



Monitoring des Schweizerischen Handelsblattes (SHAB)

Automated classification and customer matching
for official Swiss commercial-register publications

Print:

11.06.2026

Team / Authors:

Martin Arendar and Matteo Mahler

Advisor:

Prof. Laurent Metzger

Co-advisor:

Prof. Dr. Remo M. Frey

Proofreader:

Prof. Dr. Thomas Bocek

Abstract

The Schweizerisches Handelsamtsblatt (SHAB) publishes legally relevant notices about Swiss companies and persons. For banks, these publications can indicate events that affect customer due diligence, but manual monitoring is impractical because of the daily publication volume and the need to identify customer-relevant changes quickly. This thesis develops a proof of concept for an in-house automated SHAB monitoring system aimed at increasing the reliability and performance of the current solution.

The implemented prototype combines deterministic publication processing with two machine learning components. A Spring Boot service orchestrates import, enrichment, persistence, audit trails, and the user-facing API. A separate FastAPI based inference service provides LLM-based multi-label classification for HR02 publication texts and probabilistic record linkage for customer matching. Classification combines structured SHAB attributes with an LLM using a constrained prompt and structured output. Customer matching uses separate person and company models based on Splink and DuckDB, allowing explainable candidate matches without requiring a labelled real customer dataset. A React user interface supports review of publications, classifications, matching explanations, and manual corrections.

On 130 manually labelled HR02 publications, the classifier reaches an F1 score of 0.86. It assigns the exact set of correct labels in 81.54% of cases, and captures all expected labels in 83.08% (improved from 26% and 33% respectively). In the synthetic customer matching evaluation, person matching reaches 0.93 accuracy and company matching 0.87, with match-class F1 scores of 0.96 and 0.92 respectively. Neither matching model produces false positives on the available negative examples, because the system favors recall.

Table of contents

I	Documentation	1
1	Introduction	1
1.1	Assignment	1
1.2	Conditions	1
1.3	State of the Art	1
1.3.1	LLM-Based Text Classification	2
1.3.2	Entity Matching	2
1.4	Context & Domain	2
1.4.1	Purpose and Content	2
1.4.2	HR02 Classification	3
1.4.3	Publication Behavior	3
1.4.4	Publication Volume and Distribution	3
1.4.5	Publication Language Distribution	4
2	Requirements	6
2.1	Functional Requirements	6
2.2	Non-Functional Requirements	9
3	Architecture	11
3.1	C4	11
3.1.1	Context diagram	11
3.1.2	Container diagram	12
3.1.3	Component diagram	12
3.2	Architectural Decisions	15
3.2.1	Runtime Separation for AI and Business Logic	15
3.2.2	Jobs vs. Message Queue	15
3.2.3	Job Scheduling Approach	16
3.2.4	Containerization Strategy	17
3.2.5	Probabilistic Record Linkage Library	17
3.2.6	DuckDB	18
3.2.7	Linker Loading Strategy	18
3.2.8	Shared Matching Model Package	19
3.2.9	Strategy for Publication Format Handling	19
3.2.10	International Data Format Handling	20
3.3	Data Model	21
3.3.1	SHAB Monitoring Database Schema	21
3.3.2	Inference API Database Schema	22
3.4	Job Processing Model	23
3.4.1	Job lifecycle & states	23
3.4.2	Idempotency	23
3.4.3	Audit Trail	24
3.5	Machine Learning Architecture	25
3.5.1	Classification Architecture	25
3.5.2	Customer Matching Architecture	25
4	Implementation	27
4.1	Publication Processing Pipeline	27
4.1.1	Workflow Overview	27
4.1.2	Classification	30
4.1.3	Customer Matching	31
4.2	SHAB Monitoring (Spring Boot)	32
4.2.1	Package structure	32
4.2.2	SHAB API Client	33
4.2.3	Inference API Client	33
4.2.4	Sub-rubric Handlers	35
4.2.5	Persistence Layer	35

4.2.6	REST API	35
4.2.7	Notification System	36
4.3	DB Sync Tool	37
4.3.1	Step 1: Entity Extraction	37
4.3.2	Step 2: Customer Seeding	37
4.3.3	Step 3: DuckDB Creation	37
4.3.4	Available datasets	37
4.4	Splink Shared Package	38
4.4.1	Package Structure	38
4.4.2	Preprocessing Function	38
4.4.3	Model Configuration	39
4.5	Matching Model Training Tool	40
4.5.1	Input and Configuration	40
4.5.2	Training Process	40
4.5.3	Output	40
4.6	Inference API (FastAPI)	41
4.6.1	Package Structure	41
4.6.2	HTTP API	41
4.6.3	Classification Service	42
4.6.4	Matching Service	42
4.7	User Interface	44
4.7.1	Purpose and Scope	44
4.7.2	Technologies Used	44
4.7.3	Views	44
4.7.4	Design	46
4.7.5	Limitations	46
4.8	Deployment	48
4.8.1	CI/CD	48
4.8.2	Services	48
4.8.3	Routing	48
4.8.4	Configuration	48
4.9	Testing	49
4.9.1	SHAB Monitoring Testing	49
4.9.2	UI Testing	49
4.9.3	Inference API Testing	52
5	Machine Learning Evaluation	53
5.1	Classification Evaluation	53
5.1.1	Preparation	53
5.1.2	Processing	53
5.1.3	Results	54
5.1.4	Classification Error Distribution	55
5.2	Synthetic Customer Dataset	57
5.3	Customer Matching Evaluation	57
5.3.1	Threshold Sensitivity	58
5.3.2	False Negative Analysis	59
5.4	Cost Analysis	59
5.4.1	OpenAI API Costs	59
5.4.2	Estimated Production Cost	59
6	Results	60
6.1	Implemented System	60
6.2	Classification Results	60
6.3	Customer Matching Results	60
6.4	PoC Boundaries	60
7	Conclusions & Outlook	61
7.1	Limitations	61
7.2	Outlook	61
7.2.1	User Management	61

7.2.2	Audit Log	61
7.2.3	Confirm Publications for continuous Dataset improvement	62
7.2.4	Customer Matching	62
7.2.5	Classification	62
7.2.6	Production Setup	62
II	Glossary and List of Abbreviations	63
III	Bibliography	64
IV	List of Illustrations	67
V	List of Tables	69
VI	Code-Listings	71
VII	Appendix	72
8	SHAB Sub-rubrics	72
9	HR02 Classification Codes	73
10	List of aids	74

Chapter I

Documentation

1 Introduction

This chapter provides the context for the thesis by introducing the environment in which the project was carried out, the initial assignment, and the relevant domain conditions. It establishes the background needed to understand the problem space, the stakeholders involved, and the constraints that shaped the solution developed in the following chapters.

1.1 Assignment

The goal of this thesis is to provide a proof of concept for an automated “Schweizerisches Handelsblatt” (SHAB) [\[1\]](#) monitoring solution in the context of a Swiss bank’s customer due diligence processes. SHAB publications contain legally relevant information about companies and persons, which is currently provided to our industry partner by a third-party provider. This creates several disadvantages: delayed availability because processing happens overnight, recurring provider costs, dependency on the provider’s availability, and data quality that is not always satisfactory. The assignment therefore aims to connect SHAB publications directly to the bank’s systems, replacing the external solution while improving the quality and actuality of the available data.

Manually reviewing the daily publication volume and identifying customer-relevant events is not feasible. Hence they need to be analyzed programmatically. To achieve this, this thesis documents the creation of a system that automatically retrieves SHAB publications, stores the relevant publication data, classifies publications into event types, and matches involved entities against an internal customer database.

The main deliverable is a working prototype. The system should support automated processing while still keeping human review possible through explanations, manual corrections, and transparent processing states in a user interface. In addition to the software prototype, the thesis documents the architectural decisions, implementation approach, evaluation results, and known limitations of the solution.

The prototype is not intended to replace the bank’s operational processes directly, but to demonstrate whether the monitoring workflow can be supported effectively with a combination of deterministic processing, LLM-based classification, and probabilistic customer matching. The solution should aim for the most complete coverage possible within the project scope and remain technologically compatible with the bank’s existing setup.

1.2 Conditions

This work was carried out as a bachelor thesis. The allocated workload is 12 ECTS credits per person. With the standard conversion of 30 hours per ECTS credit, this corresponds to a time budget of 360 hours per person.

1.3 State of the Art

Automated monitoring of official publications combines two well established information processing tasks: text classification and entity matching. Text classification determines which business event a publication describes, while entity matching determines whether a person or company mentioned in the publication corresponds to an existing customer record. Both tasks have their traditions, but the constraints of this assignment differ from regular benchmarks: Regarding the classification the input texts are legal and administrative notices, the labels are domain specific, publications may contain multiple relevant events. For the customer matching no production customer dataset with manually labelled matches was available for model training due to confidentiality. Its contribution is the integration and evaluation of current techniques in a realistic SHAB monitoring workflow.

1.3.1 LLM-Based Text Classification

Text classification has traditionally been implemented with supervised machine learning. Earlier approaches relied on manually engineered features or statistical representations of words, while modern systems commonly use pre-trained language models. For tasks with sufficient labelled data, models such as BERT can be fine-tuned for a specific classification task [2]. Large language models also support instruction-based classification, where the task is described in natural language and the model is prompted to return labels without task-specific training [3], [4].

In this project, no model fine tuning was performed. Instead, the classifier uses an existing LLM through a constrained prompt and structured response format [5], [6].

1.3.2 Entity Matching

Entity matching (also known as record linkage or entity resolution) addresses the problem of deciding whether two records describe the same real-world entity. In customer due diligence scenarios, exact matching is usually insufficient because names, addresses, and company identifiers may be incomplete, formatted differently, abbreviated, outdated, or misspelled. The classical statistical foundation is the Fellegi-Sunter model, which combines agreement and disagreement signals from multiple fields into a probabilistic decision about whether two records match [7]. This model remains influential because it is interpretable. Each compared field contributes evidence, and the final score can be inspected instead of being treated as a black box.

Modern entity matching research also includes systems built around supervised machine learning and deep learning. The Magellan project framed entity matching as an end-to-end data science workflow involving blocking, feature generation, model training, debugging, and human involvement [8]. DeepMatcher later showed how neural architectures can learn matching functions over structured records [9]. These approaches can reach high quality when representative labelled training pairs are available, which was not the case in this project. Recent work investigates LLMs for entity matching as well. LLM-based matchers can compare record descriptions from natural language prompts and may need no or only few task-specific examples to perform competitively with fine-tuned pre-trained language models [10]. Further research studies how LLMs can reason over pairwise matches or select candidates from sets of records, but also highlights cost, prompt sensitivity, and efficiency issues [11]. These drawbacks were not acceptable for this project, because the handling of the daily publication volume is time sensitive. In the context of banking data, these concerns are amplified by privacy requirements and the need to control which customer attributes leave the system boundary.

1.4 Context & Domain

This section describes the domain in which the SHAB monitoring prototype operates and the external data source it depends on. It focuses on the publication characteristics, data structures, and business relevance that shaped the system's processing and enrichment workflow.

The SHAB is the official gazette of the Swiss Confederation and the only publication channel through which a defined set of legal notices become binding on third parties [12]. It has appeared continuously since 1883 and is published five times per week, on business days from Monday to Friday.

1.4.1 Purpose and Content

The content is organized into several categories identified by specific rubric labels. The most relevant for this work include:

- **HR02 (Commercial Register):** entries about Swiss companies, including new registrations, mutations (e.g., changes in management, share capital, or address), and deletions.
- **KK (Insolvency/Bankruptcy):** bankruptcy notices and related proceedings.
- **LS (Lost Securities):** declarations of nullity for lost or destroyed securities.
- **BH (Commercial Register Ordinance Notices):** calls to remedy organizational deficiencies under the Commercial Register Ordinance.
- **NA (Notary/Public Notice):** official notices related to notary offices.
- **SR (Debt Enforcement):** notices regarding debt collection proceedings.
- **SB (Debt Enforcement Office Notices):** debt enforcement office notices, including real estate auctions, bankruptcy warnings, and auction announcements.
- **ES (Estate Notices):** estate-related notices, including will openings, calls for heirs, and creditor calls for public inventories.

The complete list of all available categories can be found in [Tab. 48](#).

Across all categories, between 1'200 and 1'500 individual publications are released on each publication day [\[12\]](#). For a bank, these notices represent events that may affect existing customer relationships, for example a customer company entering insolvency, changing its legal representatives, or being deregistered. Monitoring SHAB is therefore a relevant input for customer due diligence and risk management.

Publications are available through the SHAB API [\[13\]](#), which the system uses to retrieve new entries. The API provides structured records containing metadata (publication date, rubric, sub-rubric, involved entities) and the textual content of the notice.

1.4.2 HR02 Classification

In the context of this thesis the HR02 sub-rubric is special. The industry partner has a use case for categories inside the HR02 sub-rubric.

The structured attributes cannot be treated as a complete or authoritative classification of the publication's content. In practice, some publications omit attributes that would apply, meaning the structured fields underrepresent the actual events described in the text. For example, one publication formats its address as a custom string rather than using the structured address fields, and sets the attribute `noUID=true` despite the text containing a CHE-XXX.XXX.XXX number [\[14\]](#). Another publication has `others=false` (which means that all changes are covered by attributes) but describes events in the text that are not reflected in any structured attribute [\[15\]](#). The observed pattern is that attributes are more likely to be missing than to be incorrectly set, so the structured data tends to underclassify rather than overclassify.

1.4.3 Publication Behavior

Since the system's polling strategy depends on when publications appear and how volume varies across the week, a small monitoring tool was built that polls the SHAB API every 15 minutes and records changes in the number of publications available per day. The following patterns are based on a sample observation period from 12.05.2026 to 02.06.2026.

Publications appear in a single batch at midnight UTC. For every day observed, the publication count moved from zero to its full daily value within one to two seconds after 00:00:00 UTC. This indicates that SHAB releases the day's publications as a single atomic batch at the start of the day.

Publication volume varies significantly by day of week and is reduced on holidays. The observed pattern differs from the documented five-day schedule:

Day	Observed count	Notes
Mon - Fri	1'590 - 2'820	Standard business days
Saturday	15 - 65	Reduced but non-zero
Sunday	0 - 2	Effectively no publications
Public holidays	28 - 131	Varies by holiday (e.g. Ascension Day 131, Whit Monday 28)

Tab. 1: SHAB publication volume by day of week (observation period: 12.05.2026 to 02.06.2026)

Although SECO documents a five-day publication schedule, Saturdays were not empty in any observed week. The polling strategy therefore cannot exclude any calendar day. The observed weekday volume of 1'590-2'820 also exceeds the 1'200-1'500 figure cited in the official documentation [\[12\]](#), suggesting the system has grown since that figure was last updated, which needs to be kept in mind during the implementation.

Publications can be retracted after release. On several days, the publication count decreased after its initial value, sometimes days after the original release. For example, three publications from 15.05.2026 were retracted across 18.05.2026 and 19.05.2026. Published records are therefore not strictly immutable.

1.4.4 Publication Volume and Distribution

The figures below are based on publications imported by the system from the 05.01.2026 until the 27.02.2026, covering only the sub-rubrics ([Tab. 49](#)) relevant to this project.

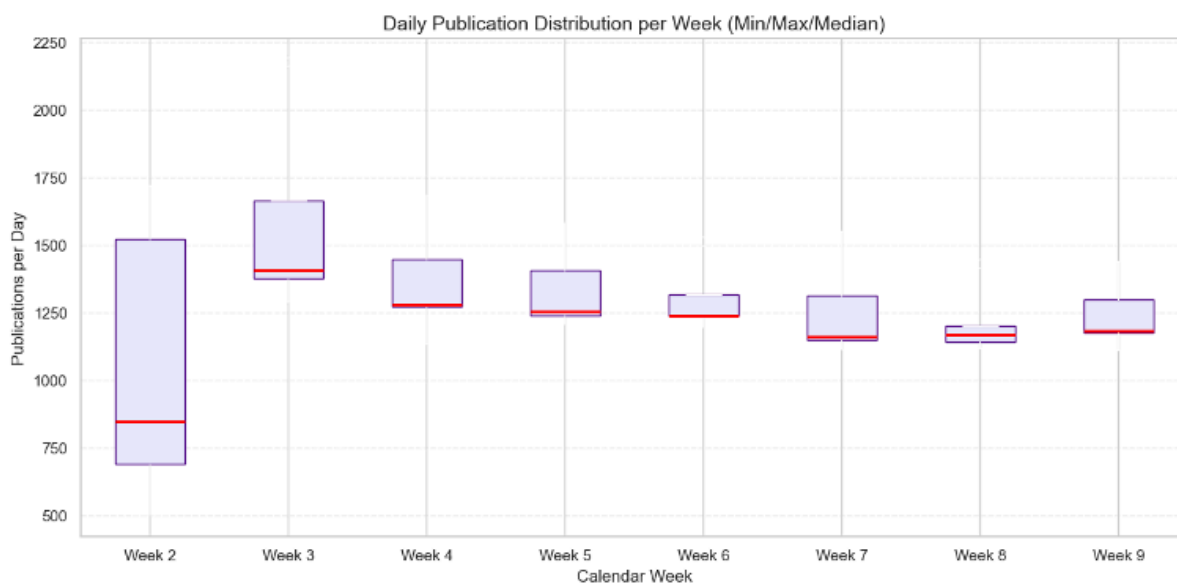


Figure 1: Daily publication distribution per calendar week 2026 (Min/Max/Median)

Figure 1 shows the daily publication count per calendar week for the relevant sub-rubrics. The median daily volume is consistently between 1'100 and 1'500 publications, with week-to-week variation driven primarily by weekday composition and public holidays. Week 2 shows the highest spread, reflecting the inclusion of lower-volume days.

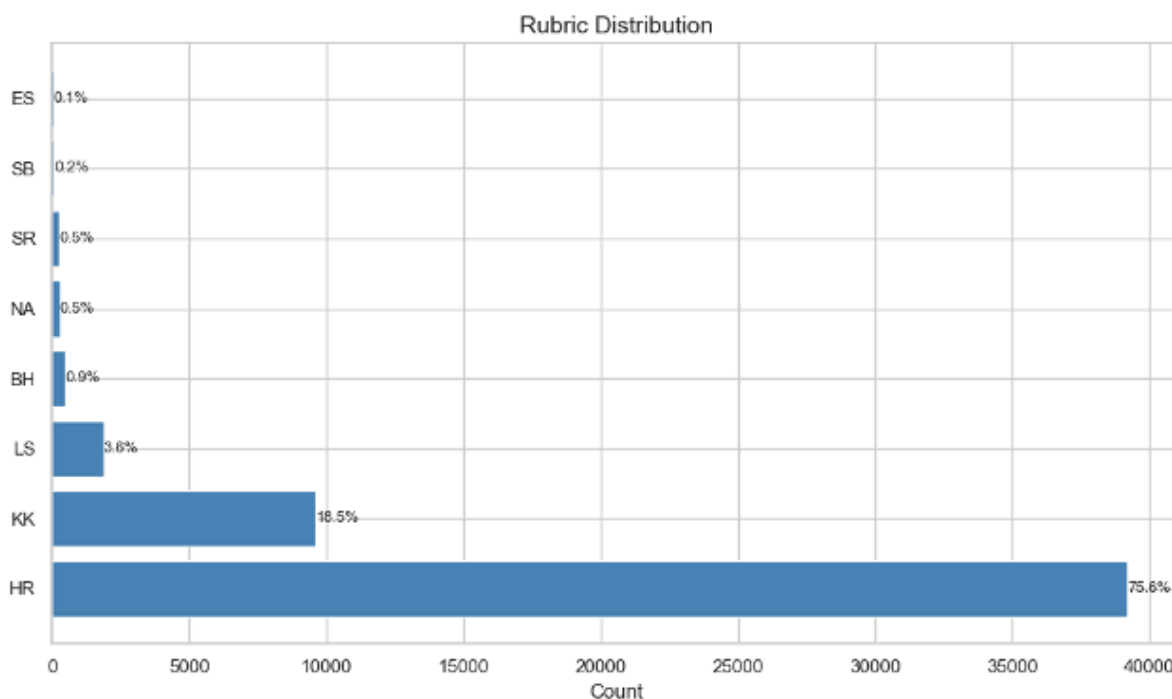


Figure 2: Rubric distribution across imported publications

Figure 2 shows the distribution across sub-rubrics. HR dominates at 75.6% of all imported publications, followed by KK at 18.5% and LS at 3.6%. The remaining sub-rubrics (BH, NA, SR, SB, ES) together account for less than 2.5%.

1.4.5 Publication Language Distribution

SHAB covers all Swiss cantons and thereby all official national languages (DE, FR, IT).

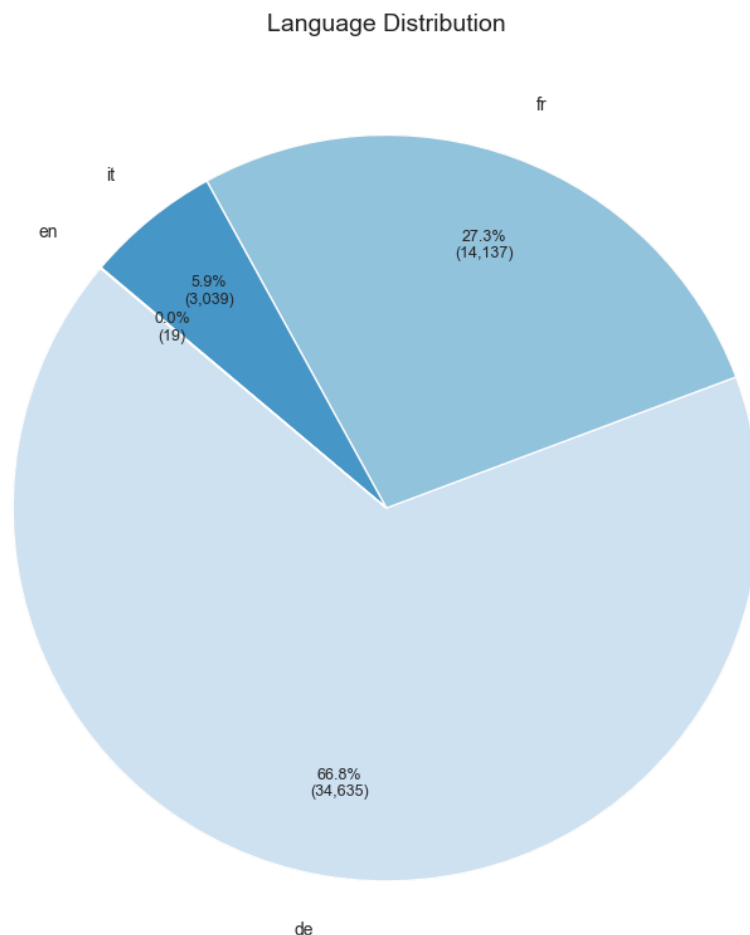


Figure 3: Language distribution across imported publications

Figure 3 shows the distribution of languages across imported publications in the analyzed time frame.

2 Requirements

2.1 Functional Requirements

The following functional requirements define the specific behaviors and capabilities that the system must exhibit to meet the needs of its users and stakeholders. The results are added at the end of the implementation phase, after testing is completed.

