Engineering Entity Resolution for the LLM Era: Requirements, Architecture, and Evaluation of an Open-Source Library

Álvaro de Carvalho Alves
PESC - UFRJ
Rio de Janeiro, Brazil
alvaro@cos.ufrj.br

Cláudia Maria Lima Werner
PESC - UFRJ
Rio de Janeiro, Brazil
werner@cos.ufrj.br

Diego Cardoso Borda Castro
PESC - UFRJ
Rio de Janeiro, Brazil
diegocbcastro@cos.ufrj.br

Abstract

Entity resolution (ER) research has converged on dense embeddings for blocking and large language models (LLMs) for matching, but open-source tooling has not followed: mainstream libraries centre on older method families, and where LLM matching appears, a matcher failure (a refusal, a timeout, or malformed output) is silently scored as a non-match and enters the reported quality metrics. This paper frames packaging that pipeline as a software-engineering problem for an LLM-enabled system, and presents Denselinkage, an open-source, typed Python library structured as a requirements-driven ports-and-adapters architecture: ten typed ports around a `numpy`/`pandas` core, with heavy backends behind optional extras. Its distinctive decision is a typed *failure contract*: a matcher failure becomes a per-pair typed outcome, retried or excluded from precision and recall, not scored as a non-match. The architecture is evaluated against its requirements using re-runnable checks: the public contract stayed stable across three releases and evolved only by addition, heavy dependencies were kept out of the core, and structured-output matching produced no malformed verdicts, whereas a brittle free-text parser on the same data masked a 99% failure rate on a deliberately weak model. Controlled fault injection shows the silent convention understating F1 by 9.7 points at a 20% failure rate, and shows exclusion to be conditional on coverage when failures are not random. The library and a reproducibility package are openly available.

Keywords

Entity Resolution, Software Engineering for AI, ML-Enabled Systems, LLM Reliability, Software Architecture, API Design, Ports and Adapters, Large Language Models, Software Reuse

1 Introduction

Entity resolution (ER), also called record linkage or deduplication, decides which records refer to the same real-world entity [4]. Because comparing all pairs is quadratic in the input size, practical systems use two stages: a cheap *blocking* stage proposes candidate pairs, and a more expensive *matching* stage decides them [19]. Both have changed over the last decade. Matching moved from probabilistic and string-similarity models to fine-tuned language models [14], and more recently to prompted LLMs that match competitively with little or no task-specific training [21, 31]. Blocking moved from key- and token-based schemes to dense embeddings with nearest-neighbour retrieval, validated experimentally as a strong default for candidate generation [30, 37]. The effectiveness of this combination is established in the literature.

The engineering of the combination is not. As Table 1 shows, the libraries practitioners install centre on older method families, and where the newer components appear they lack the robustness, packaging, and stability that production pipelines require. This paper presents Denselinkage, an open-source, typed Python library that implements the modern ER pipeline as a reusable component architecture, and reports its design and evaluation as software engineering for an LLM-enabled system. The artifact is a stable `1.0.0` release in which embedding-based blocking, pluggable threshold or LangChain-compatible LLM matching, and clustering are components behind ten typed ports around a dependency-free core, with heavy backends as optional extras. Its requirements trace to four sources: the domain literature, the Resolvi reference architecture [18] (which we believe this library is the first to implement), the shortcomings of existing tools, and empirical API-design research [3, 17]. The most distinctive decision is a typed *failure contract* that engineers reliable behaviour around an unreliable LLM component: an LLM failure is a typed, per-pair outcome excluded from quality metrics rather than a silent non-match, adapting the reject-option and coverage framing of selective prediction [6] into a typed API that propagates the abstention downstream.

Table 1: Open-source ER libraries and the modern pipeline. Dense: embedding-based blocking. LLM: LLM-based matching. Fail.: a typed matcher-failure outcome excluded from quality metrics. Light: a dependency-light core with heavy stacks as optional extras. Typed: a public contract shipped to type checkers. A check (✓) marks full support, a dot (•) partial support, a blank cell none. Verified June 2026 against current releases, source code, and issue trackers.

Library	Dense	LLM	Fail.	Light	Typed
Splink	•			✓	✓
recordlinkage				✓	
pyJedAI	✓	•			
Libem		✓			
dedupe, py_entitymatching, Ditto, DeepMatcher					
Denselinkage	✓	✓	✓	✓	✓

The distinction is what a developer sees when a pipeline degrades. A matcher that refuses, times out, or returns text no parser accepts has produced no evidence about the pair, yet the examined tools record it as a decided non-match. On `llama3.2:1b` that convention reports F1 0.044 on DBLP–ACM, which reads as a weak matcher and invites threshold tuning; the same run under the contract's accounting reports F1 0.615 over the 1.3% of pairs the model actually decided, which identifies the model as unusable and sends the developer to the matcher instead (Section 4).

