO2 [{cell['sector']}]")
```

Output:

```
  (28.350000, 85.850000): 0.0000 kt CO2 [total]
  (28.350000, 85.650000): 0.0000 kt CO2 [total]
  (28.350000, 85.550000): 0.0000 kt CO2 [total]
```

Parameters:

Parameter	Type	Description
`year`	int	Filter by year
`gas`	str	Gas type
`sector`	str	Sector filter
`min_value`	float	Minimum emission value
`bbox`	str	Bounding box "min_lon,min_lat,max_lon,max_lat"

Parameter	Type	Description
`coordinates`	str	Point "lon,lat" for radius search
`radius`	float	Radius in meters (used with coordinates)

Pagination note: Grid emissions use cursor-based pagination (not page numbers). The client's `fetch_all_pages()` handles this automatically.

Temporal profiles

Monthly and hourly emission distribution factors by sector.

```
profiles = client.get_edgar_temporal_profiles(sector="Waste Water Treatment", limit=3)
for p in profiles["results"][:3]:
    print(f"  Month {p['month']}: factor={p['factor']} ({p['temporal_level']})")
```

Output:

```
  Month 12: factor=0.08333333 (monthly)
  Month 12: factor=0.08333333 (monthly)
  Month 12: factor=0.08333333 (monthly)
```

Fast-track data

EDGAR Fast-track provides preliminary emission estimates ahead of the full EDGAR release. These are early estimates using simplified methodologies — useful for near-real-time analysis but less precise than the final country totals. Fast-track data is gradually replaced by final data as the full inventory is published.

The method signature and parameters are identical to `get_edgar_country_totals()`.

```
fasttrack = client.get_edgar_fasttrack(country_code="NPL", gas="CO2", limit=5)
print(f"Results: {len(fasttrack['results'])}")
for row in fasttrack["results"][:3]:
    print(f"  {row['year']}: {float(row['value']):,.0f} kt CO2 (sector: {row['sector']})")
```

Output:

```
Results: 0
```

> **Note:** Fast-track data availability depends on the EDGAR release cycle. When no fast-track data is available for a country/gas combination, the results list is empty. Use `get_edgar_country_totals()` for the complete, finalized inventory.

Parameters:

Parameter	Type	Description
`country_code`	str	ISO 3-letter code

Parameter	Type	Description
`year`	int	Filter by year
`gas`	str	Gas type: "CO2", "CH4", "N2O", etc.
`sector`	str	Sector filter
`provisional`	bool	Include provisional data

GLEIF — Legal Entity Identifiers

The **Global Legal Entity Identifier Foundation (GLEIF)** manages the Legal Entity Identifier (LEI) system — a standardized 20-character alphanumeric code (ISO 17442) assigned to legal entities in financial transactions. The Jana API provides access to 3.25M+ entities, 637K+ ownership relationships, and 5.8M+ reporting exceptions.

What You Can Do

- **Search** for legal entities by name, LEI, BIC code, or registry ID
- **Browse** corporate ownership hierarchies (parents, children, subsidiaries)
- **Filter** by country, entity status, category, jurisdiction
- **Look up** why an entity cannot report a parent (reporting exceptions)
- **Cross-reference** with Climate TRACE assets via company-asset matches

Entity Search and Lookup

```
# Search entities by name
entities = client.get_gleif_entities(search="Apple Inc", limit=10)
for e in entities['results']:
    print(f"{e['lei']}: {e['legal_name']} ({e['legal_address_country']})")

# Filter by country and category
us_funds = client.get_gleif_entities(
    legal_address_country="US",
    entity_category="FUND",
    entity_status="ACTIVE",
    limit=50,
)
print(f"US active funds: {us_funds['count']}")

# Get full detail for a single entity by LEI
apple = client.get_gleif_entity("HWUPKR0MPOU8FGXBT394")
print(f"Name: {apple['legal_name']}")
print(f"City: {apple['legal_address_city']}")
print(f"Jurisdiction: {apple['jurisdiction']}")
print(f"Status: {apple['entity_status']}")
```

List response fields (compact): `id`, `lei`, `legal_name`, `legal_address_country`, `headquarters_country`, `jurisdiction`, `entity_category`, `entity_status`,

`registration_status`

Detail response fields (full): All list fields plus `legal_name_language`, `other_names`, `legal_address_lines`, `legal_address_city`, `legal_address_region`, `legal_address_postal_code`, `headquarters_city`, `entity_sub_category`, `legal_form_code`, `legal_form_other`, `entity_creation_date`, `initial_registration_date`, `last_update_date`, `next_renewal_date`, `managing_lou`, `bic`, `registered_as`, `successor_lei`, `ingested_at`, `updated_at`

Corporate Hierarchy

```
# Get parent entities (direct and ultimate)
parents = client.get_gleif_entity_parents("HWUPKR0MPOU8FGXBT394")
if parents['direct_parent']:
    print(f"Direct parent: {parents['direct_parent']['legal_name']}")
if parents['ultimate_parent']:
    print(f"Ultimate parent: {parents['ultimate_parent']['legal_name']}")