ID	FR-1
Name	Persist Publication Data
Description	The system stores the original publication data as well as the extracted data fields persistently and in a uniquely referenceable manner for each publication.
Result	Fulfilled Data Model 3.3

Tab. 2: FR-1: Persistent storage of original and extracted publication data

ID	FR-2
Name	Record and Display Ingestion Timestamp
Description	The system records the ingestion timestamp for each publication and displays it within the publication view.
Result	Fulfilled Audit Trail 3.4.3 , User Interface 4.7

Tab. 3: FR-2: Traceability of publication ingestion time

ID	FR-3
Name	Expose Ingestion Status and Errors
Description	The system logs the ingestion status for each publication (pending, in progress, successful or failed) and provides administrators with an overview of failed ingestion processes.
Result	Fulfilled, though states have changed to JobRunr default states: Schedules, Enqueued, Processing, Succeeded, Failed Job lifecycle & states 3.4.1 .

Tab. 4: FR-3: Transparency of ingestion status and error cases

ID	FR-4
Name	Manually Re-Trigger Ingestion
Description	The system enables administrators to manually re-trigger the ingestion process for a publication.
Result	Fulfilled Manual triggering 3.4.2.2

Tab. 5: FR-4: Manual re-execution of ingestion processes

ID	FR-5
Name	Automatically Classify Publications
Description	Publications with the sub-rubric HR02 are automatically classified by the system, based on publication text and/or attributes, to provide more detailed information. The relevant classification codes are listed in Tab. 48 .
Result	Fulfilled Classification 4.1.2

Tab. 6: FR-5: Automated publication classification

ID	FR-6
Name	Provide Classification Explanation
Description	The system provides a comprehensible explanation for each classification decision (relevant text passages or signals).
Result	Fulfilled Persistence and Traceability 4.1.2.3 , User Interface 4.7

Tab. 7: FR-6: Explainability of classification decisions

ID	FR-7
Name	Allow Manual Classification Override
Description	Users can manually override the system's classification of the sub-rubric. The system stores the manual adjustment as the valid result.
Result	Fulfilled <u>User Interface 4.7</u>

Tab. 8: FR-7: Manual correction of classification results

ID	FR-8
Name	Automatically Perform Customer Matching
Description	The system automatically assigns publications to customers (customer matching) and stores the matching result for each publication.
Result	Fulfilled <u>Customer Matching 4.1.3</u>

Tab. 9: FR-8: Automated customer assignment process

ID	FR-9
Name	Provide Matching Explanation
Description	The system provides an explanation for each matching result, which can be accessed by administrators per publication.
Result	Fulfilled <u>User Interface 4.7</u>

Tab. 10: FR-9: Explainability of customer matching decisions

ID	FR-10
Name	Identify and List Unmatched Publications
Description	The system detects publications with no assigned customer or with multiple candidate matches and provides a list of these publications.
Result	Fulfilled <u>User Interface 4.7</u>

Tab. 11: FR-10: Visibility of unmatched/ overmatched publications

ID	FR-11
Name	Store and Display Confidence Score
Description	The system calculates a confidence score (between 0 and 100%) for each customer match and makes it accessible.
Result	Fulfilled <u>User Interface 4.7</u>

Tab. 12: FR-11: Availability of matching confidence scores

ID	FR-12
Name	Allow Manual Customer Assignment
Description	The system enables users to manually match a customer if automatic matching fails or is incorrect.
Result	Fulfilled <u>User Interface 4.7</u>

Tab. 13: FR-12: Manual customer assignment capability

ID	FR-13
Name	Notify Users About Required Actions
Description	A notification is sent to a statically configurable email address and informs about the results of a customer case. The email content includes the classification code, the matched customer and a link to the original SHAB message.
Result	Fulfilled <u>Notification System 4.2.7</u>

Tab. 14: FR-13: User notification of required customer-related actions

ID	FR-14
Name	Maintain Audit Trail for Classification and Matching
Description	The system maintains an audit trail for classification and matching decisions including timestamp, result, score(s), and responsible source or trigger (system or manual), and makes it accessible.
Result	Fulfilled <u>Audit Trail 3.4.3</u> , <u>User Interface 4.7</u>

Tab. 15: FR-14: Auditability of automated and manual decisions

ID	FR-15
Name	Export Quality Metrics
Description	The system enables users to export quality metrics and relevant key performance indicators in CSV.
Result	N/A. Metrics are accessible during model training and evaluation <u>Results 6</u>

Tab. 16: FR-15: Export capability for quality and performance metrics

ID	FR-16
Name	Filter Historical Decisions
Description	The system enables administrators to filter historical classification and matching decisions by date and outcome.
Result	Fulfilled <u>User Interface 4.7</u>

Tab. 17: FR-16: Filtering of historical decision records

2.2 Non-Functional Requirements

The following non-functional requirements define the quality attributes and constraints that the system must adhere to. Each non-functional requirements has defined landing zones in which the final results can be categorized. The testing procedure for each non-functional requirement is described here, and the final results are added at the end of the implementation phase, after testing is completed.

ID	NFR-1	
Name	Publication processing	
Category	Performance	
Description	The system can fully process a publication within the defined time brackets.	
Landing zones	• Minimum: 3 min • Target: 1 min • Outstanding: 15s	
Testing	Measure the time needed from starting the processing until it is fully completed.	
Result	Fulfilled - Outstanding (avg. 3s per publication)	

Tab. 18: NFR-1: Performance - Publication processing

ID	NFR-2	
Name	Publication classification accuracy	
Category	Reliability	
Description	The system has a classification accuracy within the defined percentage brackets.	
Landing zones	• Minimum: 80% • Target: 95% • Outstanding: 98%	
Testing	Compare the system's classification results with manually classified publications to calculate the accuracy.	
Result	Fulfilled - Minimum (Section 6.2)	

Tab. 19: NFR-2: Reliability - Publication classification accuracy

ID	NFR-3	
Name	Customer matching coverage	
Category	Reliability	
Description	The system covers the expected classification codes for HR02 publications within the defined percentage brackets.	
Landing zones	• Minimum: 80% • Target: 95% • Outstanding: 98%	
Testing	Compare the systems predicted classification code set with manually classified publications. A publication is counted as covered when all expected codes are included in the predicted code set, even if additional codes are returned.	
Result	Person: Fulfilled - Target; Company: Fulfilled - Minimum (Section 6.3)	

Tab. 20: NFR-3: Reliability - Customer matching accuracy

ID	NFR-4	
Name	System recovery	
Category	Reliability	
Description	The system is able to recover after a shutdown without losing any data and without the need for manual intervention.	
Landing zones	System recovers without any data loss and automatically resumes processing.	
Testing	Simulate a system shutdown during processing and verify that the system continues processing without errors.	
Result	Fulfilled, system can be stopped and restarted without data loss	

Tab. 21: NFR-4: Reliability - System recovery

ID	NFR-5	
Name	Code quality measures	
Category	Maintainability	
Description	To allow further development, Java will be used for the core functionality.	
Landing zones	• Minimum: Core functionality is written in Java • Target: "Minimum" + testing code coverage of 95% • Outstanding: "Target" + code is formatted uniformly across the whole project	
Testing	Verify the quality measures are applied.	
Result	Fulfilled - Outstanding (Section 4.9)	

Tab. 22: NFR-5: Maintainability - Code quality measures

3 Architecture

This section documents the architectural design of the application and its machine learning parts. Required architectural decisions are also documented here.

3.1 C4

Diagrams follow the C4 model [16] and are created using IcePanel [17]. The Code view is omitted; such low-level diagrams diverge quickly from the actual code and add little beyond reading the code directly.

3.1.1 Context diagram

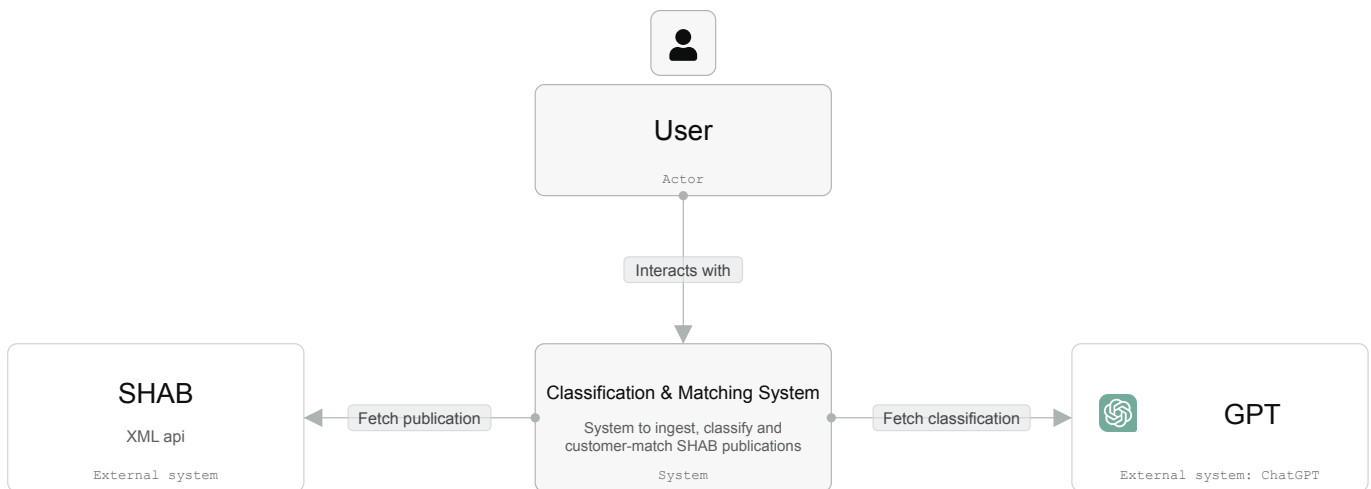


Figure 4: C4 - Context Diagram

The context diagram ([Figure 4](#)) shows the user interacting with the *Classification & Matching System* and the two external systems it depends on: the SHAB API for publication data and OpenAI for classification. An additional system, the SMTP mailserver, which sends out email notifications is omitted on this level but included in ([Figure 5](#)).

3.1.2 Container diagram

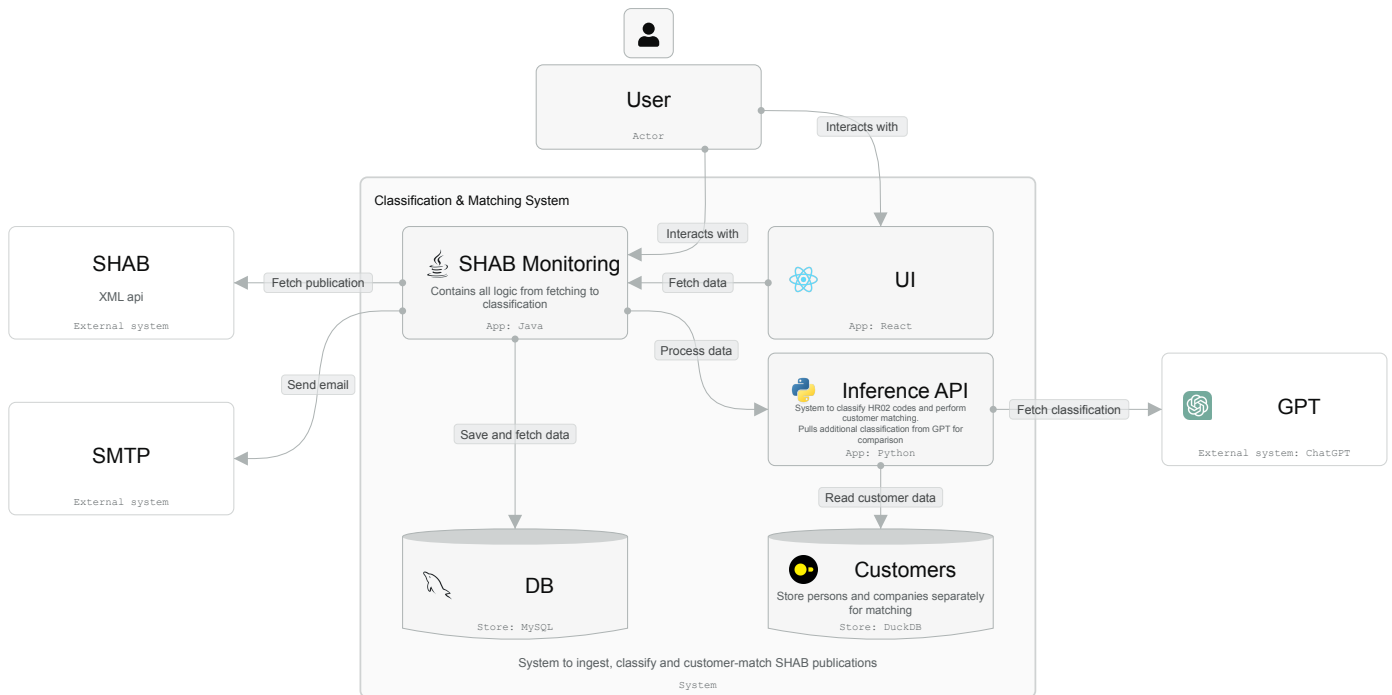


Figure 5: C4 - Container Diagram

The container diagram ([Figure 5](#)) shows the three applications: *SHAB Monitoring* (Spring Boot), the *Inference API* (FastAPI), and the *UI* (React). *SHAB Monitoring* fetches and processes publications from the SHAB API; during processing it invokes the *Inference API* for classification and entity matching. The *Inference API* groups all AI-related logic: it proxies classification requests to OpenAI and exposes an endpoint for entity matching. The *UI* provides an interface for bank employees to review processing results and perform follow-up actions.

3.1.3 Component diagram

The following subsections cover the internal structure of each application.

3.1.3.1 SHAB Monitoring

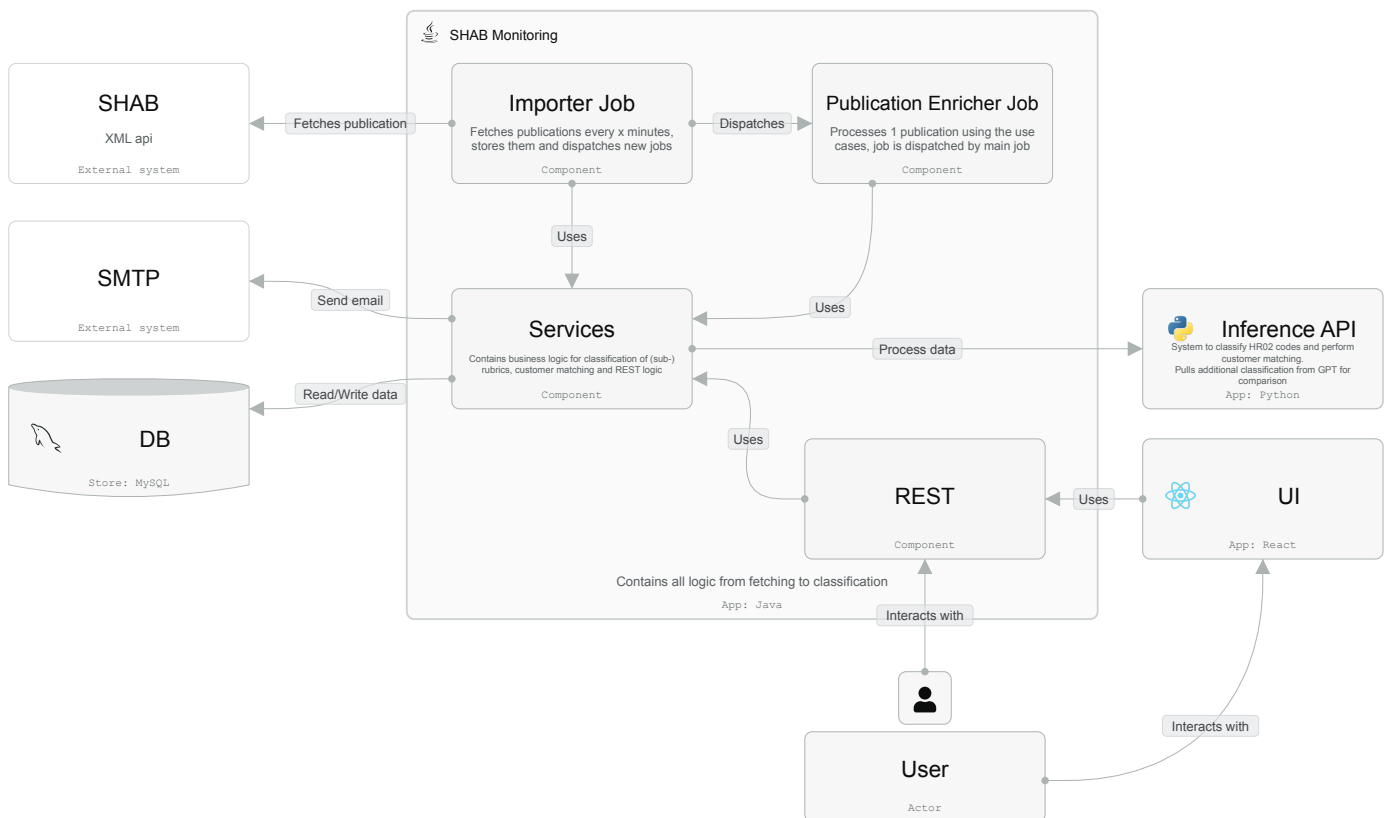


Figure 6: C4 - Component Diagram - SHAB Monitoring

The component diagram (Figure 6) shows the internal structure of *SHAB Monitoring*: it fetches data from the SHAB API, processes it using the *Inference API*, and stores results in MySQL. Users are notified by email and can access results through the *UI* and API.

3.1.3.2 Inference API

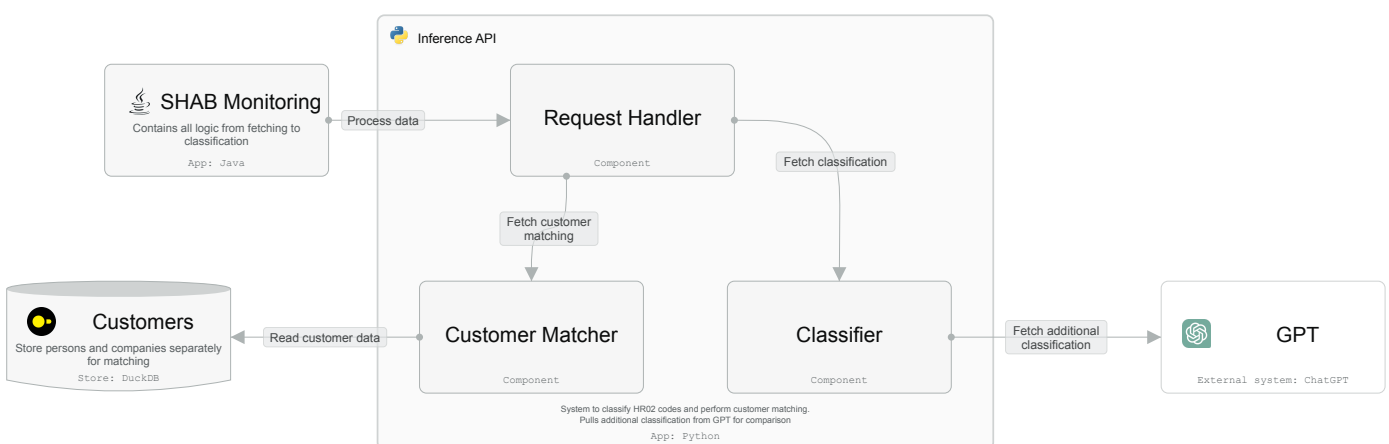


Figure 7: C4 - Component Diagram - Inference API

The *Inference API* (Figure 7) exposes endpoints for classification and matching requests from *SHAB Monitoring*. Classification requests are forwarded to the OpenAI API; entity matching runs locally against the DuckDB customer dataset.

3.1.3.3 User Interface

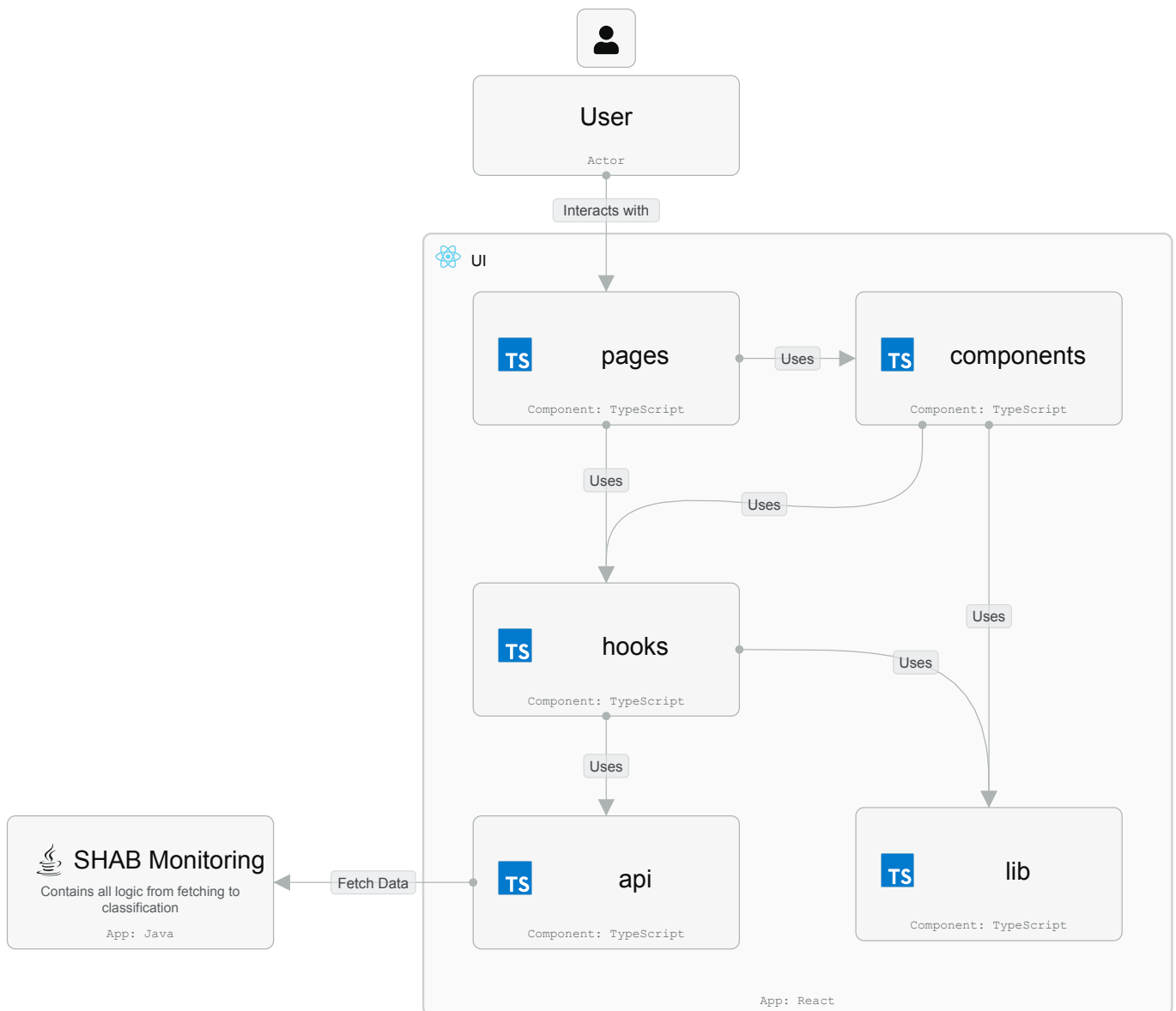


Figure 8: C4 - Component Diagram - User Interface

The *UI* ([Figure 8](#)) is structured around two pages: a list view for browsing publications and a detail view for inspecting results, managing classifications, and handling customer matches. Data fetching and mutations go through a hooks layer using React Query, which communicates with *SHAB Monitoring* via an Axios-based API client.