The table covers maintained general-purpose Python ER libraries on PyPI, plus two research codebases (Ditto, DeepMatcher) widely used as accuracy baselines. The last four share none of the five properties and are listed together, as evidence that the gap is the field's rather than any one project's. The columns are this paper's own design axes, so on other criteria several libraries lead: accuracy (Ditto, DeepMatcher, Libem), maturity (Splink, dedupe), scale (Splink), and active learning (dedupe). Mainstream libraries centre on Fellegi–Sunter probabilistic linkage [15], learned string similarity, or feature-based machine learning; of the libraries in Table 1, embedding-based blocking is native only to pyJedAI [13]. Where LLM matching exists, it lacks engineering for failure: pyJedAI's LLM module decides a match by comparing the raw model response against a fixed string, so a refusal or a hedged answer is recorded as a non-match and enters the library's own evaluation, and Libem [8] similarly substitutes a default negative answer for failed responses. In both, the API, the result object, and the reported F1 do not distinguish an LLM failure from a genuine non-match. The gap Denselinkage addresses is the engineering of the established modern pipeline: typed, explicit about failure, light in its dependencies, and stable in its public contract.

2 Denselinkage: Requirements, Functionality, and Architecture

2.1 Requirements

The requirements were derived from four sources before committing to a design: (i) the ER pipeline and its current methods (Section 1); (ii) the Resolvi reference architecture for entity resolution [18], a Type-3 reference [1] synthesised from nine existing systems, whose quality attributes and component inventory were examined systematically; (iii) the shortcomings observed in the tool landscape (Table 1); and (iv) empirical API-design research: small surfaces hard to misuse [3, 17], defaults over required constructor parameters [28], operations discoverable from one central object [29], and packaging practice [22, 23, 27].

Seven functional requirements (FR) name capabilities and correspond to the stages of the modern pipeline: embedding-based blocking with retrieval parameters adjustable without rebuilding the index (FR1); LLM matching with structured output, per-pair retry, and a typed soft-failure outcome so a failure is distinguishable from a non-match (FR2); a dependency-free reference stack that runs the whole pipeline with no API key (FR3); an end-to-end pipeline in a few operations (FR4); built-in evaluation with pairwise P/R/F1 from which failed pairs are excluded, pair completeness at k [19], and B^3 [2] (FR5); reuse of the costly embedding step through build-once/query-many operation and persistence (FR6); and pandas interoperability with distributed type information (FR7).

Five quality requirements (QR) name properties the architecture must have and drive the design decisions in Section 2.3: extensibility, so a new embedder, index, or matcher arrives as a new adapter behind a port rather than a modification of the core [9] (QR1); a light footprint, so a user who needs only lexical comparison installs no FAISS, PyTorch, or GPU stack and the core imports none of the heavy backends [23] (QR2); robustness, so a single failed pair neither aborts a batch nor enters precision and recall, and a programmer error stays distinguishable from a data error (QR3); evolvability, so the public contract is typed, stable under semantic versioning [22], and evolved only by extension, because breaking changes drive users away [24, 35] (QR4); and usability, a short quickstart with sensible defaults over a strictly typed surface (`py.typed` [27]) that IDEs and type checkers consume [28] (QR5).

```python
from Denselinkage import DenseLinker, LabeledPairs, Source
from Denselinkage.metrics import linkage_metrics
# left, right: two pandas DataFrames, each with an "id" column

linker = DenseLinker.with_defaults()  # lexical stack:
result = linker.link(                 # no extras, no API key
    Source(left,  id_column="id"),
    Source(right, id_column="id"))
print(result.to_frame().query("match"))

# gold: the known true matches (ground truth)
gold = LabeledPairs.from_pairs([("A1","B1"), ("A2","B2")])
m = linkage_metrics(result, gold=gold)  # errored pairs are counted
      apart, never in P/R

# Upgrading to semantic blocking + LLM matching changes the
# assembly only; link() above is called exactly as before.
linker = DenseLinker(
    blocker=DenseBlocker(
        embedder=SentenceTransformerEmbedder("all-MiniLM-L6-v2"),
        vector_index=FaissFlatIndex(), top_k=5),
    matcher=LangChainMatcher(
        llm=ChatOpenAI(model="gpt-4o-mini"),
        retry=RetryPolicy(max_retries=3)))
```

Figure 1: The quickstart (top) and the upgrade to semantic blocking with LLM matching (bottom). Replacing the embedder, the index, and the matcher changes only the assembly; the call to `link` is unchanged. Imports and the matcher's `prompt=` argument are elided in the lower block.