# Get child entities (subsidiaries)
children = client.get_gleif_entity_children("HWUPKR0MPOU8FGXBT394")
print(f"Subsidiaries: {len(children)}")
for child in children:
    print(f"  {child['lei']}: {child['legal_name']} ({child['legal_address_country']})")
```

The `parents` response contains `direct_parent` and `ultimate_parent` keys. Both are `null` for top-level entities (e.g., Apple Inc. has no parent).

Relationships

```
# All relationships where an entity is the child
rels = client.get_gleif_relationships(
    start_lei="HWUPKR0MPOU8FGXBT394",
    relationship_type="IS_DIRECTLY_CONSOLIDATED_BY",
)
for r in rels['results']:
    print(f"{r['start_lei']} -> {r['end_lei']} ({r['relationship_type']})")
```

Relationship types: `IS_DIRECTLY_CONSOLIDATED_BY`, `IS_ULTIMATELY_CONSOLIDATED_BY`, `IS_FEEDER_TO`, `IS_FUND-MANAGED_BY`

Parameters:

Parameter	Type	Description
`start_lei`	str	Filter by child entity LEI
`end_lei`	str	Filter by parent entity LEI
`relationship_type`	str	Relationship type filter
`relationship_status`	str	ACTIVE or INACTIVE

Reporting Exceptions

```
# Why can an entity not report a parent?
exceptions = client.get_gleif_exceptions(lei="001GPB6A9XPE8XJICC14")
for exc in exceptions['results']:
    print(f"{exc['exception_category']}: {exc['exception_reason']}")
```

Exception categories: `DIRECT_ACCOUNTING_CONSOLIDATION_PARENT`, `ULTIMATE_ACCOUNTING_CONSOLIDATION_PARENT`

Exception reasons: `NON_CONSOLIDATING`, `NO_KNOWN_PERSON`, `NO_LEI`, `NATURAL_PERSONS`, `NON_PUBLIC`, `BINDING_LEGAL_OBSTACLES`, `CONSENT_NOT_OBTAINED`, `DETRIMENT_NOT_EXCLUDED`, `DISCLOSURE_DETRIMENTAL`

Cross-Reference with Climate TRACE

GLEIF entities can be linked to Climate TRACE emission-producing assets via the company-asset match table:

```
# Find an entity
entities = client.get_gleif_entities(search="Alphabet Inc", limit=1)
lei = entities['results'][0]['lei']

# Get subsidiaries
children = client.get_gleif_entity_children(lei)

# Check each subsidiary for emission-producing assets
for child in children:
    matches = client.get_climatetrace_company_matches(legal_entity_lei=child['lei'])
    if matches.get('count', 0) > 0:
        print(f"{child['legal_name']}: {matches['count']} emission sources")
```

GCP — Global Carbon Project

The Global Carbon Project quantifies global carbon emissions and their sources, sinks, and distribution. The Jana API provides access to four GCP datasets.

National Emissions

National territorial and consumption-based CO2 emissions by country and year, including per-capita values. The dataset covers most countries from 1750 to 2023, with 60,000+ records.

```
# USA CO2 emissions for 2020
usa = client.get_gcp_national_emissions(country_code="USA", year=2020)
for r in usa['results']:
    print(f"{r['country_name']} ({r['year']}): {r['territorial_mtco2']:.1f} MtCO2")
```

```
# Top emitters across all years
all_2020 = client.get_gcp_national_emissions(year=2020, limit=10)
```

Parameter	Type	Description
`country_code`	str	ISO-3 country code (e.g. `USA`, `CHN`) (optional)
`year`	int	Emission year (e.g. `2020`) (optional)
`budget_version`	str	GCP budget version (e.g. `2024`) (optional)
`limit`	int	Results per page (optional)
`offset`	int	Pagination offset (optional)

Response fields: `country_code`, `country_name`, `year`, `territorial_mtco2`, `consumption_mtco2`, `transfer_mtco2`, `per_capita_tco2`, `data_source_name`, `budget_version`

Emissions by Fuel

Global fossil CO2 emissions by fuel type — coal, oil, gas, cement, flaring, and other sources.

```
fuel_2020 = client.get_gcp_emissions_by_fuel(year=2020)
for r in fuel_2020['results']:
    print(f"{r['year']}: coal={r['coal_gtco2']:.2f}, oil={r['oil_gtco2']:.2f}, gas={r['gas
```

Parameter	Type	Description
`year`	int	Emission year (optional)
`budget_version`	str	GCP budget version (optional)
`limit`	int	Results per page (optional)

Response fields: `year`, `coal_gtco2`, `oil_gtco2`, `gas_gtco2`, `cement_gtco2`, `flaring_gtco2`, `other_gtco2`, `total_gtco2`, `budget_version`

Carbon Budget

Global carbon budget components: fossil emissions, land-use change, atmospheric growth, ocean sink, land sink.

```
budget = client.get_gcp_carbon_budget(year=2020)
```

> **Note**: Carbon budget data has not yet been ingested. This endpoint is available but currently returns empty results.

Methane Budget

Global methane budget components: anthropogenic and natural sources and sinks.

```
methane = client.get_gcp_methane_budget(year=2020)
```

> **Note**: Methane budget data has not yet been ingested. This endpoint is available but currently returns empty results.

NOAA — Storm Events

The NOAA Storm Events Database documents significant weather phenomena in the United States. The Jana API covers events from 2015 to 2024.

Storm Events

Query severe weather events with standard filters and spatial queries.