3.2 Architectural Decisions

This section documents the main architectural decisions made during the project. Each decision is described as an Architecture Decision Record (ADR), outlining the context, decision drivers, considered options, chosen solution, and resulting consequences. The final decision is documented in an ADR Y-Statement [\[18\]](#).

“In the context of <use case/user story>, facing <concern> we decided for <option> to achieve <quality>, accepting <downside>.”

The goal is to make the reasoning behind the architecture explicit and traceable, especially where trade-offs were necessary.

3.2.1 Runtime Separation for AI and Business Logic

Context: To keep the PoC close to a real production system, the industry partner’s existing Spring Boot and MySQL stack is reused for orchestration and business logic. The system additionally requires AI-related logic for publication classification and customer matching. Python provides the most mature ecosystem for these tasks: the OpenAI SDK, Splink, DuckDB, and lightweight serving frameworks such as FastAPI are well-supported and widely used in the ML community [\[19\]](#).

Decision Drivers:

- Alignment with the industry partner’s existing Java stack
- Access to mature AI and ML tooling in Python
- Isolation of Python-specific dependencies from the business application
- Implementation effort

Alternatives Considered:

- **Implement everything in Java:** Keeps the system in one runtime and avoids HTTP communication between services, but classification and matching would require heavier integration work or less mature library support for Splink, DuckDB, and the OpenAI SDK.
- **Introduce a separate Python AI service:** Preserves the existing Spring Boot application for orchestration and business logic while moving classification and matching into a runtime with stronger AI and ML library support.

Decision: In the context of integrating publication classification and customer matching into the existing *SHAB Monitoring* system, facing the need for mature AI and ML tooling while preserving the industry partner’s Java-based business application, we decided for a separate Python *Inference API* alongside the Java backend to achieve a clear service boundary and fit-for-purpose technology choices, accepting two runtimes, two deployments, and HTTP-based communication between them.

Consequences: *SHAB Monitoring* remains responsible for orchestration, persistence, and business rules, while the *Inference API* exposes the two inference operations required by the enrichment workflow. This keeps the Java service independent from Python-specific dependencies. The trade-off is additional operational overhead because the system now spans two technology stacks and communicates across a service boundary.

3.2.2 Jobs vs. Message Queue

Context: The system fetches SHAB publications, classifies them, and matches extracted entities against customer records. Publications arrive in batches once a day, so the workload is batch-oriented rather than latency-sensitive. The architecture must decide whether to model this processing as scheduled jobs or as a queue-based pipeline. Hohpe and Woolf describe the *Command Message* pattern as a message that instructs a receiver to perform an action [\[20, p. 145-146\]](#); this maps closely to how a queue-based approach would work here. Fowler notes that such patterns are often grouped loosely under “event-driven architecture”, a term that covers several distinct concepts [\[21\]](#).

Decision Drivers:

- Reliability and failure handling with retries
- Atomic persistence of job state
- Sufficient parallelism for the expected workload
- Low implementation and operational complexity
- Enqueue latency is not a relevant quality goal for daily batch workloads

Alternatives Considered:

- **Message queue:** Supports parallel processing via *Competing Consumers* [\[20, p. 502-507\]](#), but introduces a dual-write problem between database commits and message publishing.

Solving this requires the *Transactional Outbox* pattern [22, p. 97], plus additional reliability patterns such as *Guaranteed Delivery* [20, p. 122-126] and *Dead Letter Channel* [20, p. 119-121]. This complexity is not justified for a workload with no low-latency requirements.

- **Scheduled jobs:** A job scheduling framework uses the application database as the job store, giving atomic persistence, built-in retries, failure handling, and operational visibility without a message broker.

Decision: In the context of fetching, classifying, and matching SHAB publications that arrive in predictable daily batches, facing reliability requirements and the implementation complexity of a message queue, we decided for scheduled jobs to achieve robust processing with persistence, retries, and sufficient parallelism, accepting vendor lock-in and a higher enqueueing latency than a queue-based pipeline could provide.

Consequences: Publication processing is implemented with a job scheduling framework rather than a message broker. Job state is persisted atomically in the application database, avoiding the dual-write problem entirely. Retries and failure handling are provided by the framework, reducing custom implementation effort. The trade-off is that the architecture is optimized for reliable batch processing rather than immediate message consumption.

3.2.3 Job Scheduling Approach

Context: Building on the decision to use a job-based architecture (see [Section 3.2.2](#)), a suitable framework had to be selected. The publication processing jobs need persistence, retries, failure handling, and operational visibility while remaining practical to implement in the existing Spring Boot application. Three candidates were considered: Spring @Scheduled, Quartz, and JobRunr.

Decision Drivers:

- Ease of use
- Persistence of job state
- Dashboard and monitoring capabilities
- Retries and failure handling
- Dynamic and ad-hoc job support
- Documentation quality

Criteria:

Ease of use: Spring @Scheduled requires annotations [23]; Quartz needs more initial setup and per-job configuration [24]; JobRunr is straightforward and provides a lambda-based API for job definition [25].

Persistence: @Scheduled provides no persistence; Quartz supports it via a JDBC job store with additional schema setup [26]; JobRunr provides built-in persistence with automatic schema management [27].

Dashboard / monitoring: Neither @Scheduled nor Quartz provide a built-in dashboard; JobRunr includes one, though advanced features require the Pro edition [28].

Retries & failure handling: @Scheduled requires manual implementation; Quartz offers basic retries with manual policy configuration [29]; JobRunr provides exponential backoff, retry limits, and error notifications out of the box [30].

Dynamic / ad-hoc jobs: @Scheduled is limited to predefined schedules; Quartz supports dynamic jobs but requires a registered JobDetail class per job [31]; JobRunr supports dynamic scheduling and fire-and-forget jobs natively via its lambda-based API.

Documentation: @Scheduled has clear documentation scoped to the annotation [23]; Quartz documentation is dense and harder to navigate [24]; JobRunr provides comprehensive documentation with code examples [25].

The candidates were evaluated using a weighted decision matrix ([Tab. 23](#)):

Criteria	Weight %	Spring @Scheduled		Quartz		JobRunr	
		Value [1-6]	Total value	Value [1-6]	Total value	Value [1-6]	Total value
Ease of use	15	4	60	3	45	6	90
Persistence of jobs	15	1	15	4	60	6	90
Dashboard / monitoring	10	1	10	3	30	5	50
Retries & failure handling	35	2	70	3	105	6	210
Dynamic / ad-hoc jobs	15	2	30	5	75	5	75
Documentation	10	5	50	4	40	5	50
Total	100		235		355		565

Tab. 23: Decision Matrix - Job Scheduling

Decision: In the context of implementing reliable scheduled publication processing after deciding against a queue-based architecture, facing the need for persistent jobs, retries, failure handling, monitoring, and dynamic job execution, we decided for JobRunr as the job scheduling framework to achieve robust job processing with low implementation effort, accepting vendor lock-in and that some advanced features such as priority queues and enhanced dashboard capabilities are only available in the Pro edition.

Consequences: JobRunr scored highest overall, driven by its top score on retries and failure handling, the most heavily weighted criterion at 35%. Built-in persistence, monitoring, and exception handling remove the need for custom implementations that the other candidates would have required. The Pro-only features are not needed for the current PoC.

3.2.4 Containerization Strategy

Context: The PoC spans multiple runtimes: a Spring Boot backend, a FastAPI service, a React frontend, and a MySQL database. To demonstrate the full system working together and to run the final evaluation, all components need to run consistently without requiring manual setup of each runtime and database on the host machine.

Decision Drivers:

- Reproducible, single-command startup of the full stack
- No manual runtime or database installation required
- Dependency isolation per service

Alternatives Considered:

- **Host setup:** Requires manual installation of Java, Python, Node.js, and MySQL, making it harder to reproduce the demo environment reliably.
- **Containerize all components:** Packages each service with its dependencies, allowing the full stack to be started with a single Docker Compose command on any machine.

Decision: In the context of demonstrating and evaluating a polyglot PoC with a Spring Boot backend, FastAPI service, React frontend, and MySQL database, facing the need for a reproducible and easy-to-run environment, we decided for containerizing all components with Docker to achieve a consistent, single-command startup, accepting additional Dockerfile and Docker Compose configuration effort.

Consequences: The full system can be started on any machine without installing individual runtimes or databases. Each service runs with its own dependencies, avoiding version conflicts. The same images could be used as a starting point for a real deployment.

3.2.5 Probabilistic Record Linkage Library

Context: The customer matching workflow requires probabilistic record linkage because SHAB publication entities and customer records do not share a reliable unique identifier. The matching model must compare partially overlapping attributes such as names, addresses, and company details, and return explainable match candidates. Since no labeled training dataset was available for the PoC, the library had to support unsupervised model training.

Decision Drivers:

- Unsupervised model training support

- Explainable matching outputs
- Documentation quality and project maturity
- Fit with the Python-based *Inference API*
- Clear separation between model training and runtime inference

Options Considered:

- **Dedupe**: Supports fuzzy matching and entity resolution but was shut down in early 2023 [32]. Since the PoC is intended to lay a foundation for future development, starting with a discontinued library is counterproductive.
- **Python Record Linkage Toolkit**: Provides indexing, comparison functions, and unsupervised classifiers, but is centered around Pandas and NumPy workflows [33] and is described to be “developed with experimenting in first place and performance on the second place” [34].
- **Splink**: Supports probabilistic record linkage with unsupervised parameter estimation [35], and provides explainable matching outputs.

Decision: In the context of customer matching for SHAB publication entities, facing the absence of reliable unique identifiers and labeled training data, we decided for Splink as the record linkage library to achieve unsupervised, explainable probabilistic matching within the Python *Inference API*, accepting coupling to Splink’s model configuration format and supported database backends.

Consequences: Matching remains feasible without a labeled training dataset. If the dataset grows, the architecture has a clear path from local DuckDB execution to DuckDB execution on a powerful remote server. The trade-off is coupling to Splink’s model configuration format and supported database backends.

3.2.6 DuckDB

Context: The main application stores its operational data in MySQL. For customer matching, the system uses Splink, which does not support MySQL as an execution backend [36]. The customer dataset required for matching therefore has to be made available in a separate database technology supported by Splink.

Decision Drivers:

- Compatibility with Splink
- Minimal operational complexity for the PoC
- Easy to run locally and package with the *Inference API*

Options Considered:

- **Other Splink-supported backends**: Solves the compatibility issue but introduces additional operational complexity without a clear benefit for the PoC.
- **DuckDB**: Recommended by Splink [36], works as an in-process database management system [37], and keeps the runtime lightweight by packaging the matching dataset as a local database file [38].

Decision: In the context of preparing the customer dataset for Splink-based matching, facing the lack of MySQL support in Splink, we decided for DuckDB as the matching database backend to achieve compatibility with Splink and lightweight file-based execution in the PoC, accepting that customer data must be synchronized from MySQL into a second database technology.

Consequences: Customer data must be synchronized from MySQL to DuckDB before matching can run. The matching implementation uses a backend supported and recommended by Splink, keeping the runtime setup lightweight and file-based.

3.2.7 Linker Loading Strategy

Context: The matching service uses trained Splink linkers for person and company matching. Loading a linker requires reading model artifacts and initializing the underlying DuckDB connection, which is too expensive to repeat for every matching request. The *Inference API* can receive concurrent requests, so the loading strategy must avoid duplicate initialization and must not let one linker type block the other unnecessarily. During the PoC, unsynchronized lazy loading led to a reproducible failure: Concurrent matching requests could start linker initialization at the same time, leaving the service in an inconsistent state. This made synchronization around linker initialization a design requirement.

Decision Drivers:

- Low request latency after first use
- Lightweight startup; only load linkers once they are actually needed
- Concurrency safety without unnecessary contention between person and company linkers

Options Considered:

- **Load on every request:** Simple to implement, but repeats expensive initialization work on every call, significantly increasing latency.
- **Eager load on startup:** Makes the first request faster, but increases startup time.
- **Single global lock:** Avoids duplicate initialization, but unnecessarily serializes person and company linker loading.
- **Lazy loading with independent locks:** Avoids repeated initialization, keeps startup lightweight, and allows person and company linkers to be initialized independently [39].

Decision: In the context of serving person and company matching requests in the *Inference API*, facing expensive linker initialization and concurrent requests, we decided for lazy in-memory linker loading with independent caches and locks per linker type to achieve lower repeated request latency and concurrency-safe initialization, accepting that initialized linkers remain in memory for the lifetime of the service process.

Consequences: Request latency is reduced after the first use while startup remains lightweight. Person and company linkers initialize independently, limiting contention between the two matching workflows. The trade-off is that linker instances remain in memory for the lifetime of the service process, so memory usage scales with the number and size of cached linkers.

3.2.8 Shared Matching Model Package

Context: Both the matching model training tool and the *Inference API* depend on the same Splink-specific logic: loading the DuckDB customer dataset, preprocessing records, defining person and company model configurations, building trained linkers, and loading saved model definitions. If this logic were implemented separately in both consumers, implementations could drift apart and produce inconsistent matching behavior.

Decision Drivers:

- Consistent behavior between training and inference
- No duplicated code
- Model configuration kept in one place
- Training tool and *Inference API* can still evolve as separate consumers

Options Considered:

- **Duplicate logic in both consumers:** Keeps each component self-contained, but risks preprocessing rules, thresholds, or model configuration diverging between training and inference.
- **Separate internal service:** Centralizes the logic, but adds another runtime component and network boundary for functionality used only by Python components in the PoC.
- **Shared Python package:** Keeps common matching logic in one codebase while allowing both consumers to depend on it directly [40].

Decision: In the context of sharing Splink-specific matching logic between the training tool and the *Inference API*, facing the risk of duplicated and diverging preprocessing, model configuration, and linker loading code, we decided for a shared Python package to achieve consistent matching behavior and maintainable reuse, accepting an explicit package dependency between the two consumers.

Consequences: Preprocessing and model configuration remain consistent across training and inference. Changes to matching behavior can be made in one place and are picked up by both consumers. The trade-off is that changes to the package must be coordinated with both consumers, and package versioning becomes relevant because the components are deployed independently.

3.2.9 Strategy for Publication Format Handling

Context: SHAB supports many distinct publication types that vary in both XML structure and the semantics of the provided information. Because of this variance, a fully generic parsing approach cannot reliably handle publication-specific business rules. Within the project scope, 12 distinct publication structures were identified and are handled. While most formats are unique, certain core entities such as “Person” and “Company” follow a standardized schema across publication types [41].

Decision Drivers:

- Parsing precision per publication type
- Explicit handling of publication-specific business rules
- Maintainability as more formats are added
- Reuse of common entity parsing logic

- Extensible pipeline without changing orchestration logic

Options Considered:

- **Generic XML parsing:** Can read the XML structure, but cannot interpret which fields carry which meaning for a given publication type; business rules and field semantics still have to be handled explicitly per format.
- **Single large conditional parser:** Keeps all parsing logic in one place, but becomes hard to maintain as more publication types are added and mixes unrelated format-specific rules.
- **Specialized parsers with the Strategy pattern:** Isolates publication-specific logic while allowing the processing pipeline to select the correct parser dynamically based on the identified sub-rubric [42].

Decision: In the context of processing structurally and semantically different SHAB publication formats, facing high variance between sub-rubrics and publication-specific business rules, we decided for specialized parsers orchestrated with the Strategy pattern to achieve precise and extensible publication handling, accepting additional implementation effort for each supported publication structure.

Consequences: Each supported publication format is parsed with logic tailored to its XML structure and business meaning. Shared entities such as “Person” and “Company” reuse common parsing logic where the API schema is consistent across formats [41]. The trade-off is that new publication structures require an explicit parser implementation before they can be processed.

3.2.10 International Data Format Handling

Context: SHAB does not only include Swiss data (like addresses, company listings, etc.) but can also contain data from companies/ persons with an abroad address. This is a problem for the publications handling because these international publications are highly irregular.

Decision Drivers:

- Proof of concept realization
- High effort for little yield in implementation

Options Considered:

- **Custom approach for each international publication:** This approach would have created a repository of different international SHAB publications with the hopes of catching more irregularities with an increasing repository
- **Ignore international publications for customer matching:** Ignore the international publications completely, because the focus of this thesis is a proof of concept and the yield from handling those publications is minor.

Decision: In the context of handling international SHAB publications, facing high irregularity in the entity data, we decided (and conferred with the industry partner) to ignore such SHAB publications for customer matching (the classification step will still be carried out), accepting the inability to match a minor amount of SHAB publications.

Consequences: A minor amount (approx. 100 - 200 publications per year) will not be able to be fully processed by our proof of concept system.

3.3 Data Model

3.3.1 SHAB Monitoring Database Schema

This section describes the database schema used by *SHAB Monitoring* and the business entities it stores. Job-related persistence (idempotency keys and audit trail entries) is part of the job processing model and is covered in [Section 3.4](#).

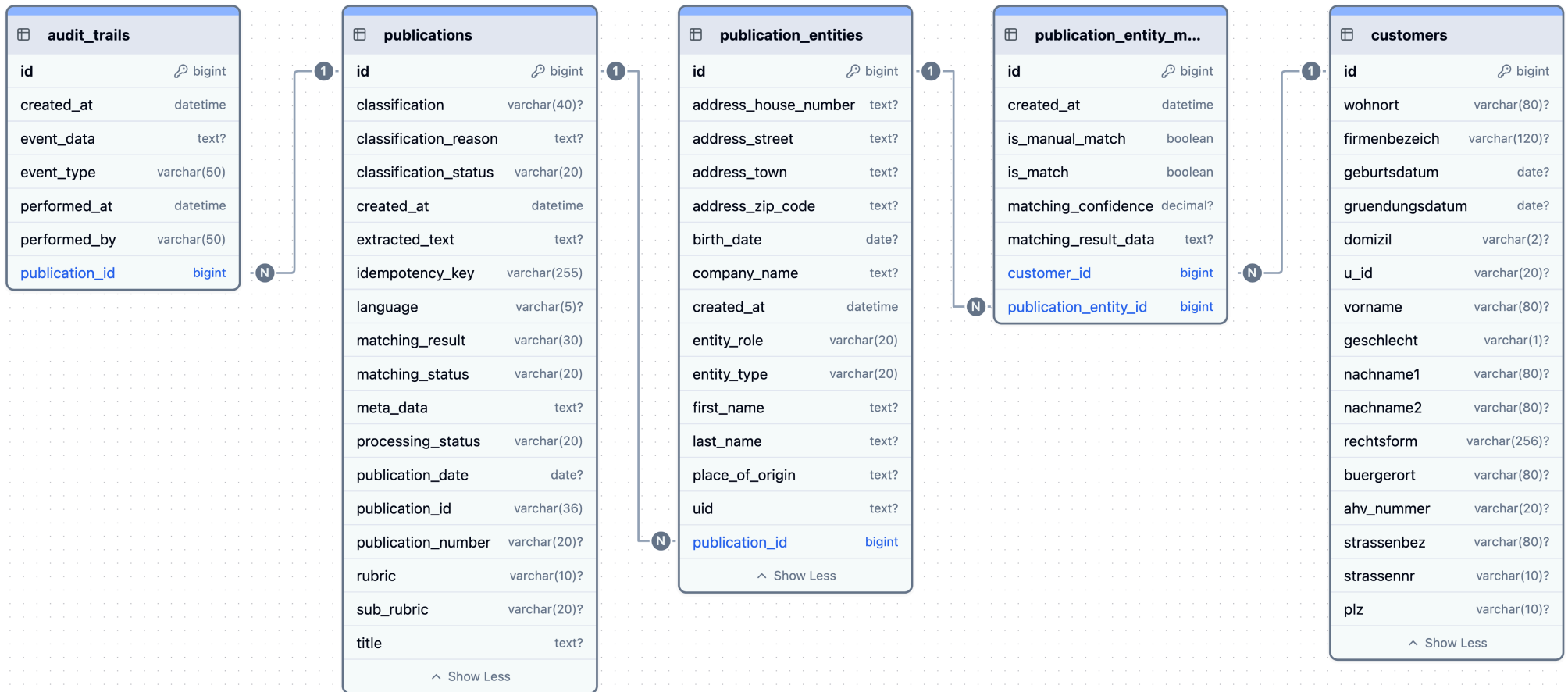


Figure 9: SHAB Monitoring - Database Schema

The schema consists of five tables organized around a central processing flow (publications → publication_entities → publication_entity_matches), backed by the customers reference table, with audit_trails recording changes across the system.

The central table is publications, which stores raw publication data, extracted text, classification result, matching status, and the current job processing state. The classification_status and matching_status columns track the completion of these two steps using a shared enum (PENDING / COMPLETED); the *Enrichment-Job* uses these flags to determine which steps still need to be executed. The idempotency_key column ensures that each publication is processed only once; the mechanism is described in [Section 3.4.2](#).

Entities extracted from a publication are stored in publication_entities, each linked to its parent publication. Candidate customer matches are stored in publication_entity_matches, which links a publication_entity to a customer record and stores the confidence score and final match decision.

The customers table holds the customer master data against which publications are matched. It stores both person attributes (name, date of birth, place of origin) and company attributes (company name, UID), reflecting that customers can be either natural persons or legal entities. The column names are in German, inherited from the source system.

Every automated state change and user action is recorded in audit_trails, which links each event to the affected publication and stores the event type, timestamp, and actor.

3.3.2 Inference API Database Schema

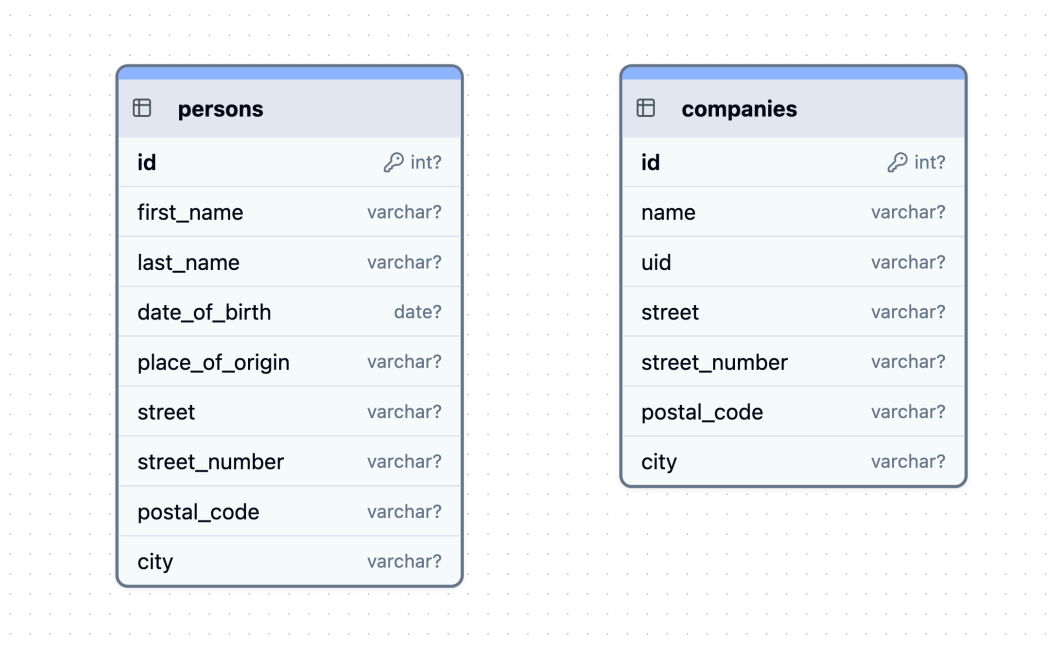


Figure 10: Inference API - Database Schema

The *Inference API* uses a DuckDB database containing two tables that mirror the customer attributes relevant for matching.