Two of these, the typed failure outcome and the dependency-light core, were derived from observed tool gaps, a circularity addressed in the threats to validity. Incremental indexing, trainable components (a reserved `Trainer` port), and distributed execution were deferred or declined by recorded decision.

2.2 Functionality

Denselinkage supports three resolution operations. It resolves entities across two tables (`link`), within one table (`dedupe`, which suppresses self-pairs and symmetric duplicates), or over caller-supplied candidate pairs (`match_pairs`, for externally blocked data). Figure 1 shows the complete quickstart. With no extras installed and no API key, `with_defaults()` assembles the dependency-free lexical stack: character n-gram embeddings computed by feature hashing, exact top-k search, and a similarity-threshold matcher. One call links the sources, scores the pairs, and returns a typed result convertible to a DataFrame. Semantic and LLM adapters are added when the data require them without changing the call site. Inputs are pandas DataFrames wrapped in a `Source` that names the identifier column and, optionally, how a record is rendered to text through the `Serializer` port, which also reconciles schema differences between sources.

The evaluation suite's distinguishing property is its failure accounting. `linkage_metrics` computes pairwise precision, recall,

and F1 with gold pairs that blocking never surfaced counted as false negatives (so recall is end-to-end rather than conditional on blocking), while failed pairs are excluded from every category and reported separately as `n_errors`; `pair_completeness_at_k` [19] and B^3 [2] score blocking and clustering. Embedding the reference side is the costly step (FR6), so `DenseLinker.index(source)` builds a reusable `LinkageIndex` for which the identity `link(a,b) == index(a).query(b)` holds by construction, and whose dependency-free form persists as inspectable `numpy` and JSON files rather than `pickle`, refusing to load under an embedder other than the one that built it. The base installation declares `numpy` and `pandas` only; the `[sentence-transformers]`, `[faiss]`, and `[langchain]` extras [23] add the heavy stacks, and requesting a heavy adapter without its extra raises an error naming the exact `pip install` command.

2.3 Architecture and Design

Denselinkage is layered as a ports-and-adapters structure [5] in which every dependency arrow points inward to the core, which imports no adapter and no heavy backend at runtime. That dependency-free core holds the immutable domain types (frozen dataclasses: `Record`, `CandidatePair`, `MatchDecision`, its sibling `MatchError`, and `Source`), the ten ports expressed as `typing.Protocol`s, the result types, and a small exception taxonomy. The adapter subpackages (embedding, indexing, blocking, filtering, matching, clustering, serializing) implement ports; `DenseLinker` and `LinkageIndex` orchestrate them. The core is the centre of the dependency graph rather than the largest component: at `1.0.0` the adapters exceed the core in code size (911 versus 586 lines) because behaviour lives in adapters, while the core is what every module depends on and what remains stable.

Each of the ten ports is a Strategy [9] expressed as a `Protocol`, encapsulating one axis of variation: how a record becomes text (`Serializer`), how text becomes a vector (`Embedder`), how vectors become a searchable index and then neighbours (`VectorIndex`/`SearchableIndex`), how records become candidate pairs (`Blocker`/`BlockingIndex`), how pairs are pre-filtered (`Filter`), how candidates become decisions (`Matcher`), how decisions become clusters (`Clusterer`), and, reserved for future training, `Trainer`. The other nine ship a dependency-free reference adapter and, where relevant, a heavy adapter behind an extra. Two of these ports (`Clusterer`, `Filter`) and the clustering-metrics surface were added in response to gaps identified against Resolvi. The orchestrator depends only on ports, so substituting any stage is invisible to it (QR1), and the LLM is one adapter among others: constructed by the caller and injected, never created inside the library.

The central state-management decision addresses the reuse requirement (FR6) by separating two concerns. A *specification* is stateless configuration whose `build()` method is a pure factory returning a new, immutable *artifact* that holds all fitted state, so one specification can build many artifacts without interference, an artifact can be shared and cached safely, and the identity `link(a,b) == index(a).query(b)` holds because building is pure. The same structure repeats one level up (`DenseLinker` builds `LinkageIndex`), following the estimator/transformer convention of scikit-learn [20]. `DenseLinker` itself is an immutable, keyword-only dataclass owning no records or vectors, which is what makes one configuration reusable across datasets [28].