```
# Texas tornadoes in January 2023
storms = client.get_noaa_storm_events(
    event_type="Tornado", state="TEXAS", year=2023, month=1, limit=100
)
for s in storms['results']:
    print(f"{s['begin_date']}: {s['event_type']} in {s['cz_name']}, {s['state']}")
    print(f"  Injuries: {s['total_injuries']}, Deaths: {s['total_deaths']}")
    print(f"  Property damage: ${s['damage_property']}")

# Spatial query: events near Houston (50km radius)
houston = client.get_noaa_storm_events(
    lat=29.76, lon=-95.37, radius_km=50.0, event_type="Tornado", year=2023
)

# Bounding box query across central Texas
bbox_storms = client.get_noaa_storm_events(
    bbox="29.0,-99.0,33.0,-95.0", year=2023, limit=500
)
```

Parameter	Type	Description
`event_type`	str	Event type: `Tornado`, `Hurricane`, `Flood`, `Flash Flood`, `Hail`, `Thunderstorm Wind`, `Winter Storm`, `Blizzard`, `Ice Storm`, `Drought`, `Wildfire`, `Tropical Storm`, `Storm Surge/Tide` (optional)
`state`	str	US state in uppercase (e.g. `TEXAS`, `FLORIDA`) (optional)
`year`	int	Event year, 2015-2024 (optional)
`month`	int	Event month, 1-12 (optional)
`bbox`	str	Bounding box as `min_lat,min_lon,max_lat,max_lon` (optional)
`lat`	float	Latitude for radius search (optional, requires `lon` and `radius_km`)
`lon`	float	Longitude for radius search (optional, requires `lat` and `radius_km`)
`radius_km`	float	Search radius in km (optional, requires `lat` and `lon`)
`limit`	int	Results per page (optional)
`offset`	int	Pagination offset (optional)

Response fields: `event_id`, `episode_id`, `event_type`, `state`, `state_fips`, `cz_type`, `cz_fips`, `cz_name`, `begin_date`, `end_date`, `injuries_direct`, `injuries_indirect`, `deaths_direct`, `deaths_indirect`, `total_injuries`, `total_deaths`, `damage_property`, `damage_crops`, `source`, `magnitude`, `magnitude_type`, `tor_f_scale`, `year`, `month`, `latitude`, `longitude`, `geometry`

Unified ESG Analytics Layer

The ESG (Environmental, Social, Governance) analytics layer sits on top of all six data sources (OpenAQ, Climate TRACE, EDGAR, GLEIF, GCP, NOAA) and provides:

- **Unified data access** — Query across sources with a single method, getting normalized results with consistent field names.
- **Cross-source correlations** — Find spatial relationships between air quality monitoring stations and emission sources.
- **Temporal aggregations** — Pre-computed monthly/daily rollups for faster analysis.
- **Quality metrics** — Data freshness, completeness, and reliability scores per table and source.
- **System health** — Monitoring of ingestion pipelines, database health, and API performance.
- **Definitions** — Standardized parameter names, units, and source metadata.

Health check

```
health = client.get_health()
print(health)
# {'service': 'esg', 'status': 'healthy', 'version': 'v1', 'timestamp': None}
```

Platform summary

```
summary = client.get_summary()
overview = summary["platform_overview"]
print(f"Total records: {overview['total_records']:,}")
print(f"Data sources: {', '.join(overview['data_sources'])}")
print(f"OpenAQ measurements: {summary['openaq']['measurements']:,}")
print(f"Climate TRACE emissions: {summary['climatetrace']['emissions']:,}")
print(f"EDGAR total emissions: {summary['edgar']['emissions']:,}")
```

Output:

```
Total records: 2,157,471,050
Data sources: openaq, climatetrace, edgar
OpenAQ measurements: 1,489,718,016
Climate TRACE emissions: 53,495,196
EDGAR total emissions: 612,557,803
```

Unified sectors

Cross-source sector view (requires country filter).

```
sectors = client.get_sectors(country_codes=["NPL"])
for s in sectors["sectors"][:5]:
    print(f"  {s['display_name']:20s} source={s['source']:15s} "
          f"records={s['emission_count']:,}")
```

Output:

```
  Agriculture          source=climatetrace  records=8,940
  Transportation       source=climatetrace  records=3,306
  Buildings            source=climatetrace  records=2,900
  Manufacturing        source=climatetrace  records=1,450
  Waste                source=climatetrace  records=916
```

System health (detailed)

Comprehensive system health including data freshness, ingestion status, and per-table details. More detailed than `get_health()`.

