ApiLinker: A Universal Bridge for REST API Integrations

16 July 2025

# Abstract

Software-driven research increasingly relies on the integration of heterogeneous web services to acquire, harmonize, and exchange data. Differences in REST semantics, authentication schemes, pagination conventions, and nested data representations introduce substantial engineering overhead and threaten reproducibility when integrations are implemented ad hoc. ApiLinker is an open-source Python framework that systematizes these concerns by providing a configuration-first approach for authenticating to APIs, retrieving and transforming records, and transmitting results to target endpoints. The framework supports both one-off data migrations and long-running synchronisation processes, and is designed to operate as a library or via a command-line interface. Its architecture separates transport, mapping, authentication, scheduling, and extensibility concerns, and embeds resilience mechanisms (e.g., retries, circuit breakers, dead-letter queues) to sustain real-world network volatility. We describe the design and implementation of ApiLinker, present its quality assurance procedures and benchmarking harness, and discuss reuse across multiple research domains including bibliometrics, chemical informatics, and climate science. By elevating integrations to version-controlled configuration artefacts and providing deterministic, composable mappings with value-level transformers, ApiLinker enables reproducible, maintainable pipelines that align with FAIR and open science practices. The software, documentation, and tests are openly available under the MIT license, with an archived release to be minted via Zenodo.

Keywords: REST, research software, data integration, API mapping, reproducibility, FAIR, Python

# Overview

Modern computational research increasingly depends on integrating heterogeneous web services. Each service exposes idiosyncratic REST interfaces, authentication schemes, pagination models, and data representations, which collectively impose significant engineering overhead on researchers. While ad hoc client libraries abstract individual services, researchers typically resort to bespoke glue code to connect systems, undermining reproducibility and maintainability. ApiLinker is an open-source Python framework that systematizes this integration layer. It provides a declarative, configuration-first approach to authenticate, retrieve, transform, and transmit data between arbitrary REST endpoints with strong guarantees for robustness and auditability. The framework supports both one-off migrations and long-running synchronisation pipelines, and is designed to be embedded within scientific workflows as well as operated via a command-line interface.

# Introduction

The proliferation of microservices has increased inter-service communication complexity [@Fowler2014]. In practice, each API embodies local assumptions concerning authentication (API keys, bearer tokens, OAuth 2.0 variants), pagination (page/size, cursor-based, next-link), payload schemas (flat vs. deeply nested JSON), and rate limiting. Researchers repeatedly re-implement authentication, pagination, and structural transformations across domains—bioinformatics [@Ison2013], environmental science [@Vitolo2015], and computational social science [@Lomborg2014]—often with limited reuse potential and fragile error handling [@Verborgh2013]. These bespoke scripts resist review, are brittle under provider changes, and are difficult to archive or share in a manner that supports exact procedural reproducibility.

ApiLinker addresses this methodological gap by: (i) elevating integrations to versionable, declarative configurations; (ii) providing a principled mapping model for heterogeneously nested structures and value-level transformations; and (iii) embedding resilience patterns (retries, circuit breakers, dead-letter queues) to sustain long-running acquisition under real-world API volatility. This design enables rigorous, reproducible pipelines consistent with FAIR principles [@Wilkinson2016]. In addition to a Python application programming interface (API), a command-line interface (CLI) supports scripted execution in containerised and scheduled environments. The architecture emphasises deterministic behaviour, explicit configuration of side effects (e.g., rate limiting, retries), and composable extensions (connectors, transformers) to support domain specialisation without core modifications.

# Design goals and key capabilities

ApiLinker is built with a modular architecture that separates concerns into logical components:

## Security Features

APILinker provides security capabilities for handling sensitive data:

* **Secure Credential Storage**: Optional encrypted storage for API credentials
* **OAuth Support**: Implementation of modern OAuth flows including PKCE (for mobile/SPA) and Device Flow (for IoT/CLI)
* **Role-Based Access Control**: Fine-grained permissions for multi-user environments

## API Connectors

The connector module handles communication with REST APIs, supporting various HTTP methods, customizable endpoints, and automatic pagination. Each endpoint is configured with its path, method, headers, and optional parameters.

connector = ApiConnector(  
 connector\_type="rest",  
 base\_url="https://api.example.com/v1",  
 auth\_config=auth\_config,  
 endpoints={  
 "list\_users": {  
 "path": "/users",  
 "method": "GET",  
 "pagination": {"data\_path": "data", "next\_page\_path": "meta.next"}  
 }  
 }  
)

## Field Mapping

The mapper module translates data structures between source and target formats using a flexible mapping system. It supports nested field paths, conditional mappings, and data transformations.

mapper.add\_mapping(  
 source="list\_users",  
 target="create\_user",  
 fields=[  
 {"source": "id", "target": "external\_id"},  
 {"source": "profile.name", "target": "full\_name", "transform": "strip"}  
 ]  
)

## Authentication

Multiple authentication methods are supported, with secure environment variable handling:

auth:  
 type: oauth2\_client\_credentials  
 client\_id: ${CLIENT\_ID}  
 client\_secret: ${CLIENT\_SECRET}  
 token\_url: https://auth.example.com/token  
 scope: read write

## Scheduling

The scheduler component enables recurring synchronizations with interval or cron-based timing:

scheduler.add\_schedule(type="cron", expression="0 \*/6 \* \* \*") # Every 6 hours  
scheduler.start(sync\_function)