Failure handling realises the robustness requirement (QR3) as three disjoint tiers. Per-pair soft failures are values: `Matcher.match` returns, for each input pair, either a `MatchDecision` or a `MatchError(reason)`. The `LangChainMatcher` binds structured output to a standard-library `TypedDict`, so the response format is owned by the schema and the library never parses free text; failing pairs are retried under a `RetryPolicy`, and exhausted failures become position-aligned `MatchError` values that the result carries in a channel disjoint from `decisions`. Hard data failures, such as an unknown identifier column or a dimension mismatch, form a small exception family rooted at `DenseLinkageError`. Programmer errors raise plain `ValueError`, intentionally outside that family, so a caller who writes `except DenseLinkageError` to handle data problems cannot inadvertently suppress a defect in their own code.

Dependency isolation gives the light-footprint requirement (QR2) its implementation in the import graph: heavy backends are imported inside the methods that use them. The guarantee is enforced as an architectural fitness function [7]: a CI test starts a fresh interpreter, imports the package, and fails the build if `faiss`, `torch`, `sentence-transformers`, or `langchain` appears in `sys.modules`. The same rule rejected `pydantic` for the domain types.

The typing discipline rests on structural `Protocol`s for the ports, which permits third-party extension without coupling: an external class with the required methods satisfies the `Matcher` type without importing Denselinkage or inheriting from it, whereas an abstract base class would require both. First-party adapters do subclass their port, so `mypy --strict` verifies each implementation is complete, and all fifteen adapters pass this check. The package ships `py.typed` [27], so these guarantees reach users' IDEs and type checkers.

3 Development Process

The process is not itself a contribution, but it explains how the public contract acquired the stability Section 4 measures. Development was contract-first because breaking changes to a published interface are among the costliest defects a library can produce [24, 35]: the entire public surface was written first as typed stubs with empty bodies, type-checked under `mypy --strict`, and frozen, after which changes are restricted to additions. Three checks validated the frozen surface. Comparing it against Resolvi [18] resolved each divergence into surface to add, behaviour to defer, or a recorded decline. The example programs are treated as the specification of intended use and are compiled in CI. The third proved the most productive: a minimal vertical slice against the stubs. The original index ports modelled construction as in-place mutation and passed every static check, yet an implementation attempt showed that a frozen configuration object holding a mutable index forces either defensive copying or state corruption. The specification/artifact separation corrected this at the cost of two signature-only ports before the freeze; discovered afterward, the same fix would have broken every index adapter. Only the contract was fixed early, while

behaviour was implemented incrementally across three releases, so the process is not big design up front.

4 Evaluation

The architecture is evaluated against its requirements using checks a reader can re-run (user-centred usability evaluation [25] is future work). Most requirement checks are *descriptive*, recording how a named decision realises the goal it was designed for; the *validating* weight rests on evidence the design did not place and a third party can re-derive: the version-control history (QR4), the dependency-cut fitness function (QR2), and the failure runs. Matching accuracy is not the target, because the effectiveness of embedding blocking and LLM matching is established in the literature [21, 37], and accuracy is a property of prompt and model rather than of the architecture.

Contract stability (QR4) is the strongest evidence because it is longitudinal and independent of our judgment. From the pre-beta freeze to `1.0.0` the package grew from 1,703 to 2,732 lines while `core/ports.py` received no modification at all, an empty diff reproducible from the public tags; the only core changes were additive. What matters is where the contract held: the LLM matcher, the component most likely to expose a missing affordance, required only an adapter behind the unchanged `Matcher` port, its failure modes carried by the existing error channel.

Extensibility (QR1) holds at the port boundary: all fifteen adapters pass `mypy --strict`; three backend swaps (lexical to sentence-transformer embeddings, numpy to FAISS, threshold to LLM) run with no core change, a differential test confirming the FAISS index returns the numpy reference's neighbours; and a custom embedder is about 35 lines, importing only its port.