```
sys_health = client.get_system_health()
ss = sys_health["system_status"]
print(f"Overall: {ss['overall_health']}, Status: {ss['status']}")

df = sys_health["data_freshness"]
print(f"\nStale tables ({len(df['stale_tables'])}):")
for t in df["table_details"][:5]:
    print(f"  {t['table']:35s} {t['record_count']:>14,} records  "
          f"latest: {t['latest_record'][:10] if t['latest_record'] else 'N/A'}")
```

Output:

```
Overall: critical, Status: degraded

Stale tables (14):
  openaq_locations                           50,514 records  latest: 2026-03-06
  openaq_measurements                 1,489,718,052 records  latest: 2026-01-01
  openaq_sensors                            320,783 records  latest: 2025-10-31
  climatetrace_assets                     1,337,615 records  latest: 2026-01-25
  climatetrace_emissions                 53,422,192 records  latest: 2025-10-01
```

> **Note:** "critical" or "degraded" status typically reflects stale data (ingestion hasn't run recently) rather than a system outage. The API is still fully functional for querying existing data.

Definitions

Reference metadata for parameters, units, and data sources used across the platform.

```
defs = client.get_definitions()
print(defs)
# {'categories': ['units', 'sources'],
#  'endpoints': {'units': '/api/v1/esg/definitions/units/',
#                'sources': '/api/v1/esg/definitions/sources/'}}
```

Unit definitions

```
units = client.get_unit_definitions()
for u in units["units"][:5]:
    print(f"  {u['unit']:15s} ({u['parameter_type']:15s}) {u['description'][:60]}")
```

Output:

```
  ug/m3           (air_quality    ) Micrograms per cubic meter — mass concentration of pol
  ppm             (air_quality    ) Parts per million — volume concentration of gas in air
  ppb             (air_quality    ) Parts per billion — volume concentration of gas in air
  tonnes          (emissions      ) Metric tonnes (1000 kg) — total mass of emissions
  tonnes_co2e     (emissions      ) Tonnes of CO2 equivalent — emissions normalized to CO2
```

Source definitions

```
sources = client.get_source_definitions()
for s in sources["sources"]:
    print(f"  {s['id']:15s} {s['name']:20s} {s['description'][:70]}")
```

Output:

```
  openaq          OpenAQ               Real-time and historical air quality measurements f
  climatetrace    Climate TRACE        Facility-level greenhouse gas emissions estimates u
  edgar           EDGAR                JRC/EU comprehensive global emissions inventory for
```

Parameter definitions

```
parameters = client.get_parameter_definitions()
```

Unified data access

Query across data sources with a single call. Returns normalized locations and measurements with consistent field names regardless of source.

```
data = client.get_data(
    sources=["openaq"],
    country_codes=["NP"],
    parameters=["pm25"],
```

```
    date_from="2025-01-01",
    date_to="2025-12-31",
    limit=3,
)
print(f"Locations: {data['total_locations']}, Measurements: {data['total_measurements']}")
for loc in data["locations"][:2]:
    print(f"  {loc['name']} ({loc['country_code']}) @ {loc['coordinates']}")
for m in data["measurements"][:3]:
    print(f"  {m['parameter']}={m['value']} {m['unit']} at {m['timestamp'][:16]}")
```

Output:

```
Locations: 3, Measurements: 3
  Embassy Kathmandu (NP) @ [85.336206, 27.738703]
  Phora Durbar Kathman (NP) @ [85.315703, 27.712464]
  pm25=25.0 ug/m3 at 2025-12-22T09:15
  pm25=24.0 ug/m3 at 2025-12-22T08:15
  pm25=48.0 ug/m3 at 2025-12-22T07:15
```

Key parameters:

Parameter	Type	Description
`sources`	list[str]	Data sources: "openaq", "climatetrace", "edgar"
`country_codes`	list[str]	ISO country codes (2-letter for OpenAQ, 3-letter for CT/EDGAR)
`parameters`	list[str]	Parameter names: "pm25", "CO2", etc.
`date_from` / `date_to`	str/datetime	Date range (ISO 8601)
`location_bbox`	list[float]	Bounding box [min_lon, min_lat, max_lon, max_lat]
`location_point`	list[float]	Center point [lon, lat] for radius search
`radius_km`	float	Search radius (with location_point)
`temporal_resolution`	str	Aggregation: "hourly", "daily", "monthly"
`quality_threshold`	int	Minimum quality score (0-100)
`correlation_analysis`	bool	Include cross-source correlation
`trend_analysis`	bool	Include trend detection
`anomaly_detection`	bool	Flag anomalous values
`statistical_tests`	bool	Include statistical significance tests
`limit` / `offset`	int	Pagination

Correlations

Cross-source spatial correlation analysis. Finds relationships between air quality monitoring stations (OpenAQ) and emission sources (Climate TRACE) by geographic proximity.

```
corr = client.get_correlations(
    sources=["openaq", "climatetrace"],
    country_codes=["NPL"],
    statistical_tests=True,
)
print(f"Method: {corr['correlations']['method']}")
summary = corr["correlations"]["summary"]
print(f"Assets analyzed: {summary['total_assets_analyzed']}")
print(f"Assets with nearby monitoring: {summary['assets_with_nearby_monitoring']}")
print(f"Coverage: {summary['coverage_percentage']}%")
print(f"Correlations found: {corr['correlations']['total_correlations']}")
```

Output:

```
Method: country_spatial
Assets analyzed: 213
Assets with nearby monitoring: 0
Coverage: 0.0%
Correlations found: 0
```

> **Note:** Zero correlations means no Climate TRACE assets in Nepal have an OpenAQ monitoring station within the default spatial radius (50 km). Try increasing `spatial_radius_km` or querying a country with denser monitoring networks (e.g., USA, UK).

Parameters:

Parameter	Type	Description
`sources`	list[str]	At least 2 sources to correlate
`country_codes`	list[str]	Country filter
`parameters`	list[str]	Parameter filter
`correlation_type`	str	Type of correlation analysis
`spatial_radius_km`	float	Max distance for spatial pairing (default: 50)
`temporal_window_days`	int	Time window for temporal matching
`minimum_data_points`	int	Minimum observations required
`statistical_tests`	bool	Include p-values and significance
`sector_filter`	list[str]	Filter by emission sector

Aggregations

Pre-computed temporal aggregations for faster analysis on large datasets. Reduces millions of raw records to monthly/daily summaries.