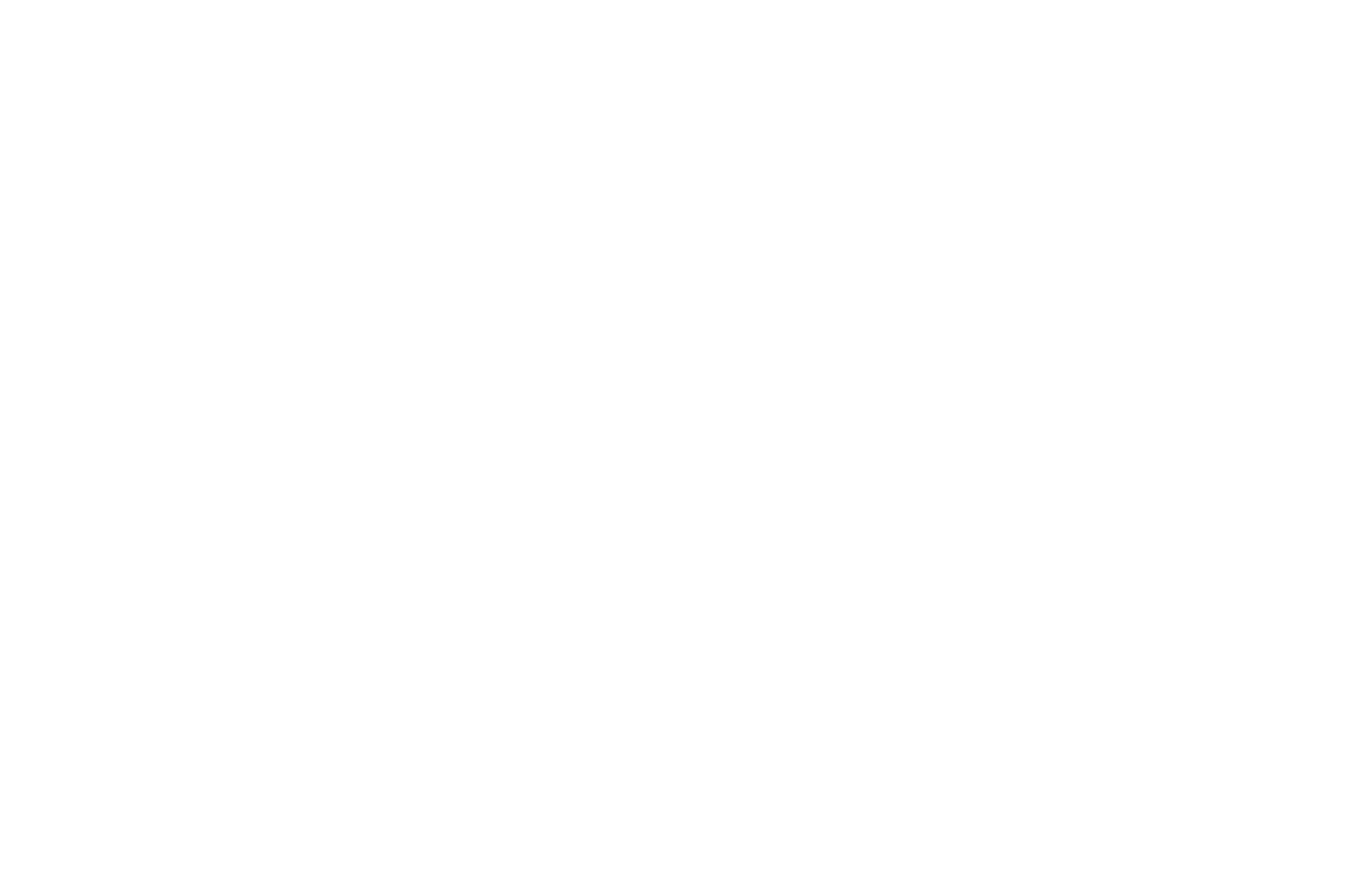
# Methods and results

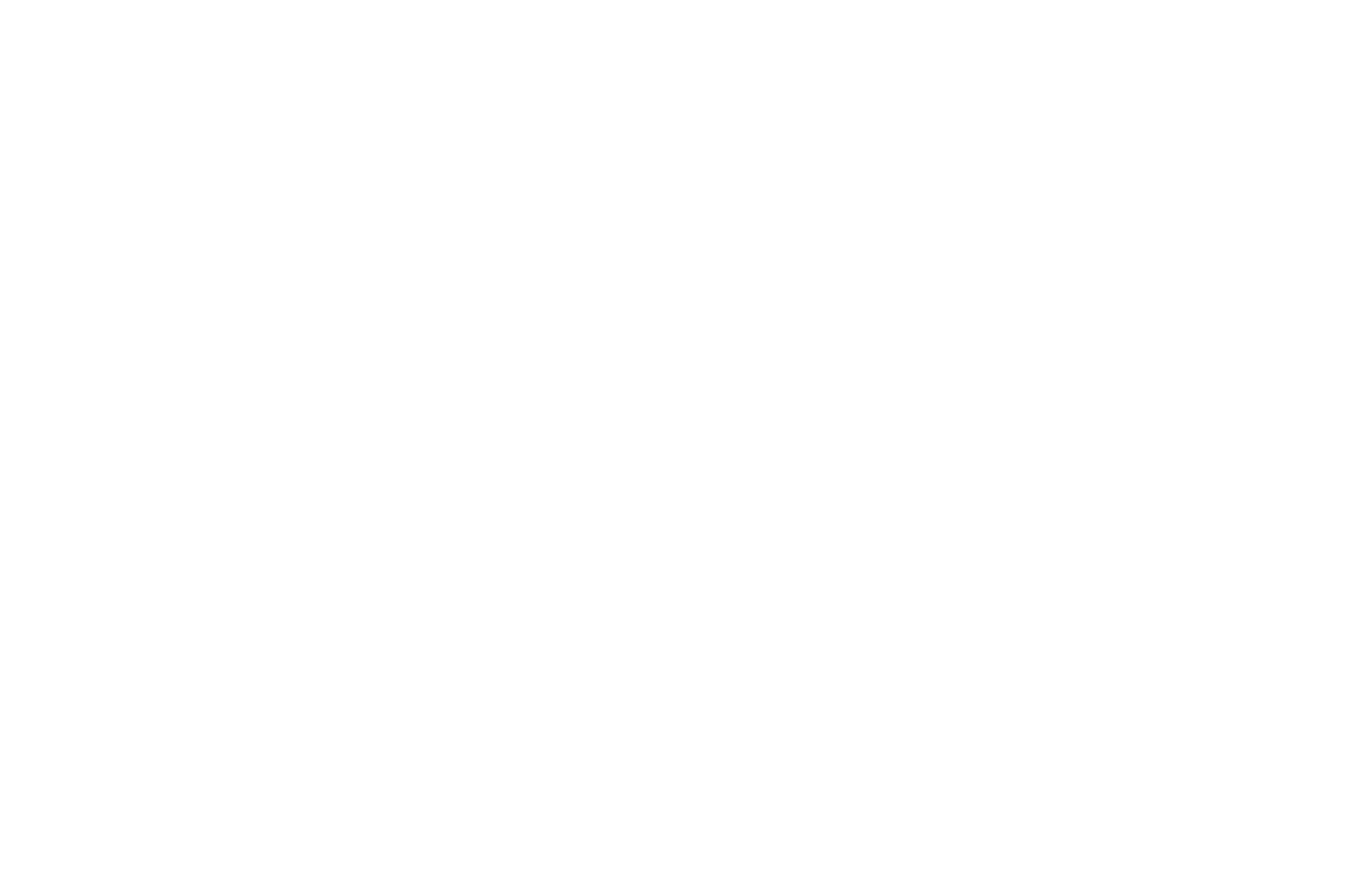
Methods: We evaluate ApiLinker by constructing representative pipelines spanning (i) bibliographic retrieval and normalisation (CrossRef → Semantic Scholar), (ii) literature sampling and metadata export (NCBI/PubMed → CSV), and (iii) issue migration between software repositories (GitHub → GitLab). Each pipeline is specified declaratively; correctness is assessed via schema conformance and record-level invariants (e.g., identifier preservation, date normalisation). Robustness is probed through induced transient failures (timeout, 5xx, 429) to verify retry, backoff, and circuit-breaking behaviour. Performance metrics include end-to-end throughput (records/second) and error-handled latency under controlled network perturbations using the included benchmark harness.

Results: The pipelines execute end-to-end without code changes across environments, with deterministic mappings and stable behaviour under induced failure modes. Circuit breakers reduce cascading failures by isolating misbehaving endpoints; DLQ capture preserves failure artefacts for audit and replay. Throughput scales with available concurrency in the underlying API rate limits, and configuration reuse reduces development effort compared to bespoke scripts. Detailed scenario descriptions and measured outcomes are available in the benchmarking utilities and examples directory.

Table 1 summarises representative evaluation scenarios, their primary objectives, and observed outcomes under nominal and fault-injected conditions.

| Scenario | Source → Target | Objective | Nominal outcome | Under faults |
| --- | --- | --- | --- | --- |
| Bibliographic enrichment | CrossRef → Semantic Scholar | Join DOIs with citation metadata | 100% schema-conformant records; deterministic mapping | Retries/backoff recover 429/5xx; DLQ captures irrecoverable items |
| Literature sampling | NCBI/PubMed → CSV | Reproducible export of abstracts/metadata | Stable throughput; exact record counts | Timeouts isolated by circuit breaker; replay via DLQ |
| Issue migration | GitHub → GitLab | Preserve titles, bodies, labels | Invariants satisfied (IDs, labels) | Intermittent failures retried; partial batches preserved |