Whether parties who did not design the contract can implement it was probed out-of-sample, with every existing adapter withheld. *This probe indicates feasibility and is not a controlled experiment: three participants and five single-shot agent attempts per port cannot support a general claim about extensibility or usability* [34]. The developers' task was one `Matcher` adapter deciding a pair by `difflib` string similarity, with the algorithm, the threshold, and the empty-text case specified, so it tests whether a specified component can be mapped onto the contract, not whether a good adapter can be designed [36]. Participants received the `Matcher` port and the value types, and self-checked against five criteria: `mypy --strict` clean, explicit subclassing, one outcome per input pair, a `MatchError` on empty text, and an end-to-end run in a real pipeline. Three independent developers each passed all five on the first run, self-reporting 5, 5, and 25 minutes and no look-ups beyond the handout; an LLM coding agent made five single-shot attempts at each of two ports (`Matcher` and the harder `Embedder`) and all ten conformed. No participant produced a contract violation, and the friction they reported lay in the task specification: whether `CandidatePair.similarity_score` is informational or a usable hint, and whether empty text includes whitespace-only text. The light core (QR2) survives the heavy adapters: each imports its backend lazily, the base install declares exactly `numpy` and `pandas`, and a dedicated CI job runs the dependency-free surface with the heavy stacks absent, the import test failing the build if any of them loads. Domain coverage was checked against Resolvi [18]: the engine pillars are covered, while training and filtering are declared in the contract but not yet wired.

An end-to-end run exercises the assembled pipeline. On DBLP–ACM [14, 16] (2,616 and 2,294 records, 2,220 matches) the semantic stack (`all-MiniLM-L6-v2` over FAISS) blocks the matches almost completely (pair completeness 0.970 at the nearest neighbour, 0.997 at the twentieth, against 0.990/0.999 for the lexical stack), a real `gemini-2.5-flash` matcher decides a balanced 200-pair sample with no failures, and the lexical stack ran the same benchmark without an API key. These are liveness checks, not quality claims. The hosted runs used `gemini-2.5-flash-lite` and, where rate limits required, `gemini-2.5-flash`; no comparison reported here is between the two.

That running pipeline is where the failure contract is needed. A matcher failure (a refusal, a timeout, a rate limit, or a response from which no verdict can be recovered) occurs on real serving workloads [32] and under structured-output generation at load [11]. Under the silent-non-match convention, a failure on a true match becomes a false negative, so the reported F1 falls as the failure rate rises even though the matcher's decisions are unchanged. The contract answers this in two classes. For the format class, binding structured output left no malformed verdict: across `gemini-2.5-flash-lite` (4,000 pairs) and two deliberately weak local models, `llama3.2:1b` and `qwen2.5:0.5b` (1,000 each), none occurred, and the 0.5B model's weakness appeared as poor *decisions*, not failures. The weak models were selected on parameter count alone, as the smallest instruction-tuned models in their families, to represent the low-capability end a practitioner can run locally, and two controls argue against a cherry-picked failure: the same family at 7B (`qwen2.5:7b`) scores F1 0.875 under pyJedAI's rule on the full candidate set, so the collapse tracks capability rather than a family-specific output habit, and re-running `llama3.2:1b` with a fourfold output budget leaves the unparseable rate at 99.3%, so it is task failure rather than truncation. The contrast with the brittle convention is sharp (Table 2): pyJedAI's rule accepts only a response literally equal to `True`, so on both weak models, whose responses are entirely non-canonical, it scores F1 0.000, though on `qwen2.5:0.5b` a lenient parser recovers the verdicts. On `llama3.2:1b` the output resists even a lenient parser, leaving no verdict on 99% of pairs: folding those into non-matches reports F1 0.044, whereas the same output under the contract's exclusion-and-coverage accounting reports F1 0.615 at 1.3% coverage, which says the model is unusable here rather than merely weak.

For the operational class, structured output cannot prevent every failure, so the contract excludes the pair. Such failures were essentially absent in our own runs on a reliable hosted model (under 0.1% over 4,000 pairs), too few to measure the accounting effect, so we injected them: a controlled injection turns a fraction f of pairs into typed `MatchError` outcomes, as a refusal, a timeout, a rate limit, or an unusable response would. Two regimes separate the concerns. Under *uniform* failure on the dependency-free stack (45,880 decided pairs, 20 seeds per rate, operating point precision 0.855, recall 0.972), the excluded F1 stays flat at 0.910 from $f = 0$ to $f = 0.20$ while the silent F1 falls to 0.813, understating by 9.7 points a matcher whose decisions never changed. Under failure *concentrated on the hardest pairs* (the lowest-similarity fifth of a

Table 2: Natural (non-injected) failure behaviour by model capability on DBLP–ACM, for the free-text convention under three accountings: "exact" (pyJedAI's literal rule), "silent" (unparseable folded into non-match), and "excl." (unparseable excluded, coverage reported). A response is non-canonical if it is not literally `True`, and unparseable if no verdict survives a lenient parse. The hosted row is 200 pairs, each local row 1,000. These F1 columns describe only the free-text alternative.