```
agg = client.get_aggregations(
    temporal_resolution="monthly",
    sources=["climatetrace"],
    country_codes=["NPL"],
    date_from="2024-01-01",
    date_to="2024-12-31",
)
print(f"Resolution: {agg['temporal_resolution']}")
for bucket in agg["data"]["climatetrace"][:3]:
    print(f"  {bucket['time_period'][:7]}: "
          f"CO2e={bucket['total_co2e']:>12,.0f} t  "
          f"assets={bucket['unique_assets']}  "
          f"records={bucket['emission_count']}")
```

Output:

```
Resolution: monthly
  2024-01: CO2e=   2,610,796 t  assets=301  records=301
  2024-02: CO2e=   2,315,728 t  assets=301  records=301
  2024-03: CO2e=   2,431,003 t  assets=301  records=301
```

Parameters:

Parameter	Type	Description
`temporal_resolution`	str	**Required.** "hourly", "daily", or "monthly"
`sources`	list[str]	Data source filter
`country_codes`	list[str]	Country filter
`location_bbox`	list[float]	Bounding box
`date_from` / `date_to`	str/datetime	Date range
`quality_threshold`	int	Minimum quality score
`parameters`	list[str]	Parameter filter
`include_correlations`	bool	Include cross-source correlation

Trends

Temporal trend analysis and forecasting. Detects long-term trends, seasonal patterns, and anomalies in environmental data.

```
trends = client.get_trends(
    sources=["edgar"],
    parameters=["CO2"],
    country_codes=["NPL"],
    date_from="2015-01-01",
    date_to="2024-12-31",
```

```
    temporal_resolution="yearly",
)
```

> **Note:** Trend analysis on large datasets (e.g., global EDGAR without country filter) may time out. Always apply `country_codes` and a date range to keep queries manageable.

Parameters:

Parameter	Type	Description
`sources`	list[str]	Data source filter
`parameters`	list[str]	Parameter names to analyze
`analysis_type`	str	"trend", "seasonal", "anomaly", or "forecast"
`temporal_resolution`	str	"daily", "monthly", "yearly"
`date_from` / `date_to`	str/datetime	Analysis window
`country_codes`	list[str]	Country filter (recommended for performance)
`forecast_days`	int	Number of days to forecast ahead
`confidence_level`	float	Confidence interval (e.g., 0.95)

Quality metrics

Data quality analysis across all tables — freshness, completeness, and reliability scores.

```
quality = client.get_quality(
    sources=["openaq"],
    parameters=["pm25"],
)
qi = quality["quality_insights"]
print(f"Overall quality score: {qi['overall_quality_score']}")
for table_name, tq in list(qi["table_quality"].items())[:5]:
    print(f"  {table_name:35s} score={tq['quality_score']:.2f}  "
          f"records={tq['total_records']:>14,}  issues={tq['issues']}")
```

Output:

```
Overall quality score: 0.704
  openaq_locations                    score=0.46  records=        50,514  issues=['stale_d
  openaq_measurements                 score=0.46  records= 1,489,718,052  issues=['stale_d
  openaq_sensors                      score=0.46  records=       320,783  issues=['stale_d
  openaq_parameters                   score=0.46  records=            30  issues=['stale_d
  openaq_datasets                     score=1.00  records=             0  issues=[]
```

> **Quality scores:** 1.0 = perfect (fresh, complete data). Scores below 0.5 typically indicate stale data (ingestion overdue). The score combines freshness (how recently data was updated) and completeness (null values, missing fields).

Parameters:

Parameter	Type	Description
`sources`	list[str]	Filter by source
`parameters`	list[str]	Filter by parameter
`location_bbox`	list[float]	Bounding box
`date_from` / `date_to`	str/datetime	Date range
`quality_detail_level`	str	"summary", "detailed", or "comprehensive"

Alerts

Quality-based intelligent alerts for data anomalies, freshness issues, and threshold violations.

```
alerts = client.get_alerts(
    severity=["high", "critical"],
    limit=10,
)
for a in alerts.get("results", []):
    print(f"  [{a['severity']}] {a['alert_type']}: {a['message']}")
```

Parameters:

Parameter	Type	Description
`alert_types`	list[str]	"quality", "correlation", "availability", "threshold", "system"
`severity`	list[str]	"low", "medium", "high", "critical"
`status`	str	"active", "resolved", "acknowledged"
`date_from`	str/datetime	Alerts since this date
`limit`	int	Results per page

GeoJSON export

Get mapping-ready GeoJSON data for visualization with folium, plotly, or leaflet.