The persons table stores natural person records with name, date of birth, place of origin, and address fields. The companies table stores legal entity records with company name, UID, and address fields. Both tables use a surrogate integer primary key that corresponds to the customer identifier in MySQL. The schema is intentionally flat; no foreign keys or constraints are defined beyond the primary key, as DuckDB is used exclusively as a read-only matching backend.

3.4 Job Processing Model

This section describes how the system processes publications as background jobs: the lifecycle of a job, the states it can be in, the idempotency mechanism that guarantees each publication is processed only once, and the audit trail that records every state change and user action.

3.4.1 Job lifecycle & states

The system uses two background jobs to process publications: the *Importer-Job*, which fetches new publications from the SHAB API, and the *Enrichment-Job*, which classifies it, extracts entities from a publication, and matches the entities against the customer database.

Because publications are released in a single batch at 00:00 UTC ([Section 1.4.3](#)), the *Importer-Job* is scheduled to run once at 00:05 UTC to capture the new batch as soon as it becomes available. If a transient failure occurs (e.g., the SHAB API is unavailable), JobRunr manages retries automatically with increasing backoff intervals.

The following states are tracked for each publication:

Status	Description
SCHEDULED	The publication is queued for processing and awaiting execution by the <i>Enrichment-Job</i> .
PROCESSING	The <i>Enrichment-Job</i> is currently processing the publication.
ENRICHED	The publication has been successfully enriched and all required data transformations have been applied.
FAILED	The processing encountered a terminal error that prevents further execution. In the PoC, this includes unhandled edge cases such as publications involving entities with addresses outside Switzerland. Unexpected errors that exhaust all retries also end up in this state.

Tab. 24: Publication processing states

Transient failures during retries are not stored as a distinct state; the publication remains in PROCESSING until the job either completes successfully (ENRICHED) or exhausts its retries and transitions to FAILED. JobRunr handles retries automatically using exponential backoff, reducing pressure on downstream services during transient failures [\[30\]](#).

Checkpoints: Before performing an operation (fetching metadata, classifying, or matching), the *Enrichment-Job* checks the corresponding flag on the publication to determine whether the step was already completed in a previous run.

Resume logic: After a failure, a retried *Enrichment-Job* skips any steps already marked complete in the database, making individual processing steps idempotent and allowing interrupted jobs to resume safely.

Terminal failures: When an unhandled edge case is encountered, the job is set to FAILED and not retried. The publication remains visible in the *UI* with whatever metadata was successfully fetched before the failure.

3.4.2 Idempotency

A core property of the processing pipeline is idempotency: a job can be retried at any point without risking data loss or duplicate processing [\[20, p. 528\]](#). This is achieved through processing states, per-job idempotency keys, and a unique constraint in the database. The system distinguishes between deterministic scheduling (automated, predictable) and manual overrides (user-initiated, unpredictable), each producing a different idempotency key.

3.4.2.1 Automated fetching

For each *Importer-Job* run, an idempotency key of the form `auto-importer-<DATE>` is generated. This ensures that each publication is saved only once per day, even if the *Importer-Job* runs multiple times.

When the *Importer-Job* schedules a publication, it derives a per-publication key for the *Enrichment-Job* of the form `auto-importer-<DATE>-<PUBLICATION_ID>`.

3.4.2.2 Manual triggering

A user can trigger the *Importer-Job* manually for a specific day or publication, for example to reprocess data after a fix. In both cases the publication is fully reprocessed: all steps of the *Enrichment-Job* are executed again, even those that completed successfully in a previous run.

An intent-based idempotency key is derived from the trigger timestamp (`manual-importer-<TIMESTAMP_MILLIS>` and `manual-importer-<TIMESTAMP_MILLIS>-<PUBLICATION_ID>`). Because the key differs from any previous one, the database creates a new publication record rather than overwriting the existing one. Both the original and the reprocessed publication persist independently, making it possible to compare their audit trails side by side.

3.4.2.3 Idempotency keys

Key	Description
<code>auto-importer-2026-05-11</code>	<i>Importer-Job</i> key for a scheduled run on 11.05.2026. Never stored in the database; used only as a prefix to derive per-publication <i>Enrichment-Job</i> keys.
<code>auto-importer-2026-05-11-<PUBLICATION_ID></code>	<i>Enrichment-Job</i> key when triggered by the scheduled <i>Importer-Job</i> .
<code>manual-importer-1778486217181</code>	<i>Importer-Job</i> key for a manually triggered run, where the suffix is the trigger time in milliseconds. Never stored in the database; used only as a prefix to derive per-publication <i>Enrichment-Job</i> keys.
<code>manual-importer-1778486217181-<PUBLICATION_ID></code>	<i>Enrichment-Job</i> key when triggered by a manual <i>Importer-Job</i> run.

Tab. 25: Idempotency keys used by the *Importer-Job* and *Enrichment-Job*

3.4.3 Audit Trail

The system maintains an audit trail for every publication processed. Each entry records a timestamp, the operation performed, and whether it was triggered by the system or a user. All entries are written within the same database transaction as the state change they describe, guaranteeing consistency between the audit trail and the data it records.

3.4.3.1 Audit Trail Operations

Event Type	Description
<code>IMPORTER_CREATE_PUBLICATION</code>	The <i>Importer-Job</i> created a new publication record in the database. The timestamp reflects when the record was created in the system, not when SHAB published the publication.
<code>ENRICHER_ENQUEUED</code>	The <i>Enrichment-Job</i> has been enqueued for processing the publication.
<code>ENRICHER_STARTED</code>	The <i>Enrichment-Job</i> has started processing the publication.
<code>ENRICHER_RETRY</code>	The <i>Enrichment-Job</i> has failed and is being retried.
<code>ENRICHER_SAVE_METADATA</code>	The <i>Enrichment-Job</i> has saved publication metadata to the database.
<code>ENRICHER_CLASSIFICATION_STARTED</code>	The <i>Enrichment-Job</i> has started classification.
<code>ENRICHER_CLASSIFICATION_COMPLETED</code>	The <i>Enrichment-Job</i> has completed classification.
<code>ENRICHER_MATCHING_STARTED</code>	The <i>Enrichment-Job</i> has started customer matching.
<code>ENRICHER_MATCHING_COMPLETED</code>	The <i>Enrichment-Job</i> has completed customer matching.
<code>ENRICHER_COMPLETED</code>	The <i>Enrichment-Job</i> has completed processing the publication.
<code>ENRICHER_FAILED</code>	The <i>Enrichment-Job</i> has failed and will not be retried.
<code>PUBLICATION_CLASSIFICATION_CHANGED</code>	A user manually changed the classification of the publication via the UI.
<code>PUBLICATION_MATCHING_CHANGED</code>	A user manually changed the customer matching of the publication via the UI.

Tab. 26: Audit trail event types

3.5 Machine Learning Architecture

The machine learning component of the system addresses two distinct tasks: classifying SHAB publication texts ([Section 1.4.2](#)) and matching extracted entities against the internal customer database. Both tasks are integrated into the enrichment workflow but differ in input structure, output semantics, and quality risks. The classification task operates on HR02 publication text and structured mutation attributes. Customer matching compares incomplete or variably formatted entity data against existing records, where exact equality checks are insufficient. Each task therefore uses a dedicated model, while both are exposed through the shared *Inference API*.

3.5.1 Classification Architecture

The classification step assigns one or more internal event codes to each processed HR02 publication. All classification codes are defined in [Tab. 48](#) . Within the system, only the numeric suffix is persisted and exchanged with the *Inference API*; for example, HR02 / 54 is stored and transmitted as 54.

Classification is multi-label: a single publication may receive more than one code when it announces multiple independent events. This is common in HR02 mutation texts, where a single entry can simultaneously report a domicile change and a change in signing authority, as in the following example [\[43\]](#) :

“F.S Food GmbH, in Wildhaus-Alt St. Johann, CHE-341.167.395, Gesellschaft mit beschränkter Haftung (SHAB Nr. 11 vom 17.01.2023, Publ. 1005654132). Domizil neu: Halden 731, 9657 Unterwasser. Eingetragene Personen neu oder mutierend: Schaller, Fabrice, von Diessbach bei Büren, in Unterwasser (Wildhaus-Alt St. Johann), Gesellschafter und Vorsitzender der Geschäftsführung, mit Einzelunterschrift, mit 20 Stammanteilen zu je CHF 1'000.00 [bisher: in Wildhaus (Wildhaus-Alt St. Johann), Gesellschafter und Geschäftsführer, mit Einzelunterschrift]; Schaller, Hans-Peter, genannt Jean-Pierre, von Diessbach bei Büren, in Wildhaus (Wildhaus-Alt St. Johann), Geschäftsführer, mit Einzelunterschrift.”

The chosen architecture is a hybrid of deterministic rules and LLM-based text classification. Structured attributes from the SHAB XML payload are always evaluated first. If the attribute `others=true`, indicating that the publication text contains events not captured by the structured attributes, LLM inference is additionally triggered through the *Inference API*. The final classification is the union of both result sets. As noted in [Section 1.4.1](#) , structured attributes can underclassify. An override option therefore exists that forces LLM inference for every publication regardless of attribute state, covered in [Section 4.1.2.2](#) .

The LLM prompt constrains the model to the labels defined in [Tab. 49](#) and requires a structured JSON response containing a list of classification codes with a short justification for each. To improve precision, ambiguous boundaries between similar event types are resolved through explicit decision rules stated directly in the prompt, ensuring a code is only returned when the publication text clearly supports it.

3.5.2 Customer Matching Architecture

The customer matching step connects entities extracted from SHAB publications with the internal customer database. Unlike classification, the output is not a category label but a ranked set of candidate customer records with match probabilities.

3.5.2.1 Probabilistic Record Linkage

Exact equality checks are insufficient for this task: names, addresses, and company identifiers may be missing, abbreviated, formatted differently, or contain minor spelling differences. Probabilistic record linkage (PRL) addresses this by combining multiple field-level signals into a single match probability, allowing records to be linked even when no single field matches exactly.

Splink was selected as the PRL library over alternatives such as Dedupe (decision described in [Section 3.2.5](#)). Splink implements the Fellegi-Sunter model with unsupervised EM-based parameter estimation, requiring no labeled training data.

3.5.2.2 Dataset

The internal customer database stores persons and companies in a single table. For matching, this is insufficient: persons and companies carry different attributes and require different comparison logic, so they must be modeled separately.

No real customer dataset was available for the PoC; instead, a synthetic dataset was derived from publication entities already extracted by the system. The technical realization of this dataset is described in [Section 4.3](#) .

Having DuckDB as a native backend of Splink allows to split the customers into separate persons and companies tables, each containing only the attributes relevant to that entity type. Customer IDs are preserved across MySQL and DuckDB to ensure that match results returned by the Inference API can be directly resolved against the records stored by *SHAB Monitoring*.

Before records are used for training or inference, they pass through a shared preprocessing function. This function is applied both when the dataset is loaded and when a new entity is received at the matching endpoint, ensuring that stored records and incoming publication entities are normalized consistently before comparison.

3.5.2.3 Model Design

To keep inference tractable, the matching model uses blocking rules before running the full probabilistic comparison. A blocking rule is a SQL-like condition that selects only candidate record pairs sharing a blocking key, such as a date of birth, postal code, UID, or name. Instead of comparing the incoming entity against every customer record, Splink evaluates the full comparison model only on records that pass at least one of these blocking rules.

Only candidates exceeding a configurable match probability threshold are returned, with the number of results capped per query.

4 Implementation

This chapter describes the concrete implementation of the system based on the architecture and requirements defined in the previous chapters. It explains how SHAB publications are processed, how the individual services realize their responsibilities, how the application is deployed, and how the implemented functionality is verified through testing.

4.1 Publication Processing Pipeline

The publication processing pipeline is the core of the system, orchestrating the lifecycle of each publication from ingestion through classification and matching. This section describes the workflow end to end, with reference to the data model ([Section 3.3](#)) and job mechanisms ([Section 3.4](#)).

4.1.1 Workflow Overview

The pipeline is driven by two cooperating background jobs:

- The *Importer-Job* retrieves new publications from the SHAB API, dispatches enrichment jobs and registers them in the database.
- The *Enrichment-Job* handles each individual publication: it fetches the full publication content, enriches it, and matches the involved entities against the customer database.

4.1.1.1 Importer-Job

The *Importer-Job* fetches the latest publications from the SHAB API. For each publication retrieved, it checks whether the publication already exists in the database using an idempotency key derived from the execution mode (e.g., auto-importer-`<DATE>` or manual-importer-`<TIMESTAMP>`), as described in [Section 3.4.2](#) . If the publication is new, the job stores the id, writes `IMPORTER_CREATE_PUBLICATION` and `ENRICHER_ENQUEUED` audit trail entries, and enqueues an *Enrichment-Job* with its per-publication key (e.g., `<prefix>-<PUBLICATION_ID>`). The publication creation, audit trail, and job enqueue are committed in a single transaction to ensure atomicity. This workflow is illustrated in [Figure 11](#) .

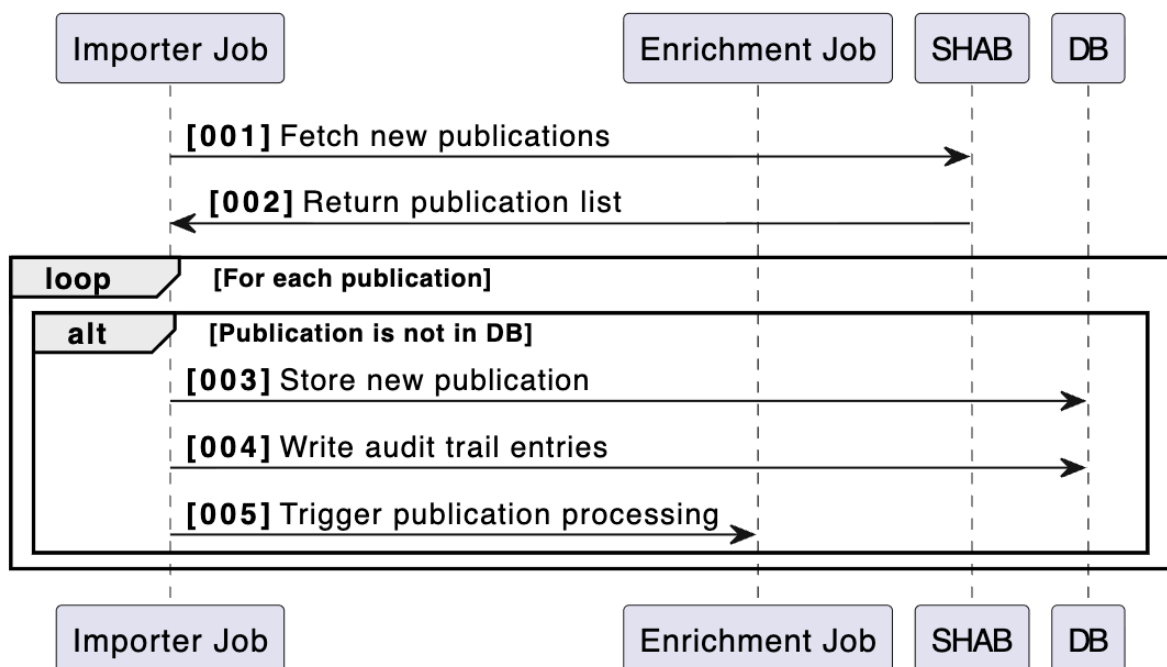


Figure 11: Sequence diagram - Importer Job

4.1.1.2 Enrichment-Job

The *Enrichment-Job* processes a single publication. Each step is guarded by the publication's per-step completion flags so that retries skip work already done. Steps that mutate the database write an audit trail entry within the same transaction (see [Section 3.4.3](#)). The workflow proceeds as follows:

1. **Metadata enrichment:** If the publication's full content is not yet in the database, it is fetched from the SHAB detail endpoint, stored, and recorded with a `ENRICHER_SAVE_METADATA` audit trail.
2. **Handler dispatch:** The job looks up the handler responsible for the publication's sub-rubric ([Section 3.2.9](#)). If no handler exists, the sub-rubric is out of scope and the job completes without further processing. If a handler exists, the remaining steps are delegated to it.
3. **Classification:** If the publication has not yet been classified, the handler performs rule-based classification using the publication's structured attributes. If the attribute `others=true`, or the `always-infer` override is enabled, the *Inference API* is additionally called to classify the unstructured text via the LLM. The combined result is persisted with the `ENRICHER_CLASSIFICATION_*` audit trail.
4. **Customer matching:** If the publication has not yet been matched, the handler extracts and calls the *Inference API* matching endpoint, persists the result, and writes the `ENRICHER_MATCHING_*` audit entries.
5. **Completion:** The job marks itself complete (`ENRICHER_COMPLETED`) and sends a notification email to the configured recipient.

This workflow is illustrated in [Figure 12](#).

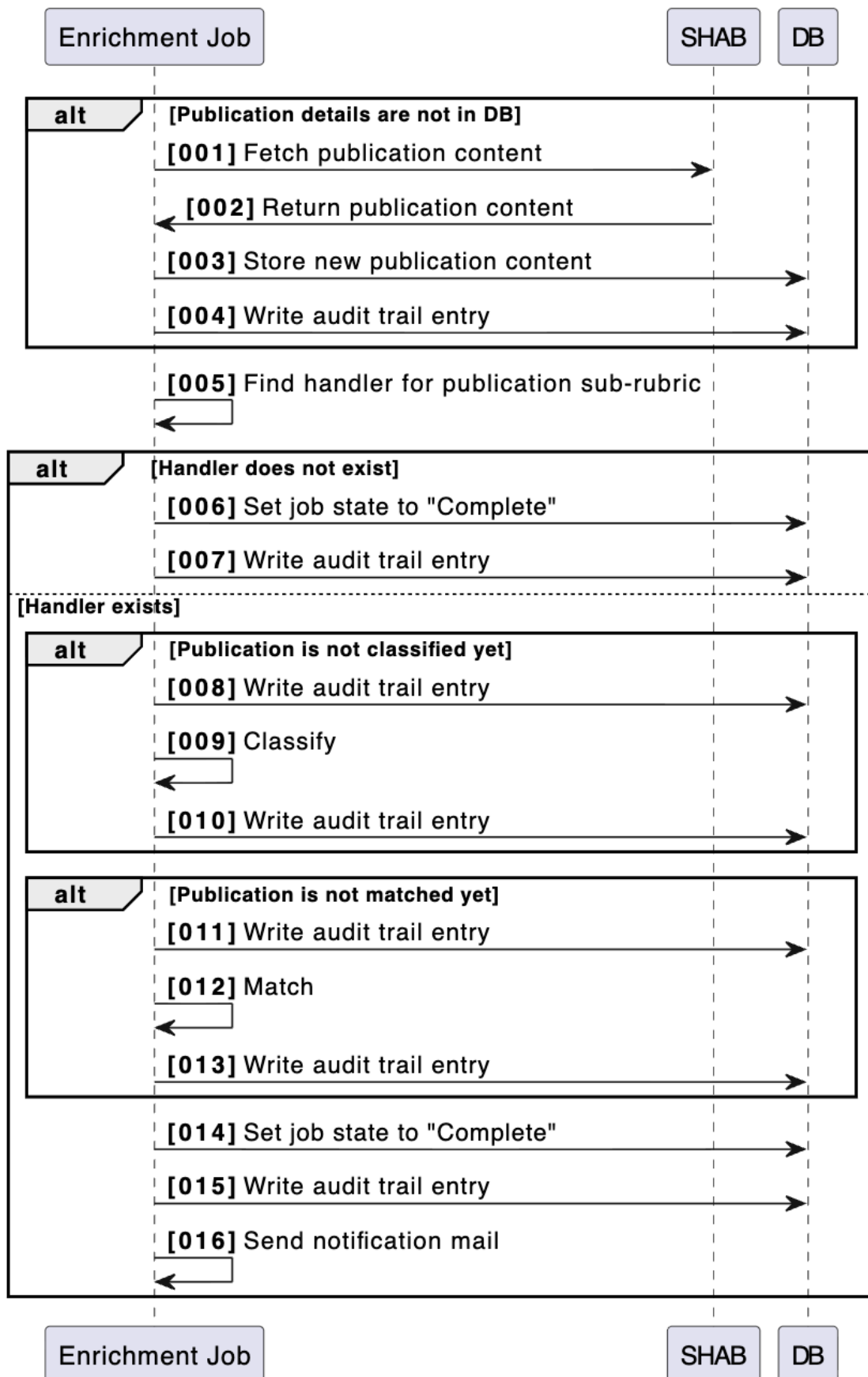


Figure 12: Sequence diagram - Enrichment Job

4.1.2 Classification

The classification architecture is described in [Section 3.5.1](#). This section covers its concrete realization in the enrichment workflow, illustrated in [Figure 13](#).

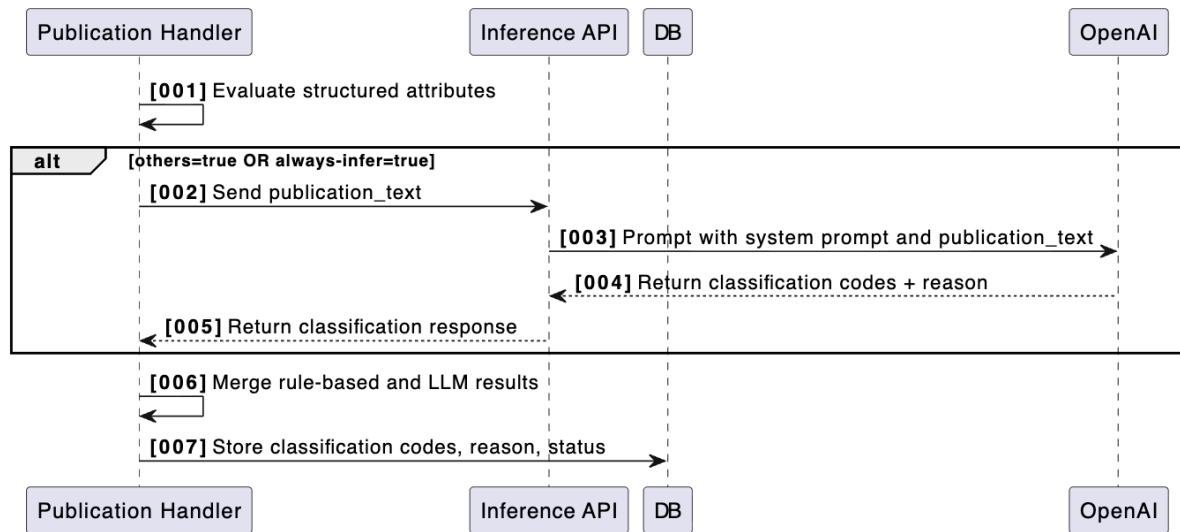


Figure 13: Sequence diagram - Classification

4.1.2.1 Rule-based Classification

The rule-based stage corresponds to step [001] in [Figure 13](#). Hr02Handler inspects the parsed HR02 DTO and maps explicit SHAB XML attributes to classification codes. Additionally before-and-after value comparisons are made for company name, company purpose and address regardless of whether the XML attribute indicates a change or not.

4.1.2.2 LLM Classification

The LLM stage corresponds to steps [002]–[006] in [Figure 13](#). It is triggered when `others=true` or the `always-infer` override is active. Hr02Handler sends the extracted publication text to the `/classification` endpoint of the *Inference API*. The rule-based and LLM results are merged into the final classification.