Model	non-canon.	unparse.	F1 exact	F1 silent	F1 excl. (cov.)
gemini-2.5-flash-lite	37%	~0%	0.879	0.880	0.880 (100%)
qwen2.5:0.5b	100%	0%	0.000	0.297	0.297 (100%)
llama3.2:1b	100%	99%	0.000	0.044	0.615 (1.3%)

200-pair sample, replayed over real `gemini-2.5-flash` verdicts), the silent F1 falls from 0.877 to 0.638 and the excluded F1 also moves, to 0.811. Exclusion therefore does not recover an unbiased estimate when abstention is non-random: it is a selective prediction whose reported quality is conditional on the covered subset [10]. The contract makes the condition visible, since coverage falls to 80% in both regimes and `n_errors` is reported beside the metric. A metric at low coverage should be read as an estimate over the decided subset only. Evaluating the tool against its own requirements also exposed gaps, including a violation of our own extensibility rule (Section 6).

5 Related Work

The introduction positioned Denselinkage against the ER tool landscape; this section positions its architecture within software engineering for AI-enabled systems, where an ML component's implicit failure modes and dependencies become hidden maintenance cost [26]. The ingredients used here are established, including the reject option of selective prediction [6, 10]; their composition into a shipped contract is not. Selective prediction is normally a property of a classifier and its evaluation protocol, exercised offline by whoever trained the model. Here abstention is a value in the public type of a library, produced by a component the library does not own and cannot retrain, and propagated into the pairwise metrics rather than silently absorbed by them. That is the difference from the other libraries of Table 1, whose APIs and result objects have no representation for the outcome at all. Reference architectures [1] are usually consumed descriptively or to evaluate existing systems; Resolvi [18] was used here as a source of requirements and a conformance check, and to our knowledge this is the first implementation to operationalise it. API design and usability research is adopted here as requirements rather than retrospective critique [3, 17, 28, 29]; studies of API evolution quantify the cost of breaking changes [24, 35] that the frozen contract addresses under semantic versioning [22], and architectural fitness functions [7], usually retrofitted to stop erosion, here guard invariants from before the first release.

6 Limitations, Roadmap, and Conclusion

Vector search is exact (flat) in both backends. Index persistence covers only the numpy stack and dispatches on concrete classes, so a port-conforming third-party index cannot be persisted, a violation of our own extensibility rule scheduled for a `Persistable` port. The `Filter` port ships an adapter but no orchestrated path, the LLM path is synchronous with fixed-interval retries, the embedder's batching parameters are not yet threaded through, and `Trainer` is contract-only. The near-term roadmap is backward-compatible additions behind existing ports; the larger steps follow the reserved surface (approximate-nearest-neighbour adapters, an asynchronous matcher, and trainers).

Some threats to validity qualify the evaluation. Several measures are necessary conditions rather than direct ones: that `mypy` accepts fifteen adapters shows the ports are implementable, not convenient. The most serious threat is circularity. One team set the requirements, built the contract and all fifteen adapters, and ran the evaluation, and two of the five quality requirements were derived from the same tool-landscape gaps the evaluation later scores against. This is the standard hazard of design-science research, where the artifact and the knowledge claim are produced together and the evaluation must be argued rather than assumed [12, 33]. Three countermeasures apply, none of which removes it: every requirement is traced to a named external source (Section 2.1), so a reader can judge whether it was chosen to be easy to satisfy; the validating claims rest on evidence the design did not place, namely the version-control history, the CI fitness function, and the failure runs; and the extensibility evidence comes from parties outside the team. The descriptive checks should therefore be read as design rationale rather than as independent validation. The QR1 probe's LLM agent shares the model family that assisted development, so the three developers are weighted more heavily. External validity is narrow: one library, one team, and one dataset (DBLP–ACM), with the operational half of the failure analysis resting on injection rather than on natural failures, so no result here characterises ER libraries or LLM matchers in general. A second-domain benchmark (Abt–Buy), natural failures at serving scale, and a user-centred study would begin to address this.

Three lessons stand out as conjectures from this single case. The design decisions with the largest effect concerned state and failure handling, not algorithms. Static checks cannot tell whether a contract is implementable without violating its invariants, since the index ports type-checked yet hid a state-ownership defect that only an implementation attempt revealed, so a small vertical slice is a low-cost check before freezing an API. And a library wrapping a component that can decline to answer owes its users a place to put that outcome: once abstention has no type, it is absorbed by whatever the metric counts by default, and the quality report stops distinguishing a weak model from an absent one. That holds wherever a metric is computed over decisions an LLM was asked but not guaranteed to make.