```
geojson = client.get_geojson(
    sources=["openaq"],
    location_bbox=[83, 26, 88, 31],  # Nepal region
    parameters=["pm25"],
)
# Returns GeoJSON FeatureCollection
print(f"Type: {geojson['type']}")
print(f"Features: {len(geojson.get('features', []))}")
```

Parameters:

Parameter	Type	Description
`sources`	list[str]	Data sources to include
`location_bbox`	list[float]	Bounding box [min_lon, min_lat, max_lon, max_lat]
`parameters`	list[str]	Parameter filter
`temporal_resolution`	str	Aggregation level
`quality_threshold`	int	Minimum quality score
`geometry_simplification`	str	Simplification level for geometries
`include_quality_styling`	bool	Include quality-based GeoJSON styling
`layer_separation`	bool	Separate features by data source into layers

Locations (unified)

Unified location view across all data sources with consistent field names.

```
locs = client.get_locations(
    sources=["openaq"],
    country_codes=["NP"],
    include_metadata=True,
    limit=3,
)
print(f"Total locations: {locs['total_locations']}")
for loc in locs["locations"][:3]:
    print(f"  {loc['name']} ({loc['source']}) @ {loc['coordinates']} [{loc['country_code']
```

Output:

```
Total locations: 121
  Embassy Kathmandu (openaq) @ [85.336206, 27.738703] [NP]
  Phora Durbar Kathman (openaq) @ [85.315703, 27.712464] [NP]
  Dhathutole, Handigaun (openaq) @ [85.330135, 27.727502] [NP]
```

Parameters:

Parameter	Type	Description
`sources`	list[str]	Data sources
`country_codes`	list[str]	Country filter
`location_bbox`	list[float]	Bounding box
`location_point`	list[float]	Center point [lon, lat]
`radius_km`	float	Search radius
`admin_areas`	list[str]	Administrative area filter
`include_metadata`	bool	Include detailed metadata
`limit`	int	Results per page

Bulk exports

Create asynchronous bulk data exports in CSV, JSON, Parquet, or GeoJSON format.

```
# Create an export job
export = client.create_export(
    format="csv",
    query={"sources": ["edgar"], "country_codes": ["NPL"]},
    compression="gzip",
)
export_id = export["id"]
print(f"Export ID: {export_id}")

# Check status (poll until completed)
status = client.get_export_status(export_id)
print(f"Status: {status['status']}")  # "pending", "processing", "completed", "failed"

# Download when ready
if status["status"] == "completed":
    data = client.download_export(export_id)  # Returns bytes
    with open("nepal_edgar.csv.gz", "wb") as f:
        f.write(data)
```

Parameters (create_export):

Parameter	Type	Description
`format`	str	**Required.** "csv", "json", "parquet", "geojson"
`query`	dict	**Required.** Query parameters (sources, country_codes, date_from, etc.)
`compression`	str	"none", "gzip", "bzip2"
`chunk_size`	int	Records per chunk
`email_notification`	str	Email address for completion notification
`expires_in_hours`	int	Download link expiration

Pagination

The API uses two pagination styles. The client handles both transparently.

Page-number pagination (OpenAQ, some endpoints)

```
# Manual page traversal
page1 = client.get_openaq_locations(country_codes="NP", page=1, page_size=10)
print(f"Total: {page1['count']}, This page: {len(page1['results'])}")
print(f"Next: {page1['next']}")
# Total: 121, This page: 10
# Next: http://api-test.jana.earth/.../locations/?country_codes=NP&limit=10&page=2
```

Cursor-based pagination (EDGAR grid, Climate TRACE emissions)

```
# First page
page1 = client.get_edgar_grid_emissions(bbox="84,27,86,29", limit=100)
# page1["next"] contains a cursor URL — not a page number
```

fetch_all_pages() — automatic traversal

For bulk data retrieval, use `fetch_all_pages()` which follows all pages automatically:

```
# Get ALL Nepal Climate TRACE emissions (follows cursor pagination)
all_emissions = client.fetch_all_pages(
    "/api/v1/data-sources/climatetrace/emissions/",
    params={"country_code": "NPL"},
    page_size=5000,
    max_pages=100,
    progress=True,       # Print progress during fetch
    progress_every=5,    # Print every 5 pages
)
print(f"Fetched {len(all_emissions)} records")
```

The method auto-detects pagination type (cursor, page-number, or limit-offset) and follows `next` URLs until all data is retrieved.

Error Handling

The client raises specific exceptions for different error conditions:

```
from eko_client.exceptions import (
    EkoClientError,          # Base exception
    EkoAuthenticationError,  # 401 - bad credentials
    EkoSessionExpiredError,  # 401 - both tokens expired, re-login needed
    EkoAPIError,             # 4xx/5xx API errors
    EkoRateLimitError,       # 429 - rate limit exceeded
    EkoNotFoundError,        # 404 - resource not found
)

try:
    data = client.get_openaq_location(999999999)
except EkoNotFoundError:
    print("Location not found")
except EkoSessionExpiredError:
    # Both access and refresh tokens expired
    client.login_device()  # Re-authenticate
    data = client.get_openaq_location(999999999)
except EkoRateLimitError as e:
    print(f"Rate limited. Retry after {e.retry_after} seconds")
```