4.1.2.3 Persistence and Traceability

Step [007] in [Figure 13](#) corresponds to persisting the classification result. The publication record is updated with the merged classification codes, the model reason, and the completed classification status. An audit trail entry records which codes originated from the rule-based stage and which from the LLM, keeping the classification traceable in the *UI*.

4.1.3 Customer Matching

This section covers the customer matching realization in the enrichment workflow as planned in [Section 3.5.2](#), assuming the customer dataset has been prepared as described in [Section 4.3](#) and the Splink models trained as described in [Section 4.5](#). The full process is illustrated in [Figure 14](#).

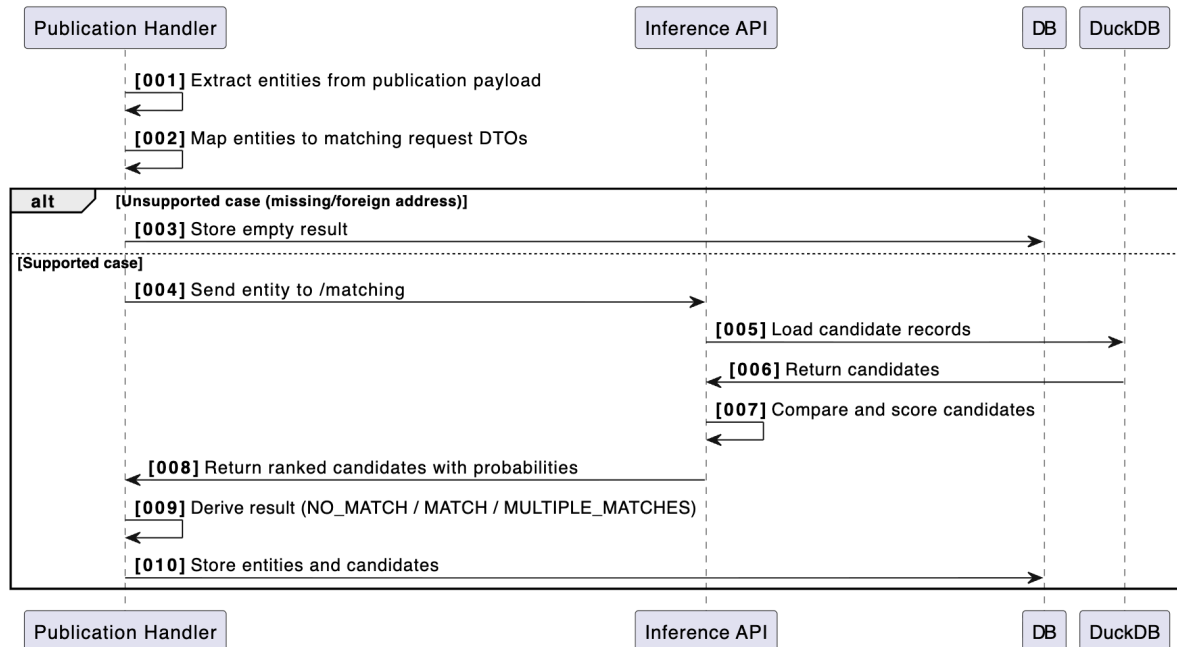


Figure 14: Sequence diagram - Customer Matching

4.1.3.1 Entity Extraction and Mapping

Steps [001]–[002] in [Figure 14](#) correspond to entity extraction and mapping. Each handler extracts the relevant entities from the parsed SHAB detail payload and maps them into matching request DTOs. Unsupported cases such as missing, foreign, or non-standard address structures are rejected at this stage, as the current model is trained on Swiss address fields (step [003]).

4.1.3.2 Candidate Resolution

Steps [004]–[009] correspond to the matching request and response. Each entity is sent individually to the `/matching` endpoint of the *Inference API*. The handler derives a high-level result from the returned candidates:

- **NO_MATCH**: no candidate was returned.
- **MATCH**: exactly one candidate was returned.
- **MULTIPLE_MATCHES**: several candidates were returned.

This result distinguishes publications that can be linked directly from those requiring manual inspection.

4.1.3.3 Persistence

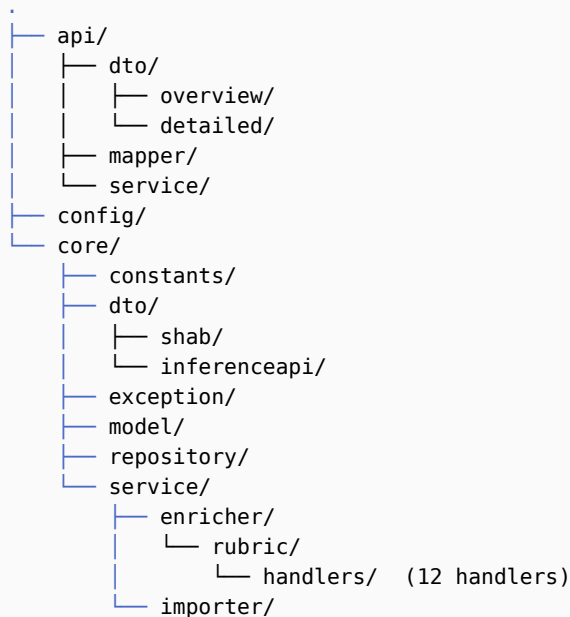
Step [010] correspond to persisting the matching result. The extracted publication entities and all returned candidate matches are stored together. The detailed candidate data is retained in all cases, allowing users to review, confirm, reject, add, or remove matches through the *UI*. The evaluation of the matching model is documented in [Section 5.3](#).

4.2 SHAB Monitoring (Spring Boot)

This section covers the internal service structure, communication with the SHAB API and *Inference API*, and the realization of the sub-rubric handlers, persistence layer, REST API, and notification logic.

4.2.1 Package structure

The application is organized into three primary top-level packages: `api`, `config`, and `core`.



Listing 1: Simplified package structure of *SHAB Monitoring*

4.2.1.1 core

The `core` package contains all domain logic, kept separate from the web layer.

- `model` defines the JPA entities and enumerations.
- `repository` provides Spring Data interfaces for each aggregate.
- `service` contains the business logic, including job implementations, the enrichment pipeline, and the notification system.
- `dto` contains the data transfer objects for both the SHAB API and the *Inference API*.
- `constants` centralizes classification labels and sub-rubric identifiers.
- `exception` defines typed exceptions to distinguish between retryable and non-retryable failures.

The enrichment logic within `service/enricher/handlers` implements the Strategy pattern ([42]) for individual processing. `RubricHandler` serves as the base interface, while `AbstractRubricHandler` provides shared behavior. Twelve concrete handlers implement the relevant sub-rubrics: `BhHandler`, `EsHandler`, `Hr02Handler`, `KkHandler`, `LsHandler`, `NaHandler`, `SbHandler`, `Sr01Handler`, `Sr02Handler`, `Sr03Handler`, `Sr05Handler`, and `Sr06Handler` ([Section 4.1.1](#)).

4.2.1.2 api

The `api` package provides the HTTP interface consumed by the *UI*.

- Three controllers (`AdminController`, `CustomerController`, and `PublicationController`) handle requests for administration, customer data, and publication management respectively.
- Query services retrieve data from the database, map results to response DTOs, and return them to the controllers.
- Mappers convert domain models into response DTOs.
- Two DTOs support the primary application views: `overview` for paginated list responses and `detailed` for a full publication view including audit trail, classification, and match results.

4.2.1.3 config

The config package contains Spring configuration classes for the REST client setup, JobRunr integration, XML deserialization, and the publication scheduler. It also includes the `ExtractCustomers` utility, which exports customer records [Section 4.3.1](#) for model training.

4.2.2 SHAB API Client

The SHAB API is accessed via Spring's `RestClient` ([Section 3.1.3.1](#)), configured in `RestClientConfig` with the base URL injected from the `shab.api.base-url` property. No authentication is required. Calls are made directly from `ImportHelper` and the `EnricherHelper` via the injected `RestClient` bean. In the development environment, the base URL points at a caching proxy rather than the live SHAB API to avoid redundant upstream calls during development and testing.

The first endpoint (`/api/v1/publications/xml`) returns a paginated list of publications for a given day in XML format, consumed by the *Importer-Job*. Each request is constructed using the query parameters in [Tab. 27](#).

Parameter	Value / Description
<code>pageRequest.page</code>	The page index, incremented until all available pages are retrieved.
<code>pageRequest.size</code>	The number of records per page, fixed at 1'000.
<code>publicationStates</code>	Restricted to PUBLISHED.
<code>pageRequest.sortOrders</code>	Set to publication number in descending order.
<code>publicationDate.start</code>	Set to the target day for daily imports.
<code>publicationDate.end</code>	Set to the target day for daily imports.
<code>subRubrics</code>	The sub-rubrics to include in the request.

Tab. 27: Parameters for SHAB API bulk publications endpoint

Pagination is driven by the `total` field in the `BulkExport` response ([Listing 2](#)), using a fixed page size of 1'000. If the total exceeds the current page boundary, the next page is fetched recursively until all publications for the day are collected.

```
public record BulkExport(long total, PageRequest pageRequest, List<Publication> publications) {}
public record PageRequest(int page, int size) {}
```

Listing 2: SHAB API bulk response DTOs

The second endpoint (`/api/v1/publications/{id}/xml`) returns the full details of a single publication, including sub-rubric-specific content. The *Enrichment-Job* uses this endpoint to retrieve the data required for classification and matching. DTOs for both endpoints are defined in `core/dto/shab/bulk/` and `core/dto/shab/single/` and deserialized using Jackson XML. Because response structures vary across sub-rubrics, the appropriate DTO type is resolved using generics, with the sub-rubric identifier derived from the bulk response; this is covered in [Section 4.2.4](#). Error handling uses the standard exceptions thrown by `RestClient`. Retry logic is delegated to the JobRunr retry mechanism ([Section 3.4.1](#)).

4.2.3 Inference API Client

SHAB Monitoring communicates with the *Inference API* for two enrichment steps: classification and matching, as described in [Section 3.5](#). The client is configured as a `RestClient` bean in `RestClientConfig`, with the base URL injected from the `shab-monitoring.inference-api.base-url` property. Both operations are invoked from `AbstractRubricHandler` and use JSON for data exchange. Error handling and retry logic follow the same approach as the SHAB API client; JobRunr retry mechanism ([Section 3.4.1](#)).

4.2.3.1 Classification

A POST request to `/classification` transmits the publication text and returns a list of classification codes with a reasoning string. This implements the LLM stage of the two-stage classification pipeline described in [Section 4.1.2](#). The classification codes and reasoning string are persisted and surfaced in the audit trail.

The following example shows a real HR02 mutation where the model correctly identifies both a domicile change (code 73) and a change in decision-makers (code 54).

```
// Request
{
  "publication_text": "Schweizerischer KMU Verein (SKV), in Naters, CHE-439.892.748, Verein (SHAB Nr. 52 vom 17.03.2026, Publ. 1006598959). Domizil neu: c/o Diego Schnydrig, Dammweg 11d, 3904 Naters. Ausgeschiedene Personen und erloschene Unterschriften: Schnydrig, Diego, von Naters, in Naters, Mitglied des Vorstandes, mit Einzelunterschrift."
}
// Response
{
  "classifications": ["73", "54"],
  "reason": "The text reports a new domicile (Domizil neu), which corresponds to code 73. It also lists a person (Diego Schnydrig) identified as a member of the board (Mitglied des Vorstandes) who is exiting, which constitutes a change in decision-makers (Vernetzung Entscheider), corresponding to code 54."
}
```

Listing 3: Classification request and response: HR02-1006601515 [15]

4.2.3.2 Matching

A POST request to /matching transmits either a person or a company extracted from the publication and returns a ranked list of candidate matches with confidence scores. The MatchingRequest contains exactly one of the two entity fields; the other remains null.

The following example matches the company Schweizerischer KMU Verein (SKV) from the same publication against the customer database.

```
// Request
{
  "person": null,
  "company": {
    "name": "Schweizerischer KMU Verein (SKV)",
    "uid": "CHE-439.892.748",
    "address_street": "Dammweg",
    "address_house_number": "11d",
    "address_zip_code": "3904",
    "address_town": "Naters"
  }
}
// Response
{
  "matches": [
    {
      "id": 4821,
      "confidence": 0.97,
      "is_match": true,
      "result_data": "..."
    }
  ]
}
```

Listing 4: Matching request and response: HR02-1006601515 [15]

The complete response, including id, confidence, is_match, and result_data, is persisted by the enrichment helper. The id field corresponds to the customer ID in the customers table, preserved through the DB Sync pipeline ([Section 4.3](#)) to ensure match results can be resolved against the records stored in *SHAB Monitoring*.

4.2.4 Sub-rubric Handlers

The sub-rubric handlers realize the Strategy pattern described in [Section 3.2.9](#). The implementation uses a three-level hierarchy: the `RubricHandler<T>` interface, the `AbstractRubricHandler<T>` base class, and twelve concrete handler implementations.

4.2.4.1 Handler Contract

`RubricHandler<T>` defines four operations each handler must implement:

- `canProcess(String subRubric)`: determines if the handler is responsible for a given sub-rubric identifier.
- `parse(String xmlData)`: converts raw XML into a structured DTO of type `T`.
- `classify(T parsed)`: executes the classification step via the *Inference API*.
- `match(T parsed)`: executes the entity matching step via the *Inference API*.

Java generics ensure each handler operates on its own type-safe data model while conforming to a unified interface.

4.2.4.2 Shared Infrastructure

`AbstractRubricHandler<T>` provides the shared implementation used by all concrete handlers:

- **XML deserialization**: uses an `XmlMapper` to transform raw strings into the DTOs required by the handler.
- **API communication**: encapsulates the `RestClient` logic for POST requests to `/classification` and `/matching`.
- **Result interpretation**: provides a default method for interpreting matching responses; two handlers override this to accommodate sub-rubric-specific entity structures.

4.2.4.3 Concrete Implementations

Twelve concrete handlers each cover one or more sub-rubrics sharing the same XML structure and processing logic. `Hr02Handler` is the most complex, implementing both rule-based attribute mapping for classification and entity extraction for multiple company roles. All remaining handlers follow the same structure but operate on simpler payloads with fewer entity types.

4.2.5 Persistence Layer

The persistence layer implements the data model described in [Section 3.3](#) using Spring Data JPA. JPA entities map directly to the database tables, with Hibernate used as the ORM provider to bridge the object-oriented domain model and the relational schema. Each aggregate root is backed by a `JpaRepository` interface, keeping the service layer decoupled from SQL implementation details. Schema migrations are managed with Liquibase [\[44\]](#), ensuring consistent schema state across all environments. A dedicated state listener synchronizes `JobRunr` job execution states with the publication domain model, updating the corresponding publication record on each state transition.

4.2.6 REST API

The REST API is the primary interface for the *UI* to interact with *SHAB Monitoring*, as illustrated in [Section 3.1.3.3](#). While the enrichment pipeline runs as automated background jobs, the API provides endpoints to manually trigger ingestion and manage the lifecycle of processed publications and all the data for the *UI*. All communication uses JSON. The full API surface is documented via SpringDoc [\[45\]](#) and accessible through Swagger UI [\[46\]](#), [\[47\]](#).

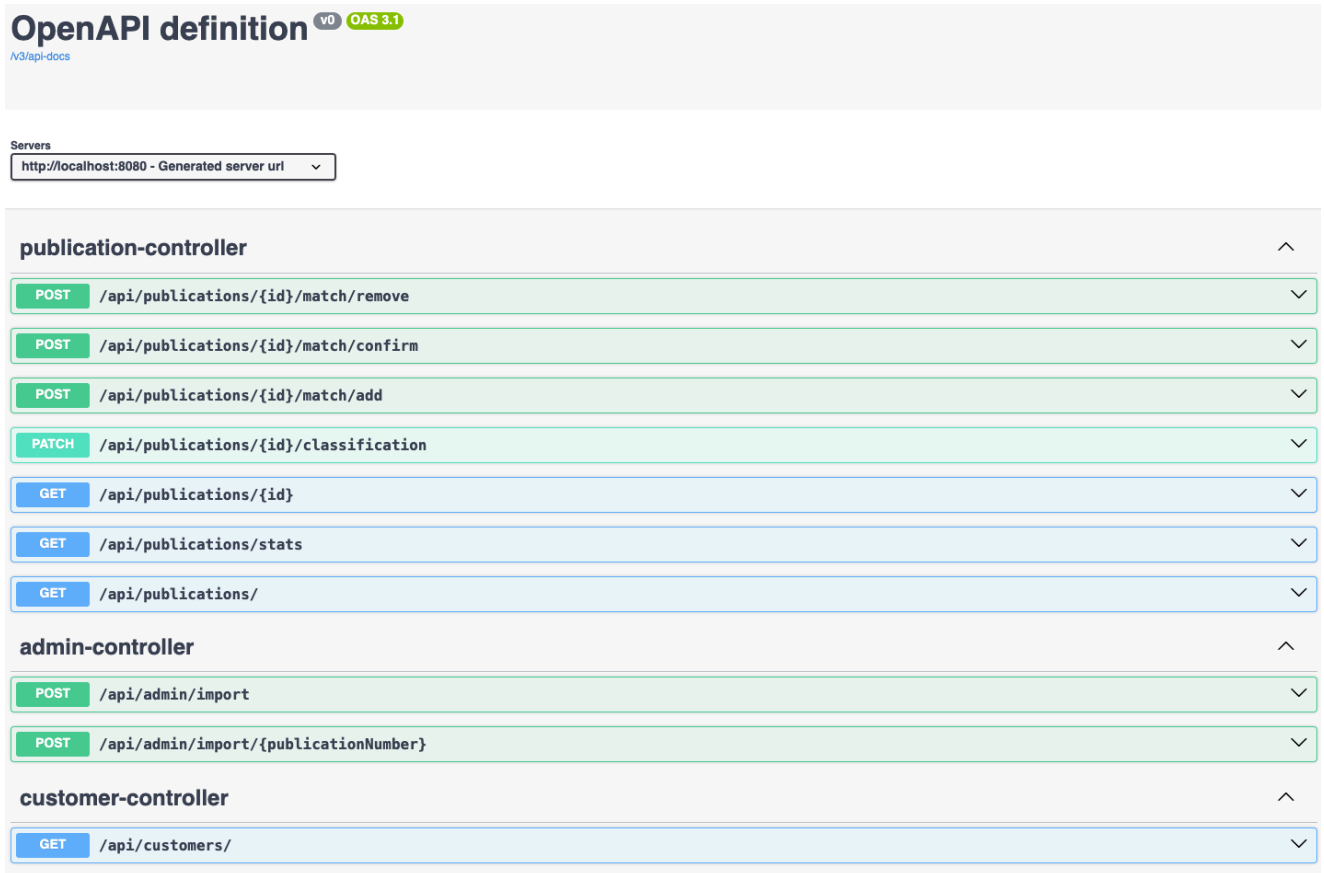


Figure 15: Swagger UI overview of the *SHAB Monitoring* REST API

4.2.6.1 Controller Organization

The API is organized into three controllers:

- **PublicationController**: manages the lifecycle and state of individual publications, including retrieval, statistics, match management (add, remove, confirm), and classification updates via PATCH requests.
- **AdminController**: exposes endpoints for manually triggering ingestion, either for a specific publication or a full day's worth of data.
- **CustomerController**: provides read access to the customer master data stored in the system.

4.2.6.2 Data Transformation and Error Handling

Domain entities are mapped to dedicated response DTOs, ensuring internal database structures are not exposed directly to the *UI*. Error handling is centralized in a `@ControllerAdvice` class, keeping controllers free of try/catch logic. `ShabMonitoringException` is mapped to an appropriate HTTP status code; unexpected exceptions are caught, logged, and returned as 500 Internal Server Error responses.

4.2.7 Notification System

Upon successful completion of the enrichment workflow, a notification email is sent to a configured recipient address. The email includes the publication title, sub-rubric, classification codes, and matched customer names, along with direct links to the *UI* and the original SHAB publication. This step corresponds to step [018] in [Figure 12](#).

4.3 DB Sync Tool

The DB Sync tool is a three-step pipeline that produces the DuckDB dataset consumed by the *Inference API* at runtime and by the matching model training tool ([Section 4.5](#)). Since Splink does not support MySQL as an execution backend (see [Section 3.2.6](#)), the customer dataset must be made available in DuckDB before matching can run. Since no real customer dataset was available for the PoC, the dataset is derived from publication entities already extracted by the system, as described in [Section 3.5.2.2](#) . Each step is implemented as a standalone Python script, allowing the pipeline to be run partially or the CSV output of step 1 to be replaced with a real customer export without modifying the remaining steps.

4.3.1 Step 1: Entity Extraction

`01-extract-publication-entities.py` reads the `publication_entities` table from MySQL in batches of 10'000 rows and splits the results by entity type into two CSV files: `persons.csv` and `companies.csv`. Duplicate records are eliminated using a SHA-256 hash of the key fields before the files are written.

4.3.2 Step 2: Customer Seeding

`02-seed-db-from-export.py` reads the two CSVs from step 1 and inserts the records into the `customers` table in MySQL. Field names are mapped from the CSV schema to the German column names used by the `customers` table (`vorname`, `nachname1`, `strassenbez`, etc.). Rows with oversized fields are skipped; all remaining rows are inserted in batches of 10'000. Using CSV as the intermediate format keeps the pipeline flexible: a real customer export can be substituted at this point without modifying the extraction or DuckDB creation steps.

4.3.3 Step 3: DuckDB Creation

`03-create-duckdb-from-sql.py` reads the `customers` table from MySQL and rebuilds the DuckDB database defined by `schema.sql`. Rows are split into `persons` (no `U_ID`) and `companies` (has `U_ID`) tables in batches of 10'000. Critically, customer IDs are preserved from MySQL into DuckDB, ensuring that match results returned by the *Inference API* can be resolved against the records stored in *SHAB Monitoring*. The `customers` table and DuckDB must be kept in sync in a production environment; in the PoC, the DB Sync tool is run manually after the initial dataset is seeded, and the MySQL `customers` table is not updated after seeding.

4.3.4 Available datasets

There are 2 preexisting datasets available for training and validation.

The `training-dataset` contains all entities from publications imported from 01.01.2025 to 15.05.2026, excluding publication of sub-rubric HR02 and classification that have been published on 19.03.2026, which are reserved for the `validation-dataset`.

The `validation-dataset` contains all entities from publications published on 19.03.2026, also without sub-rubric HR02 publications.

4.4 Splink Shared Package

The shared-matching-model package contains the reusable matching logic used by the Splink-based customer matching implementation. It provides functions for loading the prepared DuckDB customer dataset, normalizing records before comparison, constructing trainable Splink linkers, and loading serialized model definitions for inference.

4.4.1 Package Structure

The package uses a src layout and exposes the public API through `shared_matching_model/__init__.py`.

```
shared-matching-model/  
├── pyproject.toml  
└── src/shared_matching_model/  
    ├── __init__.py  
    ├── config.py  
    ├── linker_factory.py  
    ├── preprocessing.py  
    ├── repository.py  
    └── types.py
```

Listing 5: Simplified structure of the shared matching package

- `config.py` reads the DuckDB path, model paths, thresholds, and result limit from environment variables or defaults.
- `repository.py` loads either the persons or companies table from DuckDB and maps the database columns to the field names expected by Splink.
- `preprocessing.py` contains the shared record normalization function used before training and inference.
- `linker_factory.py` builds trainable linkers and loads saved linkers from JSON model files.
- `types.py` defines the supported entity types, person and company.