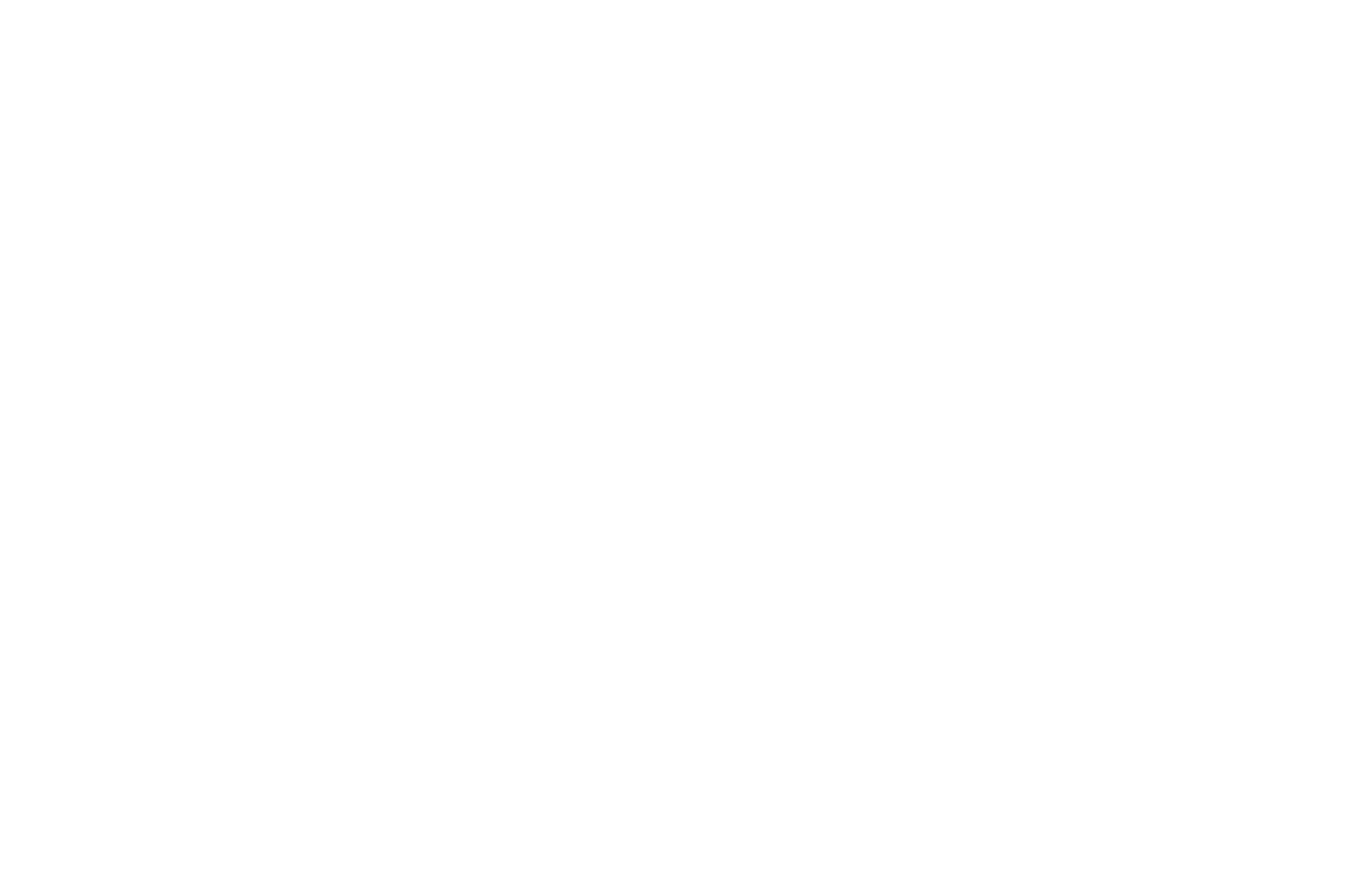
# Implementation and architecture

ApiLinker is implemented in Python (3.8+) with a deliberately small dependency surface to facilitate long-term maintainability and ease of installation. The architecture adheres to separation-of-concerns and explicit dependency boundaries:

* ApiConnector: transport and protocol concerns (HTTP requests via httpx), endpoint specification, pagination strategies, and coarse-grained retrying; responses are normalized to dictionaries/lists and optionally narrowed via response-path expressions.
* FieldMapper: structure-level mapping between source and target schemas using dot-path addressing, list indexing, conditional inclusion, and composable value transformations; designed to be deterministic and side-effect free.
* AuthManager: uniform configuration of authentication schemes (API key, bearer, basic, OAuth2 variants including PKCE and device flow), with refresh handling when available.
* Scheduler: interval/cron-based orchestration for unattended execution using a lightweight background thread model.
* PluginManager: discovery and registration of user-defined transformers and connectors to support domain-specific semantics without modifying the core.

Error management is centralized via category-aware exceptions and recovery strategies (e.g., exponential backoff, DLQ handoff), enabling consistent observability and post-hoc analytics across connectors. The system offers both a Python API and a CLI; configurations are expressed in YAML/JSON and may reference environment variables to externalize secrets.

Figure 1 depicts the system architecture, emphasising the separation between transport (connectors), schema transformation (mapping/transformers), authentication control, orchestration (scheduler), and observability (logging/analytics). This separation permits independent evolution of components and narrow interfaces that are straightforward to reason about, test, and document.



Implementation constraints: ApiLinker targets RESTful JSON APIs with synchronous request/response semantics. Binary payload handling and streaming/event-driven paradigms (e.g., Server-Sent Events, WebSockets, webhook choreography) are not first-class in the current release; users may wrap such patterns via custom connectors. Concurrency is conservative by design to minimise provider-side rate-limit violations; advanced concurrency models can be layered externally if providers permit.

## Mapping semantics and transformers

The mapping subsystem formalises the transformation from source records to target payloads using four core constructs:

* Path addressing: Dot-separated keys (e.g., user.profile.name) navigate nested documents; array indexing is supported (e.g., items[0].title). Missing paths resolve to None without raising, enabling conditional logic.
* Field rules: Each rule specifies source, target, optional transform, and optional condition. Rules are evaluated in order and write into a fresh target document using target paths.
* Transform composition: Transforms are pure functions applied by name; multiple transforms can be composed sequentially. Built-ins include casing, trimming, numeric conversion, timestamp conversion, and null-handling; users can register custom transformers at runtime or package them as plugins.
* Conditional inclusion: Rules may include a condition with an operator (eq, ne, gt, lt, exists, not\_exists) evaluated against a path in the source item; if the predicate fails, the rule is skipped.