```
except EkoAPIError as e:
    print(f"API error {e.status_code}: {e.message}")
```

Exception hierarchy

```
EkoClientError
  +-- EkoAuthenticationError
  |     +-- EkoSessionExpiredError
  +-- EkoAPIError
        +-- EkoRateLimitError
        +-- EkoNotFoundError
```

All exceptions carry:

- `message` — human-readable description
- `status_code` — HTTP status code (if applicable)
- `response_data` — raw response dict (if available)

Recommended Python Libraries

These libraries pair well with the Eko client for data analysis workflows:

Data manipulation

Library	Install	Purpose
`pandas`	`pip install pandas`	DataFrames, time series, CSV/Excel I/O
`numpy`	`pip install numpy`	Numerical arrays, math operations
`geopandas`	`pip install geopandas`	Geospatial DataFrames with shapely geometries

Visualization

Library	Install	Purpose
`matplotlib`	`pip install matplotlib`	Static plots, charts, heatmaps
`seaborn`	`pip install seaborn`	Statistical visualization built on matplotlib
`plotly`	`pip install plotly`	Interactive charts and maps
`folium`	`pip install folium`	Interactive Leaflet.js maps

Geospatial

Library	Install	Purpose
`shapely`	`pip install shapely`	Geometric operations (with geopandas)
`contextily`	`pip install contextily`	Basemap tiles for matplotlib/geopandas plots

Jupyter

Library	Install	Purpose
`jupyterlab`	`pip install jupyterlab`	Modern Jupyter interface
`ipykernel`	`pip install ipykernel`	Kernel management for venvs
`ipywidgets`	`pip install ipywidgets`	Interactive widgets in notebooks
`tqdm`	`pip install tqdm`	Progress bars

Example: API data to pandas DataFrame

```
import pandas as pd
from eko_client import EkoUserClient

client = EkoUserClient(
    base_url="https://auth-dev.jana.earth",
    api_base_url="https://api-test.jana.earth",
)
client.login_device()

# Fetch Nepal CO2 emissions and convert to DataFrame
data = client.get_edgar_country_totals(country_code="NPL", gas="CO2")
df = pd.DataFrame(data["results"])
df["value"] = df["value"].astype(float)
df = df.sort_values("year")
print(df[["year", "value", "sector"]].tail())
#    year         value     sector
# 24 2020  16456.181761  India +
# 23 2021  17382.956052  India +
# 22 2022  15045.368602  India +
# 21 2023  15432.018193  India +
# 20 2024  15891.857297  India +
```

Example: Climate TRACE emissions on a map

```
import folium
from eko_client import EkoUserClient

client = EkoUserClient(
    base_url="https://auth-dev.jana.earth",
    api_base_url="https://api-test.jana.earth",
)
client.login_device()

# Fetch Nepal emissions with location data
emissions = client.get_climatetrace_emissions(country_code="NPL", limit=50)

# Get associated assets for coordinates
m = folium.Map(location=[28.0, 84.0], zoom_start=7)
for e in emissions["results"]:
```