Artifact Availability

Denselinkage is openly available under the MIT license, on PyPI (`pip install denselinkage`) and on GitHub (https://github.com/caalvaro/denselinkage); the library figures refer to the `v1.0.0` tag. The repository includes the source, the decision log and architecture decision records, the Resolvi conformance record, the example programs, the test suite including the dependency-cut test, and the reproducibility package: the scripts behind the end-to-end run,

the failure measurement, and the injection sweeps; and the implementability probe with its handout, submissions, and harnesses. The real `gemini-2.5-flash` verdicts behind the end-to-end run are cached, so its injection sweep replays without an API key.

Acknowledgements

We thank the three developers who took part in the extensibility probe. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001, and by the Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq).

In accordance with the ISE 2026 policy on generative AI, we disclose the use of Claude (Anthropic) as a coding and writing assistant: to draft and copy-edit portions of this manuscript, to write and run the evaluation scripts, and to assist parts of the implementation. All technical claims, architecture decisions and empirical results were defined, verified against the source code and primary sources, and approved by the authors. The same model family also appears as an *object of study*: the LLM agent in the QR1 probe is a disclosed use of AI as an experimental subject, distinct from authoring. AI is not an author.

References

[1] Samuil Angelov, Paul Grefen, and Danny Greefhorst. 2012. A framework for analysis and design of software reference architectures. *Information and Software Technology* 54, 4 (2012), 417–431. doi:10.1016/j.infsof.2011.11.009

[2] Amit Bagga and Breck Baldwin. 1998. Algorithms for Scoring Coreference Chains. In *Proc. LREC 1998, Linguistic Coreference Workshop*. 563–566.

[3] Joshua Bloch. 2006. How to Design a Good API and Why It Matters. In *OOPSLA '06 Companion*. 506–507. doi:10.1145/1176617.1176622

[4] Peter Christen. 2012. *Data Matching: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection*. Springer. doi:10.1007/978-3-642-31164-2

[5] Alistair Cockburn. 2005. Hexagonal Architecture (Ports and Adapters). https://alistair.cockburn.us/hexagonal-architecture/.

[6] Ran El-Yaniv and Yair Wiener. 2010. On the Foundations of Noise-free Selective Classification. *Journal of Machine Learning Research* 11 (2010), 1605–1641.

[7] Neal Ford, Rebecca Parsons, and Patrick Kua. 2017. *Building Evolutionary Architectures: Support Constant Change* (1 ed.). O'Reilly.

[8] Silvery D. Fu, David Wang, Wen Zhang, and Kathleen Ge. 2024. Liberal Entity Matching as a Compound AI Toolchain. arXiv preprint arXiv:2406.11255.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[10] Yonatan Geifman and Ran El-Yaniv. 2017. Selective Classification for Deep Neural Networks. In *Advances in Neural Information Processing Systems (NeurIPS)*.

[11] Saibo Geng, Hudson Cooper, Michał Moskal, Samuel Jenkins, Julian Berman, Nathan Ranchin, Robert West, Eric Horvitz, and Harsha Nori. 2025. JSONSchemaBench: A Rigorous Benchmark of Structured Outputs for Language Models. *arXiv preprint arXiv:2501.10868* (2025).

[12] Alan R. Hevner, Salvatore T. March, Jinsoo Park, and Sudha Ram. 2004. Design Science in Information Systems Research. *MIS Quarterly* 28, 1 (2004), 75–105. doi:10.2307/25148625

[13] Ekaterini Ioannou, Konstantinos Nikoletos, and George Papadakis. 2025. pyJedAI: A Library with Resolution-Related Structures and Procedures for Products. *INFORMS Journal on Computing* 37, 3 (2025), 516–530. doi:10.1287/ijoc.2023.0410

[14] Yuliang Li, Jinfeng Li, Yoshihiko Suhara, AnHai Doan, and Wang-Chiew Tan. 2020. Deep Entity Matching with Pre-Trained Language Models. *Proceedings of the VLDB Endowment* 14, 1 (2020), 50–60. doi:10.14778/3421424.3421431

[15] Robin Linacre, Sam Lindsay, Theodore Manassis, Zoe Slade, and Tom Hepworth. 2022. Splink: Free software for probabilistic record linkage at scale. *International Journal of Population Data Science* 7, 3 (2022), 1794. doi:10.23889/ijpds.v7i3.1794

[16] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, et al. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD 2018*. 19–34. doi:10.1145/3183713.3196926

[17] Brad A. Myers and Jeffrey Stylos. 2016. Improving API Usability. *Commun. ACM* 59, 6 (2016), 62–69. doi:10.1145/2896587

[18] Andrei Olar. 2025. Resolvi: A Reference Architecture for Extensible, Scalable and Interoperable Entity Resolution. *arXiv preprint arXiv:2503.08087* (2025).