Determinism: Given an input document and a fixed mapping specification, the mapper is deterministic. Side-effects are confined to logging. Complexity is linear in the number of mapping rules and input records; path resolution is linear in path depth.

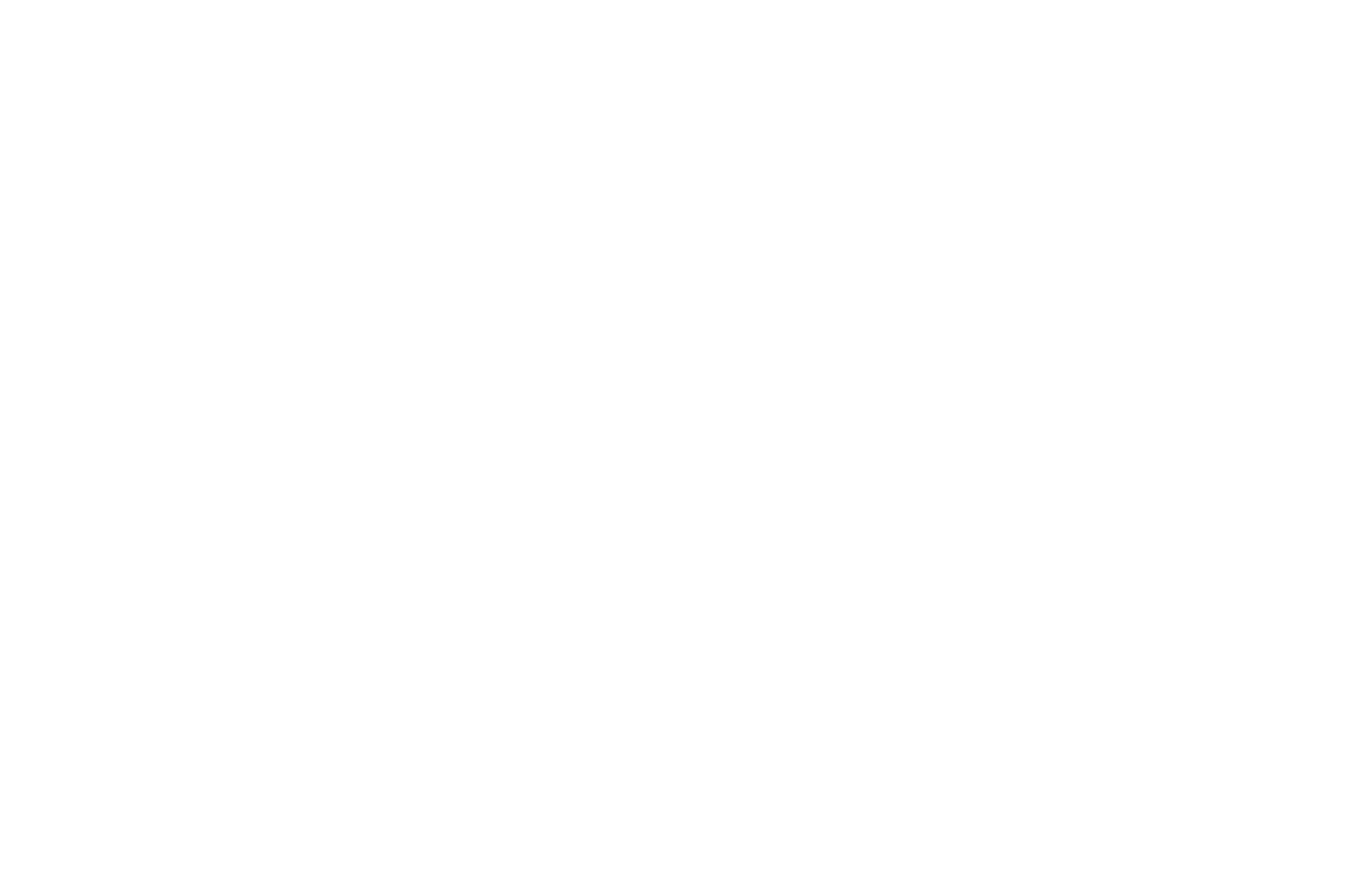
# Quality control

The codebase includes unit and integration tests spanning connectors, mapping, security, scheduling, CLI, and end-to-end sync flows (see tests/). Continuous integration executes tests across supported Python versions with static checks (type checking and linting). Coverage instrumentation is supported to enforce minimum thresholds during CI execution (see docs/coverage.md).

Empirical validation is conducted via runnable examples and scenario-based tests that simulate typical research tasks (e.g., literature retrieval, compound lookup, and cross-platform issue migration). Failure modes such as transient network errors, rate limits, and schema drift are exercised through the error-handling layer to assess recovery strategies (retry with backoff, circuit breaking, DLQ capture). Documentation (quick starts, tutorials, API reference) reduces user error and supports reproducible configuration.

User verification: To facilitate reproducibility checks by reviewers and users, we provide: (i) a configuration validator (apilinker validate) to perform structural checks prior to execution; (ii) dry-run mode to preview mappings and planned operations without side-effects; and (iii) deterministic transformers for common tasks (case conversion, trimming, timestamp conversions). Where applicable, example configurations include fixed seeds and pinned query parameters to ensure stable outputs.