The package is built as a Python dependency named `shared-matching-model`. Both the matching model training tool ([Section 4.5](#)) and the matching service of the *Inference API* ([Section 4.6.4](#)) consume it as a dependency. The training tool uses it to load the dataset and produce the two model JSON files. The *Inference API* uses the same package to load the prepared dataset, load the saved model definitions, preprocess incoming entities, and apply the configured thresholds.

4.4.2 Preprocessing Function

The shared preprocessing function is `prepare_records(entity_type, df)`. It receives a Pandas data frame for either persons or companies and returns a normalized copy with the expected columns and values. As introduced in [Section 3.5.2.2](#), this function is applied in two places: when the DuckDB customer dataset is loaded and when an incoming entity is prepared at inference time.

The function performs the following operations:

- Adds missing expected columns with empty values, so incomplete incoming entities still match the model schema.
- Converts the `id` column to a stripped string representation.
- Converts all comparison fields to strings and strips surrounding whitespace.
- Uppercases postal codes for both entity types.
- Uppercases company UUIDs.
- Replaces empty strings, hyphens, NAN, and NONE with missing values.

The expected person fields are `id`, `first_name`, `last_name`, `birth_date`, `place_of_origin`, `address_street`, `address_house_number`, `address_zip_code`, and `address_town`. The expected company fields are `id`, `name`, `uid`, `address_street`, `address_house_number`, `address_zip_code`, and `address_town`.

4.4.3 Model Configuration

The shared package defines one model configuration per entity type. Both configurations use `id` as the unique ID column, and retain matching columns and intermediate calculation columns. The training linker is created by `build_trained_linker`, while the *Inference API* loads the saved JSON settings through `load_saved_linker`.

Runtime filtering is configured through `MatchingConfig`. The configured matching threshold is 0.85 for persons and companies, with the maximum number of returned candidates set to 5. These values provide solid results while not overly restricting the models. These values are supplied through environment variables. If no environment value is provided, the package falls back to its built-in defaults. Model and dataset locations are configured through `MATCHING_DUCKDB_PATH`, `MATCHING_PERSON_MODEL_PATH`, and `MATCHING_COMPANY_MODEL_PATH`, with defaults pointing to `customer.db`, `models/person_model.json`, and `models/company_model.json` relative to the configured artifact directory.

4.5 Matching Model Training Tool

The matching model training tool is a Python CLI that trains the customer matching models and serializes them as JSON files. It produces one model definition for persons and one for companies. These files are later mounted into the *Inference API* and loaded by the matching service.

The tool consumes the DuckDB customer dataset produced by the DB Sync tool ([Section 4.3](#)). Dataset loading, preprocessing, and model configuration are delegated to the shared package described in [Section 4.4](#) , so the training path uses the same fields and normalization rules as inference.

4.5.1 Input and Configuration

The CLI reads its default configuration through `load_matching_config`, using the tool directory as the base path. Without overrides, it expects the DuckDB file at `customer.db` and writes model files to the local `models/` directory.

The following command-line options are supported:

- `--entity-type`: selects person, company, or both. The default is both.
- `--db-path`: overrides the DuckDB input file.
- `--person-model-path`: overrides the output path for the person model JSON.
- `--company-model-path`: overrides the output path for the company model JSON.
- `--overwrite`: allows existing model JSON files to be replaced.

The actual dataset is loaded through `MatchingDatasetRepository`, which reads the selected DuckDB table and applies the shared preprocessing function before training begins.

4.5.2 Training Process

For each selected entity type, the tool loads the prepared dataset and calls `build_trained_linker` from the shared package. When the default both mode is used, this creates one trained linker for persons and one trained linker for companies.

The shared training routine performs three estimation steps per linker:

- Estimate the probability that two random records match, using a high-recall rule set for the selected entity type.
- Estimate non-match probabilities using random sampling.
- Estimate comparison parameters with expectation maximisation.

The person training path uses a prior rule based on date of birth, last name, and postal code, then runs expectation maximisation with date-of-birth blocking. The company training path uses matching non-empty UIDs as the prior rule, then runs expectation maximisation with UID blocking. The concrete comparison fields, prediction blocking rules, and thresholds are documented in [Section 4.4](#) .

Sparse training data can prevent individual estimation steps from determining all parameters. The implementation handles this by catching training-step exceptions and continuing with the remaining trained or default values, allowing a model file to still be produced for inference. While possible, this approach should be avoided in a production environment. The CLI does not calculate evaluation metrics during training.

4.5.3 Output

The tool writes the trained Splink settings as JSON files. With the default configuration, the outputs are:

- `models/person_model.json`
- `models/company_model.json`

These JSON files are part of the matching artifacts mounted into the *Inference API* deployment. At runtime, the matching service loads them through the shared package and applies them as described in [Section 4.6.4](#) .

4.6 Inference API (FastAPI)

The *Inference API* provides the machine learning capabilities required by the enrichment pipeline, as described in [Section 3.5](#). It is implemented as a stateless FastAPI service that receives a request, invokes the appropriate model or matching component, and returns the result. Workflow logic and persistence are handled entirely by *SHAB Monitoring*; the *Inference API* has no knowledge of publication state.

4.6.1 Package Structure

The application is organized around three areas: routes, DTOs, and services.

```

.
├── main.py
├── routes.py
├── dtos/
│   ├── classification/
│   └── matching/
├── services/
│   ├── classification_service.py
│   ├── matching_service.py
│   └── prompt.txt

```

Listing 6: Simplified package structure of the *Inference API*

- `main.py` creates the FastAPI application and registers the router.
- `routes.py` defines the HTTP endpoints and converts unexpected exceptions into HTTP 500 responses.
- The `dtos` package contains Pydantic models for the classification and matching JSON contracts.
- The `services` package contains the integration with the LLM and the Splink matching model.

4.6.2 HTTP API

The service exposes two endpoints as part of the application workflow.

The `/classification` endpoint accepts a `ClassificationRequest` with a single `publication_text` field and returns a `ClassificationResponse` containing a list of classification codes and a short justification (see [Listing 3](#) for an example).

The `/matching` endpoint accepts a `MatchingRequest` containing exactly one of two optional fields: `person` or `company`. This constraint is enforced by the Pydantic model before matching begins. The endpoint returns a `MatchingResponse` containing zero or more candidate matches, each with a customer ID, confidence score, `is_match` flag, and field-level explanation data (see [Listing 4](#) for an example).

An additional `/matching/debug` endpoint accepts the same request as `/matching` but also a `customer_id` query parameter. When provided, it runs the full comparison against that specific customer record and returns field-level scores regardless of the probability threshold, allowing developers to inspect why a particular customer was not returned as a candidate.

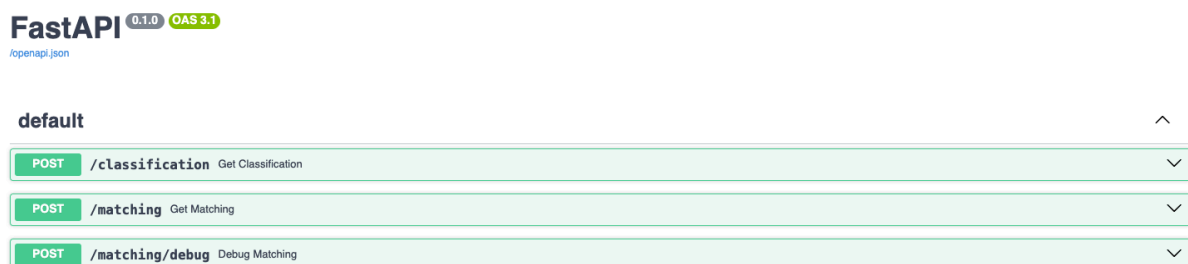


Figure 16: Swagger UI overview of the *Inference API*

4.6.3 Classification Service

The model is invoked through the Responses API with medium reasoning effort and a structured Pydantic response schema matching `ClassificationResponse`. The classification service loads the system prompt from `services/prompt.txt` and calls the configured model through the OpenAI Python client.

The prompt is structured into sections that guide the LLM from task understanding to a constrained classification result:

- **Task:** Defines the model's role as a precise classifier for SHAB publication description texts.
- **Input:** Clarifies that the model receives only the publication description and must not assume additional metadata.
- **Output:** Defines the required JSON shape with `classifications` and `reason`, matching the response schema used by the service.
- **Classification Rules:** Sets the general decision rules, including evidence-based classification, multilingual handling, and precision-oriented label selection.
- **Multi-Code Guidance:** Explains when multiple labels may be returned and how to avoid adding loosely related codes.
- **Allowed Classification Codes:** Restricts the model to the implemented label set listed in [Tab. 49](#).
- **Decision Boundaries:** Describes how to resolve common ambiguities between similar labels.
- **Unclassified Cases:** Defines the fallback behavior when the text does not clearly support any allowed code.

The service supports OpenAI models out of the box, but can also call an OpenAI-compatible local model endpoint by configuring `LOCAL_BASE_URL` and `LOCAL_MODEL`. The `IS_OPENAI_MODEL` flag selects the hosted OpenAI path, where the service uses structured parsing directly and does not set local sampling parameters. For local models, `MODEL_USE_PARSE` controls whether the service uses the parse endpoint or the create endpoint and then validates the returned JSON with Pydantic.

Param	Function
<code>temperature</code>	Controls randomness in token selection. Lower values make the output more deterministic.
<code>top_p</code>	Limits sampling to the most likely tokens whose cumulative probability reaches the configured value.
<code>top_k</code>	Limits sampling to the configured number of most likely next tokens.
<code>min_p</code>	Filters out tokens whose probability is too low relative to the most likely token.
<code>repetition_penalty</code>	Penalizes repeated tokens to reduce repetitive output.
<code>presence_penalty</code>	Penalizes already used tokens to encourage introducing new tokens.

Tab. 28: Local LLM Parameters

[Tab. 28](#) includes parameters that are used to make the local model more deterministic. If the local model response does not match the expected `ClassificationResponse` schema, the service retries with slightly higher temperatures (0.0, 0.1, 0.2) before failing the request. This retry behavior improves robustness for local models that sometimes return malformed JSON at fully deterministic settings.

4.6.4 Matching Service

The matching service dispatches incoming requests to the Splink linker corresponding to the entity type. Linker initialization, dataset loading, and preprocessing are delegated to the shared package, ensuring that inference requests and training data are normalized with the same rules before comparison.

Linkers are initialized lazily on the first request for each entity type, as described in [Section 3.2.7](#). Once initialized, Splink compares the incoming record against the customer dataset and filters results by the configured probability threshold before returning the top candidates.

Each candidate response includes field-level explanation data derived from Splink's waterfall analysis [Figure 17](#).

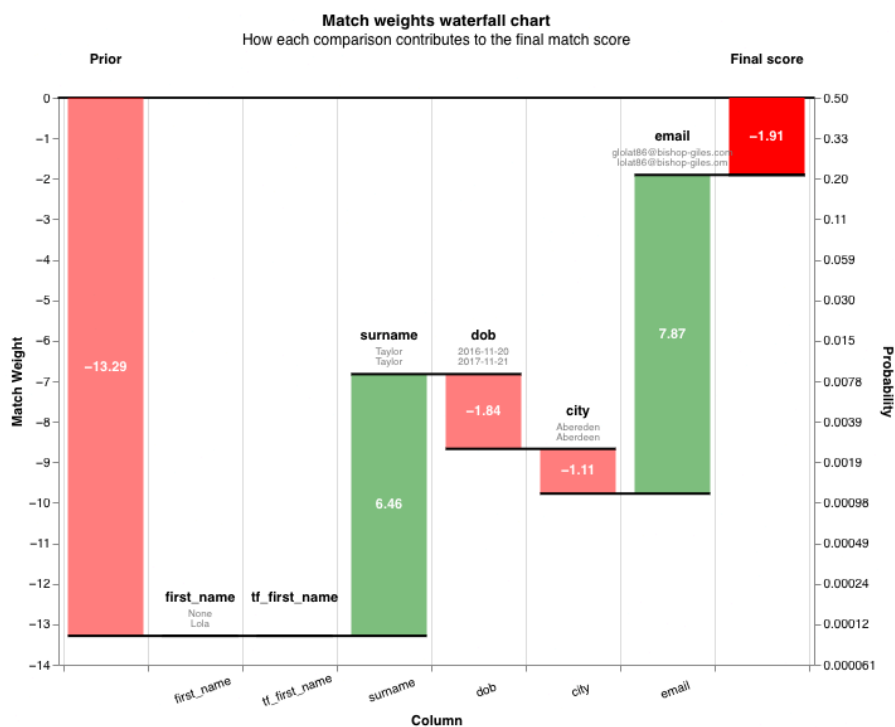


Figure 17: Example waterfall chart from Splinks official documentation [\[48\]](#)

The explanation data is stored as per-field log2 Bayes factors in `result_data` ([Listing 7](#)). This allows the *UI* to present a transparent breakdown of which fields contributed to a proposed match ([Figure 20](#)).

```
{
  "first_name": 5.828566246525742,
  "last_name": 5.711226371630047,
  "birth_date": 0.0,
  "place_of_origin": 0.0,
  "address_zip_code": 8.740147884665166,
  "address_town": 5.759389993081225,
  "address_street": 5.297960204518835,
  "address_house_number": 5.438008763872653
}
```

Listing 7: Example field-level explanation data for a person match candidate

4.7 User Interface

This section describes the *UI*, covering its scope, technologies, main views, and design.

4.7.1 Purpose and Scope

The *UI* gives bank employees visibility into the automated processing results and supports a limited set of write operations: confirming, adding, or removing customer matches, and correcting the classification when the automated result is incorrect. Follow-up actions beyond these operations are outside the project scope. All interactions are recorded in the audit trail.

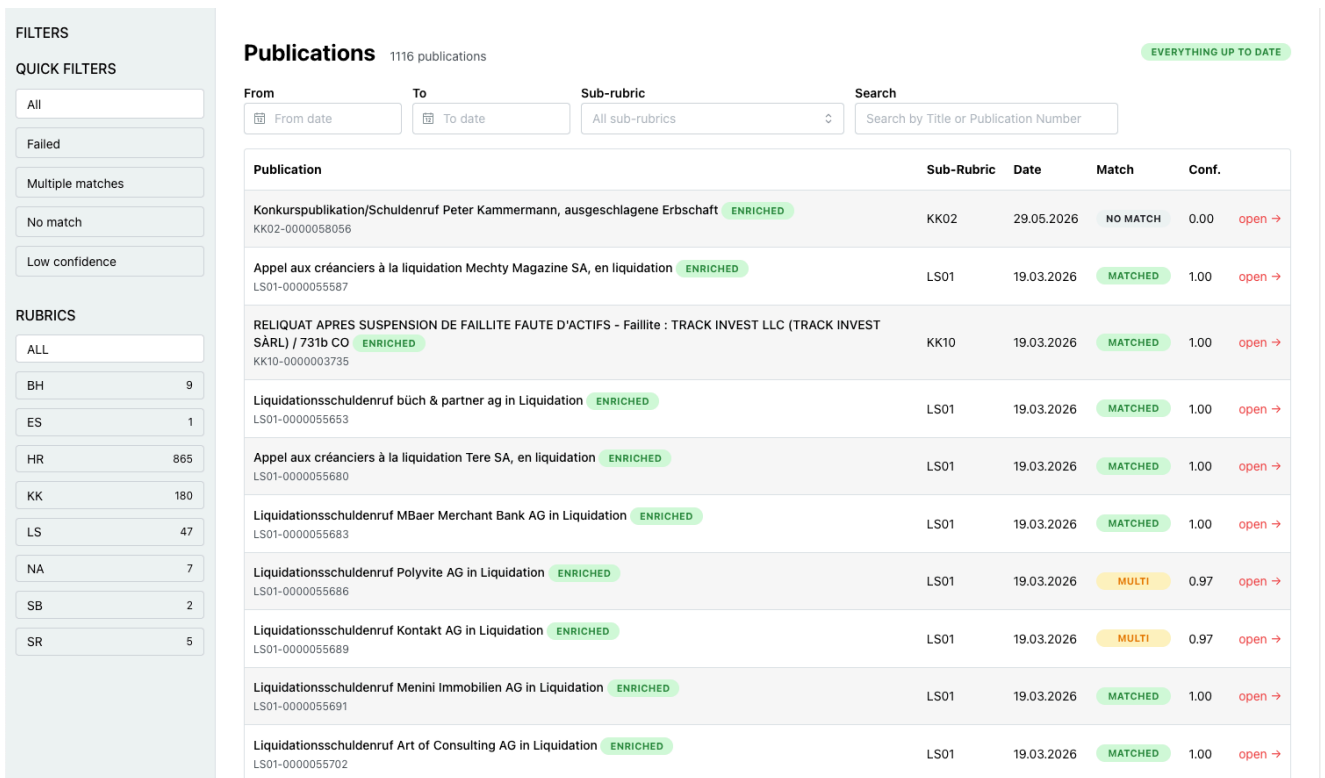
4.7.2 Technologies Used

The *UI* is built with React, using Vite as the build tool and TypeScript as the language. Mantine provides the component primitives required by the interface (inputs, tables, buttons, modals) within a consistent design system, using the official Mantine template as the starting point [49]. TanStack Query handles data fetching with built-in loading, caching, and error state management.

4.7.3 Views

The UI implements multiple views to handle Publications.

4.7.3.1 Publications View



Publications 1116 publications EVERYTHING UP TO DATE

From To Sub-rubric Search

Publication	Sub-Rubric	Date	Match	Conf.
Konkurspublikation/Schuldenruf Peter Kammermann, ausgeschlagene Erbschaft KK02-0000058056 ENRICHED	KK02	29.05.2026	NO MATCH	0.00 open →
Appel aux créanciers à la liquidation Mechty Magazine SA, en liquidation LS01-000005587 ENRICHED	LS01	19.03.2026	MATCHED	1.00 open →
RELIQUAT APRES SUSPENSION DE FAILLITE FAUTE D'ACTIFS - Faillite : TRACK INVEST LLC (TRACK INVEST SARL) / 731b CO KK10-0000003735 ENRICHED	KK10	19.03.2026	MATCHED	1.00 open →
Liquidationsschuldenruf büch & partner ag in Liquidation LS01-0000055653 ENRICHED	LS01	19.03.2026	MATCHED	1.00 open →
Appel aux créanciers à la liquidation Tere SA, en liquidation LS01-0000055680 ENRICHED	LS01	19.03.2026	MATCHED	1.00 open →
Liquidationsschuldenruf MBaer Merchant Bank AG in Liquidation LS01-0000055683 ENRICHED	LS01	19.03.2026	MATCHED	1.00 open →
Liquidationsschuldenruf Polyvite AG in Liquidation LS01-0000055686 ENRICHED	LS01	19.03.2026	MULTI	0.97 open →
Liquidationsschuldenruf Kontakt AG in Liquidation LS01-0000055689 ENRICHED	LS01	19.03.2026	MULTI	0.97 open →
Liquidationsschuldenruf Menini Immobilien AG in Liquidation LS01-0000055691 ENRICHED	LS01	19.03.2026	MATCHED	1.00 open →
Liquidationsschuldenruf Art of Consulting AG in Liquidation LS01-0000055702 ENRICHED	LS01	19.03.2026	MATCHED	1.00 open →

Figure 18: Publications View

The publications view ([Figure 18](#)) is the main view of the application, displaying the list of imported SHAB publications. Each entry shows the publication's metadata: title, processing status, publication ID, date, sub-rubric, and the automated matching result. The matching result is rendered as a colored badge indicating the outcome (pending, match, no match, multiple matches), alongside a numerical confidence score. The list is paginated at 50 entries per page, sorted by publication date in descending order.

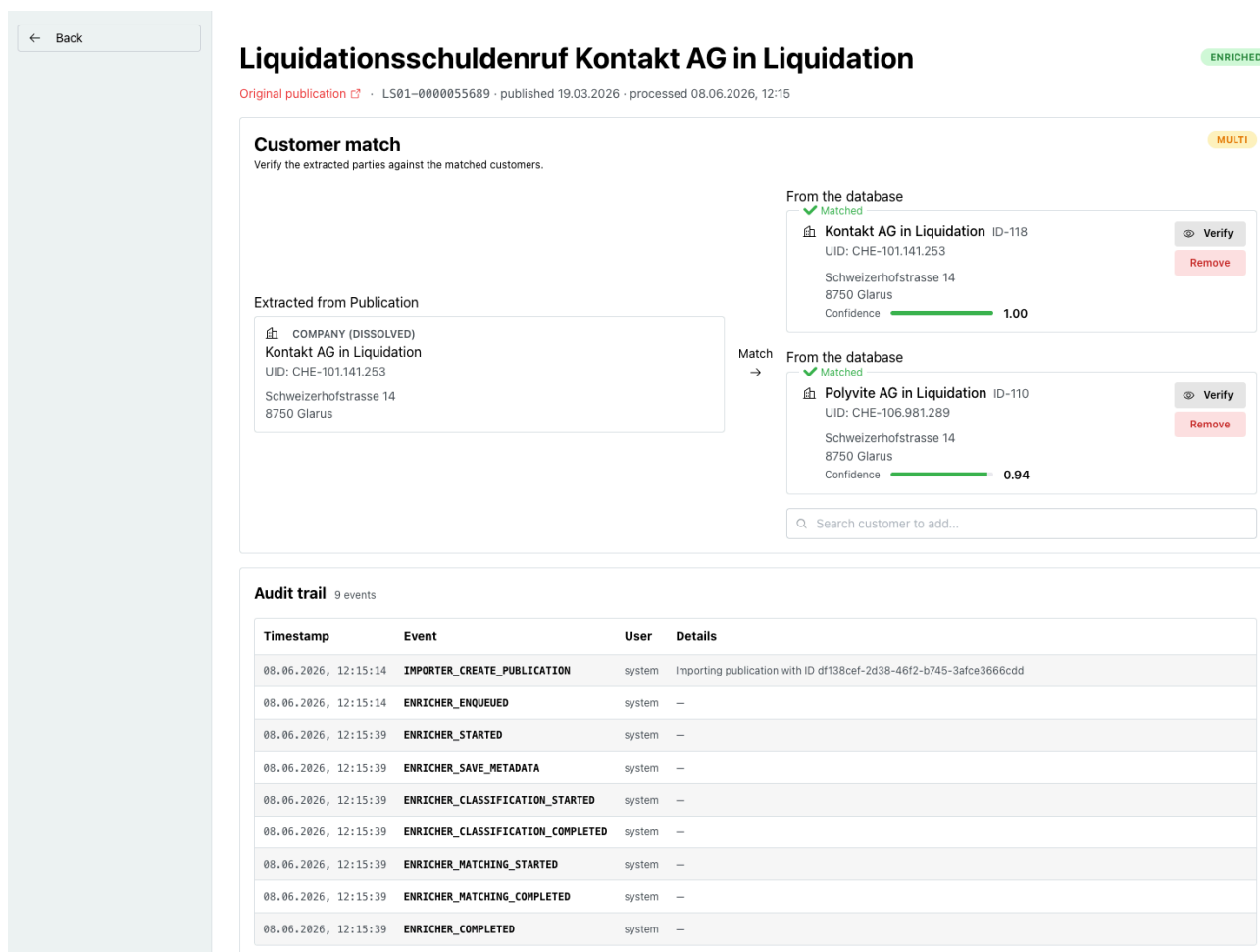
Filtering is supported through two complementary mechanisms. General filters apply to all entries and filter by date range and sub-rubric. Quick filters combine on top of the general filter to surface publications of interest:

- **All:** no additional filter applied.

- **Failed:** publications whose automated processing failed due to an unhandled edge case.
- **Multiple Matches:** publications where multiple customers were matched to a single entity, requiring manual review.
- **No Match yet:** publications that completed processing but produced no match.
- **Low Confidence:** publications whose match confidence is too low and needs manual review.