[19] George Papadakis, Dimitrios Skoutas, Emmanouil Thanos, and Themis Palpanas. 2021. Blocking and Filtering Techniques for Entity Resolution: A Survey. *Comput. Surveys* 53, 2 (2021), 31. doi:10.1145/3377455

[20] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, et al. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[21] Ralph Peeters, Aaron Steiner, and Christian Bizer. 2025. Entity Matching using Large Language Models. In *Proceedings of the 28th International Conference on Extending Database Technology (EDBT 2025)*. 529–541. doi:10.48786/edbt.2025.42

[22] Tom Preston-Werner. 2013. Semantic Versioning 2.0.0. https://semver.org/.

[23] Python Packaging Authority. 2020. pyproject.toml specification – [project.optional-dependencies] (extras). https://packaging.python.org/en/latest/specifications/pyproject-toml/.

[24] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158. doi:10.1016/j.jss.2016.04.008

[25] Irum Rauf, Elena Troubitsyna, and Ivan Porres. 2019. A systematic mapping study of API usability evaluation methods. *Computer Science Review* 33 (2019), 49–68. doi:10.1016/j.cosrev.2019.05.001

[26] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, et al. 2015. Hidden Technical Debt in Machine Learning Systems. In *NIPS 2015*. 2503–2511.

[27] Emma Harper Smith. 2017. PEP 561 – Distributing and Packaging Type Information. Python Enhancement Proposal. https://peps.python.org/pep-0561/.

[28] Jeffrey Stylos and Steven Clarke. 2007. Usability Implications of Requiring Parameters in Objects' Constructors. In *29th International Conference on Software Engineering (ICSE 2007)*. 529–539. doi:10.1109/ICSE.2007.92

[29] Jeffrey Stylos and Brad A. Myers. 2008. The Implications of Method Placement on API Learnability. In *16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-16)*. 105–112. doi:10.1145/1453101.1453117

[30] Saravanan Thirumuruganathan, Han Li, Nan Tang, Mourad Ouzzani, Yash Govind, Derek Paulsen, Glenn Fung, and AnHai Doan. 2021. Deep Learning for Blocking in Entity Matching: A Design Space Exploration. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2459–2472. doi:10.14778/3476249.3476294

[31] Tianshu Wang, Xiaoyang Chen, Hongyou Lin, Xuanang Chen, Xianpei Han, Le Sun, Hao Wang, and Zhenyu Zeng. 2025. Match, Compare, or Select? An Investigation of Large Language Models for Entity Matching. In *COLING 2025*. 96–109.

[32] Yuxin Wang, Yuhan Chen, Zeyu Li, Xueze Kang, Yuchu Fang, Yeju Zhou, Yang Zheng, Zhenheng Tang, Xin He, Rui Guo, Xin Wang, Qiang Wang, Amelie Chi Zhou, and Xiaowen Chu. 2024. BurstGPT: A Real-world Workload Dataset to Optimize LLM Serving Systems. *arXiv preprint arXiv:2401.17644* (2024).

[33] Roel J. Wieringa. 2014. *Design Science Methodology for Information Systems and Software Engineering*. Springer. doi:10.1007/978-3-662-43839-8

[34] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in Software Engineering*. Springer. doi:10.1007/978-3-642-29044-2

[35] Laerte Xavier, Aline Brito, André Hora, and Marco Tulio Valente. 2017. Historical and impact analysis of API breaking changes: A large-scale study. In *SANER 2017*. 138–147. doi:10.1109/SANER.2017.7884616

[36] Daoguang Zan, Bei Chen, Zeqi Lin, Bei Guan, Yongji Wang, and Jian-Guang Lou. 2022. When Language Model Meets Private Library. In *Findings of the Association for Computational Linguistics: EMNLP 2022*.

[37] Alexandros Zeakis, George Papadakis, Dimitrios Skoutas, and Manolis Koubarakis. 2023. Pre-trained Embeddings for Entity Resolution: An Experimental Analysis. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2225–2238. doi:10.14778/3598581.3598594