Verification protocol: Reviewers can reproduce core behaviours by (1) installing the package in a clean environment, (2) running apilinker init to generate a template config, (3) executing apilinker validate and apilinker sync --dry-run, and (4) populating environment variables for optional API keys to run a minimal end-to-end transfer with a public endpoint (see docs/getting\_started.md). CI badges and coverage configuration are provided in the repository to support automated checks.



# Availability

## Operating system

Platform-independent; tested on Linux, macOS, and Windows.

## Programming language

Python (>= 3.8)

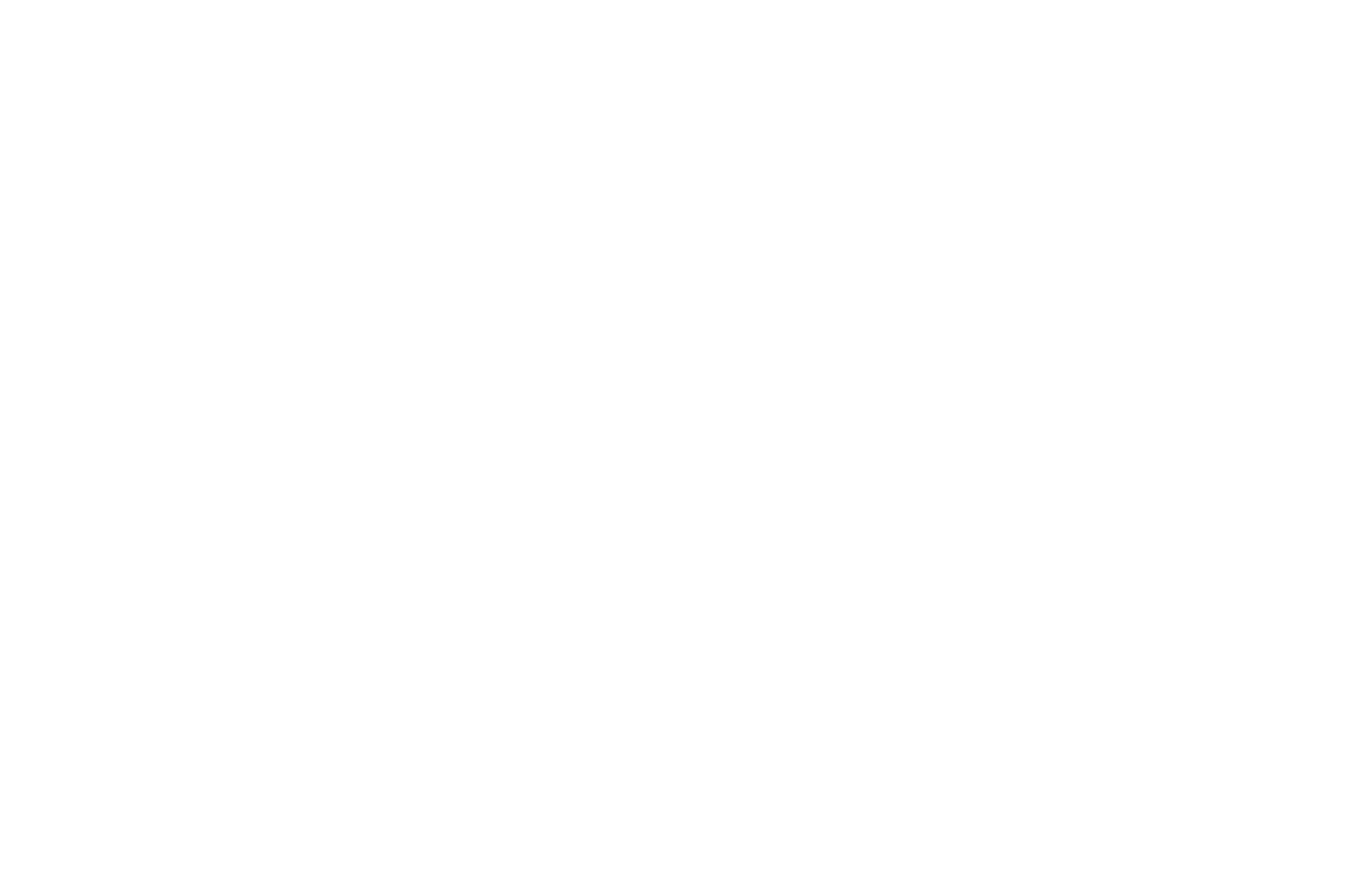
## Additional system requirements

No special hardware requirements. Internet connectivity required for interaction with remote APIs.

## Dependencies

Runtime dependencies:

* httpx, pyyaml, typer, pydantic, croniter, rich, cryptography Development/docs extras: pytest, pytest-cov, flake8, mypy, sphinx (see pyproject.toml).



## List of contributors

Kyriakos Kartas (lead developer and maintainer). Community contributions acknowledged in version control history.

## Software location: archive

* Name: Zenodo (planned)
* Persistent identifier: DOI: 10.5281/zenodo.TBD
* Licence: MIT
* Publisher: Zenodo
* Version published: 0.3.0
* Date published: 2025-07-14

## Software location: code repository

* Name: GitHub
* Identifier: kkartas/APILinker
* Persistent identifier: https://github.com/kkartas/APILinker
* Licence: MIT
* Date published: 2023–ongoing

## Language

English

## Data accessibility statement

No proprietary or third-party datasets are distributed with the software. Example configurations and scripts reference openly accessible public APIs (e.g., arXiv, CrossRef, NCBI, NASA) subject to the terms of those services. Users should ensure compliance with provider terms of use and attribution norms. Reproducible configurations used in examples can be executed without special permissions; where API keys are optional, they are referenced via environment variables. A citable software release will be archived on Zenodo; associated example outputs are generated on demand.