An indicator in the top right reflects the current import state: it shows the number of publications still pending processing, or “EVERYTHING UP TO DATE” when all imports have been processed.

4.7.3.2 Publication Detailed View



The screenshot displays the 'Publication Detailed View' for the entity 'Liquidationsschuldenruf Kontakt AG in Liquidation'. The interface includes a 'Back' button, a title bar, and a status indicator 'ENRICHED'. Below the title, it shows the publication ID 'LS01-0000055689', the publication date '19.03.2026', and the processing date '08.06.2026, 12:15'. The main section is titled 'Customer match' and contains a 'Verify the extracted parties against the matched customers.' instruction. It features a 'MULTI' tag and a 'Match' button. The 'Extracted from Publication' section shows the company name 'Kontakt AG in Liquidation', its UID 'CHE-101.141.253', and its address 'Schweizerhofstrasse 14, 8750 Glarus'. The 'From the database' section lists two matches: 'Kontakt AG in Liquidation' (ID-118, UID: CHE-101.141.253, Confidence: 1.00) and 'Polyvite AG in Liquidation' (ID-110, UID: CHE-106.981.289, Confidence: 0.94). Each match has a 'Verify' button and a 'Remove' button. An 'Audit trail' section at the bottom shows 9 events, including 'IMPORTER_CREATE_PUBLICATION', 'ENRICHER_ENQUEUED', 'ENRICHER_STARTED', 'ENRICHER_SAVE_METADATA', 'ENRICHER_CLASSIFICATION_STARTED', 'ENRICHER_CLASSIFICATION_COMPLETED', 'ENRICHER_MATCHING_STARTED', 'ENRICHER_MATCHING_COMPLETED', and 'ENRICHER_COMPLETED'.

Timestamp	Event	User	Details
08.06.2026, 12:15:14	IMPORTER_CREATE_PUBLICATION	system	Importing publication with ID df138cef-2d38-46f2-b745-3afce3666cdd
08.06.2026, 12:15:14	ENRICHER_ENQUEUED	system	—
08.06.2026, 12:15:39	ENRICHER_STARTED	system	—
08.06.2026, 12:15:39	ENRICHER_SAVE_METADATA	system	—
08.06.2026, 12:15:39	ENRICHER_CLASSIFICATION_STARTED	system	—
08.06.2026, 12:15:39	ENRICHER_CLASSIFICATION_COMPLETED	system	—
08.06.2026, 12:15:39	ENRICHER_MATCHING_STARTED	system	—
08.06.2026, 12:15:39	ENRICHER_MATCHING_COMPLETED	system	—
08.06.2026, 12:15:39	ENRICHER_COMPLETED	system	—

Figure 19: Publication Detailed View

The publication detailed view (Figure 19) is accessed by clicking a publication entry. It contains the information needed to perform follow-up actions and provides a link to the original SHAB publication for cross-referencing against the source.

The top section shows the customer matching result, enabling side-by-side comparison of the extracted entity and the matched customers. Each matched customer shows all attributes used for matching together with the confidence score.

Verify match · [redacted]
 Candidate C-70 · field-by-field comparison

COMPANY (ACQUIRING)

[redacted]

UID: [redacted]

Bernstrasse 19
3400 Burgdorf

100

Confidence Score

✓ Matched

[redacted] ID-70

UID: [redacted]

Bernstrasse 19
3400 Burgdorf

Confidence 1.00

Extracted from Publication

[redacted]

[redacted]

3400

Burgdorf

Bernstrasse

19

Company name

●●●

UID

●●●

ZIP

●●●

Town

●●●

Street

●●●

House N°

●●●

From the database

[redacted]

[redacted]

3400

Burgdorf

Bernstrasse

19

Figure 20: Publication Detailed View - Match Verification Modal

Clicking “Verify” opens a modal ([Figure 20](#)) showing all attributes side-by-side and their individual contribution to the overall confidence score. Green indicating a strong contribution, orange a neutral contribution and red a negative contribution towards the confidence score.

If a matched customer is incorrect, the employee can remove the match. Matches below the confidence threshold can be explicitly confirmed via a “Confirm” button. If no match or too few matches were found, the employee can search the customer database and add one manually.

For HR02 publications, an additional section displays the classification result and its reasoning, which the employee can correct manually [Figure 21](#) .

Classification HR02 matched by LLM + attributes

☒ **06** Handelsregistermeldung / Firma/Name

☐ **12** Handelsregistermeldung / Kapital

☐ **13** Handelsregistermeldung / Sitz

☒ **29** Handelsregistermeldung / Zweck HR

☐ **36** Handelsregistermeldung / Rechtsform

☐ **37** Handelsregistermeldung / Löschung

☐ **38** Handelsregistermeldung / Werdegang / Fusion

☐ **49** Handelsregistermeldung / Amtliche Meldungen

☐ **54** Handelsregistermeldung / Vernetzung Entscheider (Unterschriftenwechsel)

☐ **55** Handelsregistermeldung / Vernetzung / weitere Zeichnungsberechtigte (Unterschriftenwechsel)

☐ **56** Handelsregistermeldung / Revisionsstelle

☐ **73** Handelsregistermeldung / Domizil / Adresse Handelsregister

☐ **80** Handelsregistermeldung / Amtliche Meldung Inhaber

reason: The text explicitly states 'Firma neu' (new company name) and 'Zweck neu' (new purpose), indicating changes to the firm name and business purpose.

Figure 21: Publication Detailed View - Classification Section

The last section contains the audit trail, listing all automated events and manual user actions for the publication [Figure 19](#) .

4.7.4 Design

The visual design follows the Mantine default design system with minor color customizations. Phosphor icons [50] are used to indicate details such as customer type. Initial layout exploration used AI-generated mockups as starting points; the final design iterated on those during implementation. For fast interaction feedback, the *UI* uses optimistic updates when confirming, adding, or removing matches, immediately reflecting changes while the backend operation completes.

4.7.5 Limitations

Several aspects of the *UI* are intentionally simplified within the PoC scope:

- Authentication is not implemented.

- The interface targets desktop use only, with no mobile optimization.
- Error handling is present but user-facing feedback is minimal.
- Correctness was verified through manual testing as documented in [Section 4.9.2](#) .

4.8 Deployment

The system is distributed as a Docker Compose stack for local deployment, as motivated in [Section 3.2.4](#). A production deployment is not prescribed, as infrastructure requirements vary depending on the target environment. Each of the three application components (*SHAB Monitoring*, *Inference API*, and *UI*) is packaged as a Docker image and published to the OST GitLab registry.

4.8.1 CI/CD

Each application component is built and published automatically via GitLab CI/CD. A pipeline is triggered on every push that affects the respective service directory or the CI configuration file. The pipeline builds a Docker image and publishes it to the OST GitLab container registry, tagged with the commit SHA and `latest` if the push happens on the main branch. This ensures that the images referenced in the Docker Compose stack are always up to date with the latest commit on the main branch.

4.8.2 Services

The stack consists of six services:

- **shab-monitoring**: the Spring Boot backend ([Section 4.2](#)).
- **shab-ui**: the frontend ([Section 4.7](#)).
- **inference-api**: the FastAPI service, accessible only within the internal network ([Section 4.6](#)).
- **db**: MySQL 9.6.0, used by `shab-monitoring`; data is persisted in a named volume.
- **mailpit**: a local SMTP mock that captures notification emails without forwarding them to real recipients.
- **nginx**: reverse proxy that routes incoming requests to the appropriate service based on subdomain and path.

All services communicate over an internal Docker network. The `inference-api` mounts a local `models/` directory containing the trained Splink model files and the DuckDB customer dataset produced by the DB Sync tool ([Section 4.3](#)) and the matching model training tool ([Section 4.5](#)).

4.8.3 Routing

Nginx routes requests based on subdomain:

Domain	Target	Description
<code>localtest.me</code>	<code>shab-ui:80</code>	Frontend application
<code>localtest.me/api/</code>	<code>shab-monitoring:8080</code>	REST API
<code>jobrunr.localtest.me</code>	<code>shab-monitoring:8000</code>	JobRunr dashboard
<code>mail.localtest.me</code>	<code>mailpit:8025</code>	Local mail interface

Tab. 29: nginx routing configuration

`localtest.me` is a public wildcard DNS that resolves all subdomains to `127.0.0.1`, requiring no local DNS configuration [\[51\]](#).

4.8.4 Configuration

All services are configured through a shared `.env` file. A `.env.example` is provided with all required variables and placeholder values. Image versions are controlled through version-tagged variables (`FRONTEND_VERSION`, `SHAB_MONITORING_VERSION`, `INFERENCE_API_VERSION`), defaulting to `latest` if not set.

4.9 Testing

The application consists of multiple modules, each tested according to its role and complexity. Automated tests are executed in CI/CD on every commit that affects the respective service.

4.9.1 SHAB Monitoring Testing

SHAB Monitoring is tested with a combination of unit and integration tests. Integration tests are used where database interactions are involved; pure logic without database dependencies is covered by unit tests. The following tools and frameworks are used:

- **JUnit 5** for test structure and parameterized tests [52]
- **Testcontainers** with a MySQL container for isolated database testing [53]
- **WireMock** (wiremock-spring) for mocking external HTTP dependencies [54]
- **Mockito** for mocking internal Spring beans [55]
- **AssertJ** for fluent assertions [56]
- **JaCoCo** for code coverage measurement, with a minimum instruction coverage of 95% enforced as a build requirement [57]

Code formatting is enforced via Spotless [58] and checked as part of every CI build, which also executes a unit test run.

A total of 343 tests were executed across 25 test classes with no failures, errors, or skipped tests. As shown in [Tab. 30](#), instruction coverage of 99% was achieved, exceeding the enforced minimum of 95%.

Metric	Total	Missed	Coverage
Instructions	5'366	33	99%
Branches	343	14	95%
Lines	1'221	5	99%
Methods	261	5	98%
Classes	70	0	100%

Tab. 30: JaCoCo code coverage summary

4.9.2 UI Testing

The *UI* is verified through a guided manual test rather than automated tests. Test cases are defined beforehand and executed by a team member, with expected and actual results documented.

4.9.2.1 Pre-conditions and Test Data

All test cases share the following pre-conditions. A full import for the date: 19.03.2026 must have been executed successfully. The following individual publications must also be present: KK02-0000058056 and HR02-1006588285. Customer data must be loaded from the validation-dataset ([Section 4.3.4](#)). Use the db sync tool and execute steps 2 and 3 to load the dataset into MySQL and DuckDB.

4.9.2.2 Test Cases

Test cases have been verified on June 8th, 2026, using the latest state on the main branch.

ID	TC-01
Description	Date range filter
Precondition	User is on the Home page and global pre-conditions met Section 4.9.2.1
Steps	1. Select from date picker: 2026-03-01 2. Select to date picker: 2026-03-06
Expected Result	• One publication with publication Number HR02-1006588285 is visible in the publication list. • The publication count shows 1 publication.
Actual Result	As expected

Tab. 31: TC-01: Date range filter

ID	TC-02
Description	Sub-rubric and sidebar rubric filter
Precondition	User is on the Home page and global pre-conditions met Section 4.9.2.1
Steps	1. Click KK in the sidebar rubric list. 2. Remove the KK filter by clicking All under Rubrics. 3. Select HR02/06 from the sub-rubric filter in the top filter bar.
Expected Result	• After step 1: the topmost publication is KK02-0000058056. • After step 2: the full publication list is restored. • After step 3: only publications with sub-rubric HR02 and classification 06 are visible in the list.
Actual Result	As expected

Tab. 32: TC-02: Sub-rubric and sidebar rubric filter

ID	TC-03
Description	Text search
Precondition	User is on the Home page and global pre-conditions met Section 4.9.2.1
Steps	1. Enter HR02-1006588285 in the search field. 2. Clear the search field. 3. Enter Mutation Freeart GmbH, Bassersdorf, neu Freeart Trading GmbH in the search field.
Expected Result	• After step 1: only the publication with number HR02-1006588285 is visible. • After step 3: only the publication with title "Mutation Freeart GmbH, Bassersdorf, neu Freeart Trading GmbH" is visible.
Actual Result	As expected

Tab. 33: TC-03: Text search

ID	TC-04
Description	Quick filters
Precondition	User is on the Home page and global pre-conditions met Section 4.9.2.1
Steps	1. Click Failed in the quick filters. 2. Click Multiple matches in the quick filters. 3. Click No match in the quick filters. 4. Click Low confidence in the quick filters.
Expected Result	• After step 1: only publications with a failed processing status are visible. • After step 2: only publications displaying the MULTI badge are visible. • After step 3: only publications displaying the NO MATCH badge are visible. • After step 4: only publications with a manually to confirm match is visible.
Actual Result	As expected. Though there was no publication with low confidence so list was empty after step 4.

Tab. 34: TC-04: Quick filters

ID	TC-05
Description	Pagination
Precondition	User is on the Home page and global pre-conditions met Section 4.9.2.1
Steps	1. Note the publications visible on page 1. 2. Click the next page button.
Expected Result	• Page 2 is loaded and displays a different set of publications than page 1.
Actual Result	As expected

Tab. 35: TC-05: Pagination

ID	TC-06
Description	Metadata display, original publication link and Enriched badge
Precondition	User is on the Home page and global pre-conditions met Section 4.9.2.1
Steps	1. Navigate to the detail page of publication HR02-1006588285. 2. Click the Original publication link.
Expected Result	• After step 1: title shows “Mutation Freeart GmbH, Bassersdorf, neu Freeart Trading GmbH”, publication number shows HR02-1006588285, published date shows 06.03.2026, and the status badge shows ENRICHED. • After step 2: the SHAB portal opens in a new tab showing the correct publication.
Actual Result	As expected

Tab. 36: TC-06: Metadata display, original publication link and **Enriched** badge

ID	TC-07
Description	Audit trail
Precondition	User is on the detail page of publication HR02-1006588285
Steps	1. Scroll to the audit trail section.
Expected Result	• The following events are displayed in order: IMPORTER_CREATE_PUBLICATION ENRICHER_ENQUEUED ENRICHER_STARTED ENRICHER_SAVE_METADATA ENRICHER_CLASSIFICATION_STARTED ENRICHER_CLASSIFICATION_COMPLETED ENRICHER_MATCHING_STARTED ENRICHER_MATCHING_COMPLETED ENRICHER_COMPLETED • All events show system as the user. • ENRICHER_CLASSIFICATION_COMPLETED shows the classification details in the details column.
Actual Result	As expected

Tab. 37: TC-07: Audit trail

ID	TC-08
Description	Matched customer display, verify, remove and add match
Precondition	User is on the detail page of publication SR01-0000015707
Steps	1. Scroll to the Customer match section. 2. Click Verify on the ProBAS AG match. 3. Close the verification modal. 4. Click Remove on the ProBAS AG match. 5. Search for ProBAS in the search field and add the match.
Expected Result	• After step 1: two entities are displayed. ProBAS AG (UID: CHE-215.651.779) is matched to ID-67 and Infraagon Ingenieure AG (UID: CHE-454.870.814) is matched to ID-70. Both show Matched status with Verify and Remove buttons. • After step 2: a modal opens showing a field-by-field comparison between the extracted entity and the matched customer. • After step 4: the ProBAS AG match is removed and the section updates accordingly. The audit trail contains a PUBLICATION_MATCHING_CHANGED event with details “Removed match with customer id: 67”. • After step 5: ProBAS AG is matched again, the match displays a Manually Matched badge, and the audit trail contains a PUBLICATION_MATCHING_CHANGED event with details “Added match with customer id: 67”.
Actual Result	As expected

Tab. 38: TC-08: Matched customer display, verify, remove and add match

ID	TC-09
Description	Status badge update from NO MATCH to MATCHED
Precondition	User is on the detail page of publication HR02-1006601194
Steps	1. Scroll to the Customer match section. 2. Search for ProBAS in the search field of the first entity and add the match.
Expected Result	• After step 1: the status badge shows NO MATCH and both entities display “No candidates found”. • After step 2: the match is added and displays a Manually Matched badge and the customer match section status badge updates to MATCHED.
Actual Result	As expected

Tab. 39: TC-09: Status badge update from NO MATCH to MATCHED

ID	TC-10
Description	Classification update
Precondition	User is on the detail page of publication HR02-1006601194
Steps	1. Scroll to the Classification section. 2. Click the checkbox next to the 06 classification to add it. 3. Click the checkbox next to the 06 classification again to remove it.
Expected Result	• After step 2: Checkbox 06 is checked and the audit trail contains an ENRICHER_CLASSIFICATION_CHANGED event with details “Updated classification. Added: 06. Removed: “. • After step 3: Checkbox 06 is unchecked and the audit trail contains an ENRICHER_CLASSIFICATION_CHANGED event with details “Updated classification. Added: . Removed: 06.”.
Actual Result	As expected

Tab. 40: TC-10: Classification update

4.9.3 Inference API Testing

The *Inference API* is intentionally thin: it validates requests, invokes the classification or matching model, and returns typed responses. It owns no business logic or publication state. Correct integration is verified indirectly through the enrichment workflow and the *UI* views that display returned values, which is successful on the latest commit on the main branch. Missing or malformed values in these views indicate that the service or its integration requires further inspection. The predictive quality of the classification and matching models is evaluated separately in [Section 5](#).

5 Machine Learning Evaluation

This chapter evaluates the quality of the machine learning parts of the system and measures how well the models perform on their intended tasks. The two evaluated tasks are classification of HR02 publication texts and customer matching for persons and companies. The matching evaluation relies on a synthetic customer dataset, described in [Section 5.2](#).

Both evaluations follow the same general procedure. First, a dataset with input examples and expected outputs is prepared. Second, the corresponding endpoint of the *Inference API* is called for each input. Third, the predicted output is compared with the expected output and summarized with quantitative metrics.

Step	Description
1	Run the model on the evaluation dataset.
2	Compare the prediction with the expected output for each entry.
3	Calculate precision, recall, F1 score, and confusion matrices where applicable.

Tab. 41: General ML evaluation procedure

Precision describes the fraction of predicted outputs that are correct. Recall describes the fraction of expected outputs that were found. The F1 score is the harmonic mean of precision and recall. Confusion matrices are used to inspect where predictions differ from the expected labels.

5.1 Classification Evaluation

This chapter documents the evaluation of the classification part of the system.

5.1.1 Preparation

The classification model was evaluated using a manually labeled dataset of 130 HR02 publications imported on 19.03.2026, the same date reserved for evaluation across the project. The sample was drawn randomly, with language distribution reflecting the natural occurrence of German, French, and Italian across SHAB publications overall ([Figure 3](#)).

Each entry contains the publication text and the expected list of classification codes. Labels were created using a command-line helper that displayed the publication text alongside the allowed code list and stored the selected labels as JSON. The first 20 publications were classified together with the external project partner to calibrate the annotation approach. The remaining publications were then independently labeled by both team members and merged; where classifications differed, they were discussed and reconciled into a final label.

5.1.2 Processing

The evaluation imports each publication and compares the added codes with the expected labels. Because the task is multi-label classification, the expected and predicted labels are binarized before calculating precision, recall, and F1 scores. In addition to exact set equality, the evaluation also reports coverage: a prediction is counted as covered when all expected labels are present, even if additional labels are returned. This metric is relevant for the monitoring workflow because overclassification can be corrected by users, whereas an omitted label may hide a relevant event. The evaluation artifacts include a JSON classification report, confusion matrices, metric plots, and timing information.

5.1.3 Results

Figure 22 shows the macro results of Precision, F1 and Recall for each model. The macro values already provide a good overview of the performance differences.

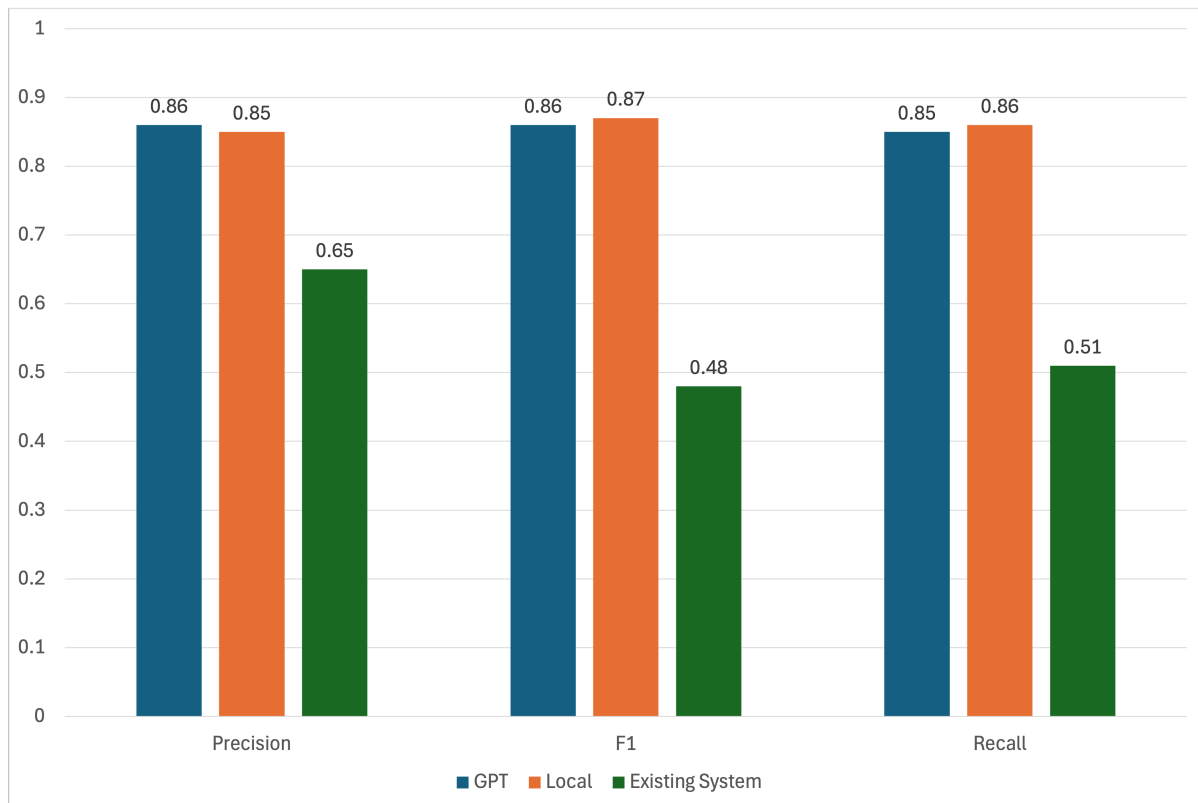


Figure 22: Evaluation - Classification - Combined Macro Statistics

The performance gap becomes especially visible in the coverage metrics shown in Tab. 42 : the external service covers all expected labels in only 33.33% of the evaluated publications, while the implemented system covers 85.38% for GPT. For the intended monitoring workflow this is problematic because missing labels reduce the chance that relevant customer events are surfaced.

The local classification run used gemma-4-31B-it-GGUF [59] as the LLM component in the full SHAB Monitoring pipeline. It achieves the best reported aggregate metrics, with micro and weighted F1 scores of 0.93. Its exact-match rate is 83.08%, compared with 81.54% for the OpenAI run, and its covered-match rate is 87.69%, compared with 83.08%.

Classifier	Samples	Micro F1	Weighted F1	Exact match	Covered match
Existing System	129	0.65	0.60	26.36%	33.33%
GPT	130	0.93	0.92	81.54%	83.08%
Local	130	0.93	0.93	83.08%	87.69%

Tab. 42: Classification comparison on labels including structured SHAB attributes

5.1.4 Classification Error Distribution

The classification report gives the aggregate performance per code, while a confusion matrix helps inspect which labels are confused with each other. Because the task is multi-label, the confusion matrix is generated from the binarized expected and predicted label vectors used for the classification report.