```
    folium.CircleMarker(
        location=[27.7, 85.3],  # Use actual asset coordinates
        radius=3,
        popup=f"{e['asset_name']}: {e['co2e_tonnes']} t CO2e",
    ).add_to(m)
m.save("nepal_emissions.html")
```

Complete Method Reference

EkoUserClient

Authentication

Method	Description
`login_device()`	OAuth 2.0 Device Code flow (opens browser)
`login_password(email, password)`	Email + password login
`refresh_jwt()`	Refresh expired access token
`get_user_info()`	Get current user profile
`is_authenticated()`	Check if access token exists
`logout()`	Invalidate token
`close()`	Close HTTP connections

OpenAQ

Method	Parameters	Description
`get_openaq_locations(...)`	country_codes, location_bbox, page, page_size, limit, offset	List monitoring stations
`get_openaq_location(id)`	location_id	Get single station
`get_openaq_sensors(...)`	location_id, parameter, country_code, location_bbox, coordinates, radius_km, limit, offset	List sensors
`get_openaq_sensor(id)`	sensor_id	Get single sensor
`get_openaq_measurements(...)`	location_id, parameter, date_from, date_to, country_code, ordering,	List measurements

Method	Parameters	Description
	page, page_size, limit, offset	
`get_openaq_measurement(id)`	measurement_id	Get single measurement
`get_openaq_measurements_date_range()`	—	Min/max dates and total count
`get_openaq_measurements_totals()`	—	Global record count and stats
`get_openaq_measurements_parameter_totals()`	—	Counts grouped by parameter
`get_openaq_measurements_country_totals()`	—	Counts grouped by country
`get_openaq_parameters(...)`	limit, offset	List available parameters
`get_openaq_parameter(id)`	parameter_id	Get single parameter

Climate TRACE

Method	Parameters	Description
`get_climatetrace_sectors(...)`	limit, offset	List emission sectors
`get_climatetrace_countries(...)`	limit, offset	List countries with data
`get_climatetrace_assets(...)`	sector_id, country_code, location_bbox, limit, offset	List emission assets
`get_climatetrace_emissions(...)`	asset_id, sector_id, sector_name, country_code, gas, date_from, date_to, limit, offset	List emission records
`get_climatetrace_emissions_date_range(...)`	country_code, sector_name, gas	Min/max dates

Method	Parameters	Description
`get_climatetrace_emissions_totals(...)`	country_code, sector_name, gas, date_from, date_to	Aggregate totals
`get_climatetrace_emissions_sector_totals(...)`	country_code, gas, date_from, date_to	Totals by sector
`get_climatetrace_emissions_country_totals(...)`	gas, date_from, date_to	Totals by country
`get_climatetrace_emissions_gas_type_distribution(...)`	country_code	Gas type breakdown
`get_climatetrace_company_matches(...)`	legal_entity_lei, company_id, matching_method, relationship_type, verified, search, ordering, limit, offset	Company-asset links
`get_climatetrace_violations(...)`	asset_id, limit, offset	Regulatory violations

EDGAR

Method	Parameters	Description
`get_edgar_country_totals(...)`	country_code, year, gas, sector, provisional, limit, offset	National totals
`get_edgar_grid_emissions(...)`	year, gas, sector, min_value, bbox, coordinates, radius, limit, offset	0.1-degree gridded data
`get_edgar_temporal_profiles(...)`	sector, temporal_level, month, hour, limit, offset	Monthly/hourly profiles
`get_edgar_fasttrack(...)`	country_code, year, gas, sector, provisional, limit, offset	Fast-track data

GLEIF

Method	Parameters	Description
`get_gleif_entities(...)`	search, entity_status, registration_status, entity_category, legal_address_country,	Legal entity search and list

Method	Parameters	Description
	headquarters_country, jurisdiction, ordering, limit, offset	
`get_gleif_entity(lei)`	lei	Single entity detail by LEI
`get_gleif_entity_parents(lei)`	lei	Direct and ultimate parent entities
`get_gleif_entity_children(lei)`	lei	Direct child entities (subsidiaries)
`get_gleif_relationships(...)`	start_lei, end_lei, relationship_type, relationship_status, ordering, limit, offset	Entity ownership relationships
`get_gleif_exceptions(...)`	lei, exception_category, exception_reason, limit, offset	Reporting exceptions

GCP (Global Carbon Project)

Method	Parameters	Description
`get_gcp_national_emissions(...)`	country_code, year, budget_version, limit, offset	National CO2 emissions by country/year
`get_gcp_emissions_by_fuel(...)`	year, budget_version, limit, offset	Global fossil CO2 by fuel type
`get_gcp_carbon_budget(...)`	year, budget_version, limit, offset	Global carbon budget components
`get_gcp_methane_budget(...)`	year, budget_version, limit, offset	Global methane budget components

NOAA (Storm Events)

Method	Parameters	Description
`get_noaa_storm_events(...)`	event_type, state, year, month, bbox, lat, lon, radius_km, limit, offset	Severe weather events with spatial queries

Unified ESG

Method	Parameters	Description
`get_health()`	—	API health check

Method	Parameters	Description
`get_system_health()`	—	System health with table details
`get_summary()`	—	Platform overview and record counts
`get_sectors(...)`	sources, country_codes, limit, date_from, date_to	Cross-source sectors
`get_definitions()`	—	Definition categories
`get_parameter_definitions(...)`	sources, parameter_types	Parameter definitions
`get_unit_definitions()`	—	Unit definitions
`get_source_definitions()`	—	Source definitions
`get_data(...)`	sources, location_bbox, location_point, radius_km, country_codes, admin_areas, date_from, date_to, temporal_resolution, parameters, quality_threshold, include_flags, correlation_analysis, trend_analysis, anomaly_detection, statistical_tests, output_format, bbox_precision, limit, offset	Unified data access
`get_aggregations(...)`	temporal_resolution, sources, country_codes, location_bbox, date_from, date_to, quality_threshold, parameters, include_correlations	Pre-computed aggregations
`get_correlations(...)`	sources, country_codes, parameters, correlation_type, location_bbox, temporal_window_days, spatial_radius_km, minimum_data_points, statistical_tests, sector_filter	Cross-source correlations
`get_trends(...)`	sources, parameters, analysis_type, temporal_resolution, location_bbox, date_from, date_to, forecast_days, confidence_level	Trend analysis
`get_quality(...)`	sources, location_bbox, date_from, date_to, parameters, quality_detail_level	Quality metrics

Method	Parameters	Description
`get_alerts(...)`	alert_types, severity, status, date_from, limit	Quality alerts
`get_geojson(...)`	sources, location_bbox, parameters, temporal_resolution, quality_threshold, geometry_simplification, include_quality_styling, layer_separation	GeoJSON export
`get_locations(...)`	sources, location_bbox, location_point, radius_km, country_codes, admin_areas, include_metadata, limit	Unified locations
`create_export(...)`	format, query, compression, chunk_size, email_notification, expires_in_hours	Create bulk export
`get_export_status(export_id)`	export_id	Check export progress
`download_export(export_id)`	export_id	Download export file

Pagination helper

Method	Parameters	Description
`fetch_all_pages(endpoint, ...)`	endpoint, params, page_size, max_pages, progress, progress_every	Auto-paginate any endpoint