## Installation and quick verification

Install from PyPI (pip install apilinker) or from source (pip install -e .). Verify with:

apilinker --help  
apilinker init --output demo.yaml --force  
apilinker validate --config demo.yaml  
apilinker sync --config demo.yaml --dry-run

If optional API keys are available, set them in the environment and re-run the dry run as an actual transfer to confirm end-to-end operation.

# Reuse potential

ApiLinker is intended as a reusable substrate for constructing research data acquisition and interoperability pipelines across domains. Reuse is facilitated by:

* Version-controlled configurations that can be archived alongside datasets and analysis code to ensure procedural reproducibility [@Wilkinson2016].
* A generalised mapping model that handles heterogeneously nested structures and value transformations without bespoke imperative code.
* Extensibility points (connectors, transformers) enabling domain-specific adaptation with minimal coupling to the core.
* Operational robustness via retrying, circuit breaking, and DLQ capture, supporting long-running longitudinal studies.

Typical scientific uses include: citation graph construction (CrossRef, Semantic Scholar), literature sampling for systematic reviews (NCBI/PubMed, arXiv), compound and assay aggregation (PubChem), researcher disambiguation (ORCID), and climate data retrieval (NASA) combined with downstream analytics. Community support is provided via issue tracking and contribution guidelines; the documentation and examples accelerate onboarding and promote correct reuse.

Guidance for extension: New connectors can subclass the connector interface, implement endpoint specification, and optionally provide response-path extraction and pagination strategies. Transformers can be registered at runtime or packaged as plugins. For governance, we recommend contributors include unit tests, minimal reproducible examples, and documentation snippets so that new extensions preserve the library’s guarantees of determinism and reproducibility.

# Limitations and future work

ApiLinker currently assumes REST-style JSON APIs and relies on user-provided mappings; semi-automated schema inference and validation would further reduce configuration effort. Native support for streaming endpoints and event-driven choreography (e.g., webhooks, message queues) is limited and is an area for future extension. While OAuth variants are supported, pluggable integrations with enterprise secret managers could improve operational security in some deployments. Finally, richer telemetry (metrics/tracing) and policy-driven backpressure could enhance observability and stability in high-throughput scenarios.

Planned enhancements include: (i) schema probing utilities to auto-suggest initial mapping templates; (ii) first-class webhook and message-queue connectors to support event-driven pipelines; (iii) optional integrations with secret management systems (e.g., HashiCorp Vault, AWS Secrets Manager) with least-privilege defaults; and (iv) structured metrics/trace emission (OpenTelemetry) for production observability.

# Comparison with existing tools

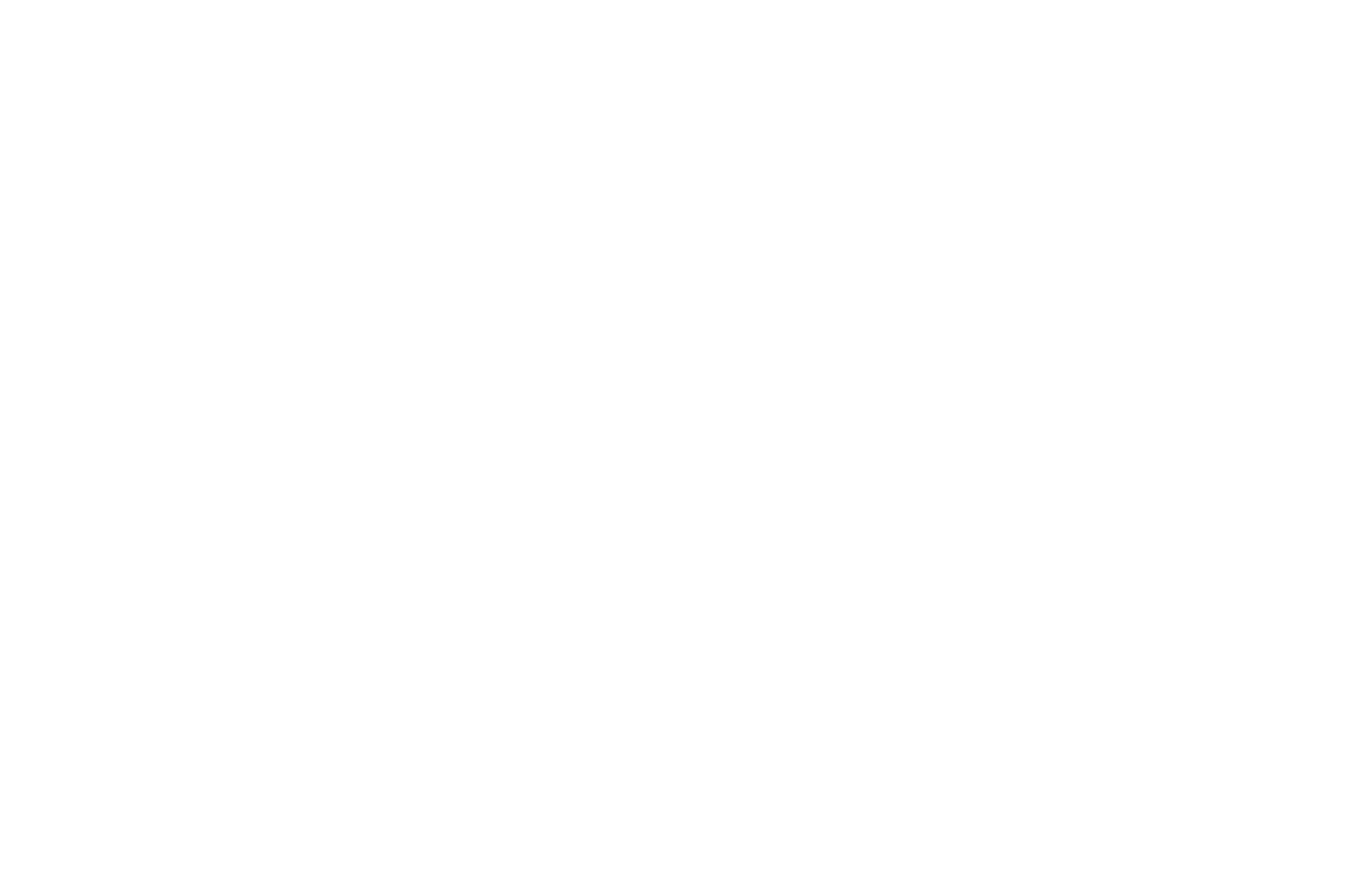
ApiLinker occupies a unique position in the API integration ecosystem, balancing simplicity with research-specific requirements:

| Feature | ApiLinker | Apache Airflow | Zapier | n8n |
| --- | --- | --- | --- | --- |
| Configuration-driven API mapping | ✓ | Partial | ✓ | ✓ |
| Advanced data transformations | ✓ | Partial | Limited | Limited |
| Open source | ✓ | ✓ | ✗ | Partial |
| Local deployment | ✓ | ✓ | ✗ | ✓ |
| Minimal dependencies | ✓ | ✗ | N/A | ✗ |
| Python-native library | ✓ | ✓ | ✗ | ✗ |
| Research workflow focus | ✓ | Partial | ✗ | ✗ |

**Distinctive advantages** for research applications:

* **Minimal infrastructure**: Unlike Airflow's complex architecture, ApiLinker runs as a simple Python library
* **Configuration reproducibility**: YAML configurations can be version-controlled and shared with research data
* **Embedded usage**: Designed to be imported and used within existing research codebases
* **Research-oriented features**: Built-in support for common research patterns like pagination, retry logic, and data validation

Limitations of comparators: Airflow targets workflow orchestration at cluster scale, not schema-level mapping between arbitrary web APIs; Zapier and n8n emphasise GUI-driven automation with constrained transformation semantics and limited research provenance. ApiLinker complements such systems by focusing on reproducible, text-based configurations, determinism, and integration as a library component within scientific codebases.



# Conclusion

ApiLinker fills a significant gap in the open-source ecosystem by providing a flexible, configuration-driven approach to REST API integration specifically designed for research workflows. The software addresses common pain points in research computing: the need for reproducible data pipelines, minimal infrastructure dependencies, and robust handling of the diverse API patterns encountered in interdisciplinary research.

By abstracting common integration patterns into a declarative configuration model with strong typing, validation, and error handling, ApiLinker enables researchers to focus on their domain-specific analyses rather than the mechanics of API communication. The plugin architecture ensures adaptability to new API types and transformation requirements without modifying the core codebase, supporting the evolving needs of research communities.

The software's design principles—reproducibility, simplicity, and extensibility—align with open science and FAIR data practices, making ApiLinker a pragmatic foundation for reliable, maintainable API integrations within research software engineering.

## Figure preparation guide (for authors)

To create and include the figures referenced in this manuscript:

* Figure 1 (Architecture): Export a high-resolution PNG from your architecture source (e.g., draw.io/Lucidchart). Use consistent typography, align boxes and arrows, and include a legend if needed. Target width ~1400–1800 px.
* Figure 2 (Mapping workflow): Derive from docs/architecture.md mapping diagram or create a new schematic illustrating source fields, mapping directives (including transforms), and target fields. Keep labels readable and show at least one conditional mapping.
* Figure 3 (Positioning chart): Create a simple 2D scatter/axis chart (e.g., in matplotlib or vector tool) with axes “Reproducibility” and “Transformation expressiveness”; place ApiLinker, Airflow, Zapier, n8n.
* Figure 4 (Benchmarks): Run benchmarks/run\_benchmarks.py or reproduce scenarios from benchmarks/scenarios.py; collect records/second and latency; plot with error bars using matplotlib; export PNG.
* Figure 5 (Error flow): Translate the error-handling flow diagram from docs/architecture.md into a clean flowchart with states for circuit breaker and DLQ.
* Figure 6 (CLI screenshot): On a terminal with a demo config, capture outputs of apilinker validate, apilinker sync --dry-run, and a short scheduled run; use a monospace font and high contrast theme.

File placement: Save images under paper/figures/ using the filenames referenced in the manuscript. Ensure captions are concise and descriptive. Verify that images render in your chosen manuscript build pipeline (e.g., pandoc or journal submission system).

# Funding statement

No external funding was received for this work.

# Competing interests

The author declares no competing interests.

# Acknowledgements

We acknowledge contributions from the open-source community and feedback from early adopters who helped shape the design and functionality of ApiLinker.

# References