Figure 23: Evaluation - Classification - GPT Confusion Matrix

The missing and extra rows and columns are added because this is a multi-label evaluation, not a standard single-label confusion matrix. Missing represents expected labels that were not predicted, while extra represents predicted labels that were not part of the expected label set.

The confusion matrix shows that the model mostly predicts the frequent classes correctly. The visible errors are mainly missing labels and additional labels rather than systematic swaps between two specific classes. This fits the multi-label nature of HR02 texts: many publications contain several independent changes, and the main risk is not choosing one wrong class, but omitting or adding a second event code. Classes with very small support, especially 38 and 80, cannot be interpreted reliably from this evaluation set.

If we compare this to the confusion matrix of the existing system ([Figure 24](#)) we can see that the existing system makes much more mistakes, especially in categories 13 and 49.



Figure 24: Evaluation - Classification - Existing System Confusion Matrix

5.2 Synthetic Customer Dataset

No real customer data was available for the PoC. To enable both model training and evaluation, a synthetic dataset was derived from publication entities extracted from historical SHAB publications. Since SHAB publications contain structured person and company data in the same format the matching model processes at runtime, these entities serve as a realistic proxy for customer master data in the absence of real customer records.

For the evaluation dataset, 66 records per entity type were taken from these CSV files as the base for generating positive examples. This number corresponds to the selected evaluation CSV size and keeps the person and company evaluation sets balanced. Together with two generated perturbations per base record, it yields 198 positive examples per entity type, which is close to a 200-case evaluation set while keeping local LLM generation effort manageable. Two perturbed variants were generated per record using a locally hosted gemma-4-E4B-it-GGUF [60] model, prompted to introduce realistic noise: typos, abbreviations, missing fields, and OCR-style mistakes [61], a technique shown to produce controllable and diverse synthetic data.

Invalid perturbations such as malformed UUIDs were automatically rejected and retried until a valid result was produced. Each of the 66 base records produces one original and two perturbed variants, yielding 198 positive examples per entity type, each retaining the original customer ID as the expected match target.

Negative examples consist of 30 fully synthetic records per entity type, generated using the Faker library [62] with a Swiss German locale. Person negatives include a randomly generated name, date of birth, place of origin, and address. Company negatives include a randomly generated name, a random but structurally valid CHE UID, and a Swiss address. None of these records correspond to any entry in the customer dataset.

5.3 Customer Matching Evaluation

The customer matching model was evaluated separately for persons and companies using the synthetic dataset. Each evaluation entry contains the input entity, the expected customer ID, and a boolean indicating whether a match is expected, making this a binary match/no-match task. For positive examples, the prediction is correct when the expected customer ID is returned as a match. For negative examples, when no match is returned.

The person evaluation contains 228 entries: 198 positive and 30 negative examples. At the current threshold of 0.85, the model returns no matches for all 30 negative examples and identifies 181 of the 198 positive examples, resulting in an accuracy of 0.93 and an F1 score of 0.96 for the match class (Tab. 43).

Class	Precision	Recall	F1	Support
No match	0.64	1.00	0.78	30
Match	1.00	0.91	0.96	198
Accuracy	-	-	0.93	228

Tab. 43: Person matching evaluation metrics

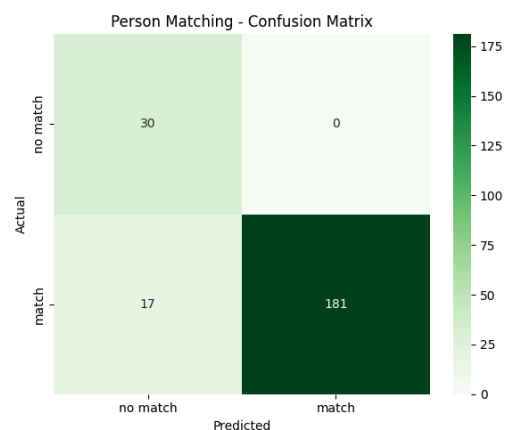
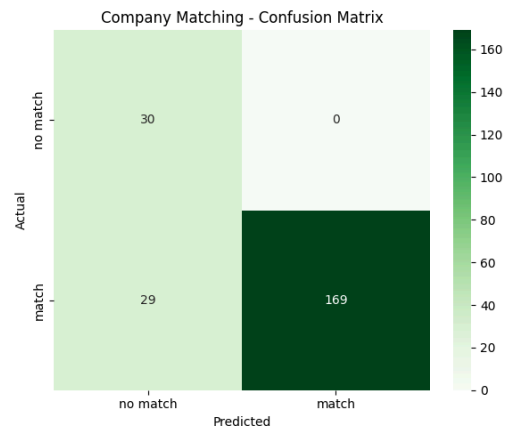


Figure 25: Matching - Person Model - Confusion Matrix

The company evaluation contains the same number of entries. At the same threshold, the model again returns no matches for all 30 negative examples and identifies 169 of the 198 positive examples, resulting in an accuracy of 0.87 and an F1 score of 0.92 for the match class (Tab. 44).

Class	Precision	Recall	F1	Support
No match	0.51	1.00	0.67	30
Match	1.00	0.85	0.92	198
Accuracy	-	-	0.87	228

Tab. 44: Company matching evaluation metrics**Figure 26:** Matching - Company Model - Confusion Matrix

5.3.1 Threshold Sensitivity

The matching endpoint filters candidates by a configurable probability threshold. The reported production oriented configuration uses a threshold of 0.85. To understand the precision-recall trade-off, the evaluation was also run at lower and higher thresholds and compared separately for persons and companies. Lower thresholds are expected to increase recall by returning more candidates, while potentially reducing precision by accepting weaker matches.

Entity type	Threshold	Accuracy	Match precision	Match recall	Match F1
Person	0.10	0.96	1.00	0.95	0.97
Person	0.35	0.94	1.00	0.93	0.97
Person	0.65	0.93	1.00	0.92	0.96
Person	0.70	0.93	1.00	0.92	0.96
Person	0.75	0.93	1.00	0.91	0.96
Person	0.85	0.93	1.00	0.91	0.96
Person	0.90	0.92	1.00	0.90	0.95
Company	0.10	0.90	1.00	0.89	0.94
Company	0.35	0.90	1.00	0.88	0.94
Company	0.65	0.89	1.00	0.87	0.93
Company	0.70	0.88	1.00	0.86	0.93
Company	0.75	0.88	1.00	0.86	0.93
Company	0.85	0.87	1.00	0.85	0.92
Company	0.90	0.87	1.00	0.85	0.92

Tab. 45: Threshold sensitivity for the match class

The threshold tests shows an unusual result: the reported quality increases when the threshold is lowered, with the best aggregate metrics at 0.10. This does not necessarily mean that 0.10 is the best operational threshold. Across all evaluated thresholds, the generated negative examples produce zero false positives. As a result, lowering the threshold only improves recall on the positive examples and does not reveal the expected precision cost.

The most likely reason is that the negative set is too easy: the 30 Faker-generated negative records per entity type are not close enough to real customer records to act as hard negatives. For production tuning, the threshold should therefore be selected with a broader validation set containing near matches, outdated addresses, similar company names, duplicate-looking persons, and records with partially shared identifiers.

An operational threshold should be chosen with the customer's coverage first preference in mind, but validated against harder negative examples.

5.3.2 False Negative Analysis

At the current threshold of 0.85, the model misses 17 of the 198 positive person examples and 29 of the 198 positive company examples. The false-negative debug output records the field-level log2 Bayes factors for these cases. For persons, the strongest recurring negative signal is the postal code, while first name, last name, and town often still contribute positive evidence. This indicates that perturbed address fields can dominate otherwise plausible person matches. For companies, the UID and postal code are the strongest recurring negative signals. The company model still receives positive evidence from company name, town, and street in many missed cases, but a changed or missing UID together with postal-code disagreement is enough to keep the match below the threshold.

Entity type	False negatives	Observed pattern
Person	17	Postal-code disagreement is the strongest recurring negative signal; name and town fields often remain positive.
Company	29	UID and postal-code disagreement dominate; company name, town, and street often still provide positive evidence.

Tab. 46: False negative pattern analysis

5.4 Cost Analysis

5.4.1 OpenAI API Costs

Classification requests are sent to the gpt-5.4-mini model via the OpenAI API. The cost per call depends on the number of input tokens, which consists of a fixed system prompt and the variable publication text.

The system prompt is fixed at 2'087 tokens. The publication text contribution was measured across a representative sample of 450 HR02 publications, yielding a mean of 140 tokens (95% CI ± 9), a 95th percentile of 352 tokens, and a range of 52 to 735 tokens. The mean input per call is therefore 2'227 tokens. The expected output is estimated at 120 tokens, covering the structured JSON response with classification codes and justifications.

At the current API pricing of \$0.75 per million input tokens and \$4.50 per million output tokens [61], the cost per classification call is approximately \$0.00174.

5.4.2 Estimated Production Cost

Based on the observed daily HR02 volume of approximately 1'050 publications (Section 1.4.4) and assuming all publications trigger LLM inference, the table Tab. 47 shows the estimated cost across model tiers [63]. This represents a worst-case estimate; in practice, publications where the structured attributes already provide a complete classification do not trigger LLM inference, so actual costs would be lower.

Model	Input (\$/1M)	Output (\$/1M)	Per call	Daily	Monthly	Yearly
gpt-5.4-nano	\$0.20	\$1.25	\$0.00046	\$0.49	\$14.62	\$177.90
gpt-5.4-mini	\$0.75	\$4.50	\$0.00174	\$1.82	\$54.74	\$666.06
gpt-5.4	\$2.50	\$15.00	\$0.00579	\$6.08	\$182.48	\$2'220.20
gpt-5.5	\$5.00	\$30.00	\$0.01159	\$12.17	\$364.96	\$4'440.39

Tab. 47: Estimated classification cost by model tier (1'050 daily publications, worst case)

6 Results

The project resulted in a working PoC for monitoring SHAB publications with deterministic data processing, LLM-based classification, and probabilistic customer matching. The implemented system covers the core workflow from publication import to enrichment, review, and manual correction.

6.1 Implemented System

SHAB Monitoring implements the central application workflow. It imports SHAB publications, retrieves detail data, extracts relevant entities, persists processing state, and records audit trail events for the enrichment process. The enrichment workflow integrates the *Inference API* for classification and customer matching while keeping persistence, retry handling, and user-facing state in the main application.

The *Inference API* provides the two machine learning capabilities required by the workflow. The classification endpoint assigns HR02 labels to publication texts, while the matching endpoint compares extracted publication entities against the prepared customer dataset. The customer matching implementation is supported by the DB Sync tool, the shared Splink package, and the matching model training tool, which together produce the DuckDB dataset and saved model artifacts required at runtime.

The React *UI* provides the review layer for the PoC. It includes an overview of imported publications, detailed publication pages, classification results, customer match candidates, field-level matching explanations, and manual correction actions. As a result, automatic enrichment is not treated as final state, users can inspect and correct the generated results.

Overall this setup fulfills all requirements ([Section 2](#)) defined in the beginning of this thesis.

6.2 Classification Results

The HR02 classification evaluation ([Section 5.1](#)) shows that the LLM-based classifier is usable for supporting the monitoring workflow. The final OpenAI-based evaluation reaches a micro F1 score and weighted F1 score of approximately 0.92. In the three-way comparison, both LLM-based variants clearly outperform the current existing classification system, especially on exact-match and covered-match metrics ([Tab. 42](#)).

The strongest results are achieved for labels with more examples in the evaluation set, such as company-name changes, purpose changes, signing-authority changes, auditor changes, and domicile changes. Results for rare labels must be interpreted carefully, because some classes occur only once in the evaluation dataset. The classification result is therefore sufficient for the PoC workflow, but the measured quality is more reliable for frequent HR02 event types.

6.3 Customer Matching Results

The customer matching evaluation ([Section 5.3](#)) shows that both matching models are usable for the monitoring workflow. The person model reaches an accuracy of 0.93 and a match-class F1 score of 0.96 ([Tab. 43](#)). The company model reaches an accuracy of 0.87 and a match-class F1 score of 0.92 ([Tab. 44](#)).

Neither evaluation run produced false positives but the company model has lower recall than the person model, which is visible in the higher number of false negatives described in [Tab. 46](#) .

6.4 PoC Boundaries

The implemented system demonstrates the intended architecture and workflow, but it remains a PoC. The customer dataset used for matching is synthetic, the matching models are trained and evaluated on generated variants, and the DB synchronization and model training steps are not automated as continuous production processes. The evaluation therefore demonstrates feasibility and relative model behavior, not final production quality.

The UI and backend support manual verification, which is important for the current maturity level. Automatic classification and matching reduce manual effort, but the system is designed so that users can inspect, confirm, correct, or remove generated enrichment results before relying on them operationally.

7 Conclusions & Outlook

The project shows that SHAB publication monitoring can be supported by combining deterministic processing with targeted machine learning components. The deterministic parts provide the stable workflow: importing publications, extracting structured entities, persisting processing state, and exposing review functionality. The machine learning parts add capabilities that are difficult to implement reliably with fixed rules alone: multi-label HR02 classification and probabilistic customer matching.

The separation between *SHAB Monitoring* and the *Inference API* proved useful for the PoC. It keeps the main application independent from Python-specific ML dependencies while allowing classification and matching components to evolve separately. At the same time, the integration remains transparent because classifications, match candidates, confidence values, and field-level matching explanations are surfaced to the user instead of being hidden behind an automatic decision.

The PoC addresses the main disadvantages of the existing third-party system described in [Section 1.1](#). By retrieving SHAB publications directly, the system removes the overnight processing dependency and makes publication data available as soon as it is imported. Because the enrichment workflow runs inside the implemented application, the industry partner is no longer dependent on the external provider's availability for classification and matching. The prototype also improves transparency: classifications, matching candidates, confidence values, field-level explanations, and manual corrections are stored and visible in the application instead of being delivered as opaque external results. Recurring provider costs are not present in the PoC architecture, although production operation would still introduce infrastructure, maintenance, and LLM usage costs.

7.1 Limitations

The main limitation is the available data. The classification evaluation is based on a manually labeled HR02 evaluation day, which gives useful feedback but does not cover all labels equally. Rare labels remain difficult to assess because their support in the evaluation set is very small.

The customer matching evaluation is constrained by the absence of a real customer dataset. The synthetic customer data and generated perturbations make evaluation possible, but they cannot fully represent the distribution of real customer master data, outdated records, duplicate-looking customers, or near matches. This also affects threshold selection: the generated negative examples were not difficult enough to expose a realistic precision-recall trade-off.

The operational setup is still PoC-grade. DB synchronization, model training, and model artifact deployment are implemented as separate tooling rather than a fully automated production pipeline. The system therefore demonstrates the workflow and architecture, but additional hardening would be required before operational use.

7.2 Outlook

Further improvement should focus first on turning the PoC workflow into a controlled multi-user review process. The current application demonstrates the enrichment and correction flow, but does not yet model organizational responsibility for publications.

7.2.1 User Management

Future versions should add user management, assignment of publications to individual users or user groups, and review states that distinguish pending manual review from completed or archived publications. The UI should then expose these states through filters and views, allowing users to focus on publications that still require action.

7.2.2 Audit Log

Audit logging should also be extended from system-level processing events to user actions. Manual confirmation, correction, removal, or assignment decisions should record the acting username. This would make the review process traceable and provide a reliable basis for later analysis of manual interventions.

7.2.3 Confirm Publications for continuous Dataset improvement

Confirmed publications can become an important feedback source. After a user checks a publication and confirms the final classification and matching state, the confirmed result can be stored as labeled data. These confirmations would allow the system to measure precision and recall against real workflow decisions over time and would provide training and validation data for future model improvements.

7.2.4 Customer Matching

For customer matching, the most important next step is training and evaluating on real production customer data. A real dataset would make the matching evaluation more meaningful, especially if it includes hard negative examples such as similar company names, changed addresses, partially matching UIDs, and duplicate-looking persons. The matching model can then be improved by increasing recall without sacrificing precision. Possible measures include threshold tuning, stronger company-name normalization, additional preprocessing for address variants, and a broader validation set with realistic near matches. The company model should receive particular attention because it currently misses more positive examples than the person model.

7.2.5 Classification

The classification task would benefit from clearer and more operationally precise classification codes. Some HR02 labels are semantically close, which makes both manual labeling and LLM-based classification harder. Sharper label definitions, additional examples per code, and explicit rules for ambiguous cases would reduce uncertainty and make the evaluation data more consistent.

A longer-term alternative to prompt-only classification is a fully custom machine learning loop for HR02 classification. Such a pipeline would start by collecting confirmed publication texts and their final labels from the review workflow. The texts would then be embedded with a text embedding model, producing numerical representations that capture the semantic content of each publication. Those embeddings, optionally combined with structured publication attributes, would be used to train a supervised multi-label classification model. The trained classifier would predict one or more HR02 codes for new publication texts, and its predictions would be evaluated against a held-out validation set before deployment. User confirmations from production could then be fed back into the dataset, making the model improvement process repeatable.

7.2.6 Production Setup

For production readiness, the supporting processes should be automated. This includes scheduled synchronization of customer data into DuckDB, the creation of a snapshot before midnight to have a last valid state, repeatable model training, versioned model artifacts, deployment checks, and monitoring of classification and matching outcomes over time. This part could also include a health check functionality which is able to track the processed publications and the amount of publications that were released to achieve a self-repair effect if a day of publications should have been missed for some reason. The full deployment setup should also include production logging, authentication, authorization, notification delivery, backup and recovery procedures, and operational monitoring.

Chapter II

Glossary and List of Abbreviations

ADR	Architecture Decision Record
DB	Database
DTO	Data Transfer Object
EM	Expectation Maximisation
FR	Functional Requirement
GGUF	GPT-Generated Unified Format
JPA	Java Persistence API
LLM	Large language model
ML	Machine Learning
OCR	Optical Character Recognition
ORM	Object-Relational Mapping
PoC	Proof of Concept
PRL	Probabilistic Record Linkage
RUP	Rational Unified Process
SHAB	Schweizerisches Handelsamtsblatt (en.: Swiss Official Gazette of Commerce)
SECO	State Secretariat for Economic Affairs
SMTP	Simple Mail Transfer Protocol
UID	Swiss business identification number

Chapter III

Bibliography

- [1] "Schweizerisches Handelsamtsblatt (SHAB)." Accessed: June 10, 2026. [Online]. Available: <https://www.shab.ch/>
- [2] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," 2019. Accessed: June 05, 2026. [Online]. Available: <https://arxiv.org/abs/1810.04805>
- [3] T. B. Brown *et al.*, "Language Models are Few-Shot Learners," 2020. Accessed: June 05, 2026. [Online]. Available: <https://arxiv.org/abs/2005.14165>
- [4] J. Wei *et al.*, "Finetuned Language Models Are Zero-Shot Learners," 2021. Accessed: June 05, 2026. [Online]. Available: <https://arxiv.org/abs/2109.01652>
- [5] "OpenAI - Structured Output." Accessed: June 05, 2026. [Online]. Available: <https://developers.openai.com/api/docs/guides/structured-outputs>
- [6] "OpenAI - Providing Relevant Context." Accessed: June 05, 2026. [Online]. Available: <https://developers.openai.com/api/docs/guides/prompt-engineering#include-relevant-context-information>
- [7] I. P. Fellegi and A. B. Sunter, "A Theory for Record Linkage," *Journal of the American Statistical Association*, vol. 64, no. 328, pp. 1183–1210, 1969, doi: [10.1080/01621459.1969.10501049](https://doi.org/10.1080/01621459.1969.10501049).
- [8] P. Konda *et al.*, "Magellan: Toward Building Entity Matching Management Systems over Data Science Stacks," *Proceedings of the VLDB Endowment*, vol. 9, no. 13, pp. 1581–1584, 2016, Accessed: June 05, 2026. [Online]. Available: <https://dl.acm.org/doi/10.14778/2994509.2994535>
- [9] S. Mudgal *et al.*, "Deep Learning for Entity Matching: A Design Space Exploration," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 19–34. doi: [10.1145/3183713.3196926](https://doi.org/10.1145/3183713.3196926).
- [10] R. Peeters, A. Steiner, and C. Bizer, "Entity Matching using Large Language Models," 2023. Accessed: June 05, 2026. [Online]. Available: <https://arxiv.org/abs/2310.11244>
- [11] T. Wang *et al.*, "Match, Compare, or Select? An Investigation of Large Language Models for Entity Matching," 2024. Accessed: June 05, 2026. [Online]. Available: <https://arxiv.org/abs/2405.16884>
- [12] "Seco - Schweizerisches Handelsamtsblatt." Accessed: May 21, 2026. [Online]. Available: <https://www.seco.admin.ch/seco/de/home/wirtschaftslage---wirtschaftspolitik/periodika/schweizerisches-handelsamtsblatt--shab-.html>
- [13] "SHAB API Documentation." Accessed: June 10, 2026. [Online]. Available: <https://www.amtsblattportal.ch/docs/api/>
- [14] "SHAB-Publication LS08-0000000271: Liquidationsschuldenruf Wasserversorgungs-Genossenschaft Vogelsang-Lengnau." Accessed: June 02, 2026. [Online]. Available: <https://www.shab.ch/#!/search/publications/detail/79d5fd1c-152b-4fb8-b4a3-0cfb7966488e>
- [15] "SHAB-Publikation HR02-1006601515: Mutation Schweizerischer KMU Verein (SKV), Naters." Accessed: Mar. 19, 2026. [Online]. Available: <https://www.shab.ch/#!/search/publications/detail/b41f63f2-70f3-414e-a916-5da31b768cca>
- [16] "C4 model." Accessed: Feb. 27, 2026. [Online]. Available: <https://c4model.com/>
- [17] "IcePanel." Accessed: Feb. 27, 2026. [Online]. Available: <https://icepanel.io/>
- [18] "ADR - Y Statement." Accessed: June 01, 2026. [Online]. Available: <https://adr.github.io/adr-templates/#y-statement>
- [19] Stack Overflow, "Stack Overflow Developer Survey 2025." [Online]. Available: <https://survey.stackoverflow.co/2025/>
- [20] G. Hohpe and B. Woolf, *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions*. Addison-Wesley, 2003.
- [21] M. Fowler, "What do you mean by 'Event-Driven'?", Feb. 07, 2017. Accessed: May 13, 2026. [Online]. Available: <https://martinfowler.com/articles/201701-event-driven.html>
- [22] C. Richardson, *Microservices Patterns: With Examples in Java*. 2018.
- [23] "Scheduling Tasks." Accessed: Feb. 26, 2026. [Online]. Available: <https://spring.io/guides/gs/scheduling-tasks>

- [24] "Quartz Quick Start Guide." Accessed: Feb. 26, 2026. [Online]. Available: <https://www.quartz-scheduler.org/documentation/quartz-2.4.x/quick-start.html>
- [25] "Jobrunr Configurations." Accessed: Feb. 26, 2026. [Online]. Available: <https://www.jobrunr.io/en/documentation/configuration/>
- [26] "Quartz - How to Setup Databases." Accessed: Feb. 26, 2026. [Online]. Available: <https://github.com/quartz-scheduler/quartz/wiki/How-to-Setup-Databases>
- [27] "Jobrunr Storage." Accessed: Feb. 26, 2026. [Online]. Available: <https://www.jobrunr.io/en/documentation/installation/storage/>
- [28] "Jobrunr - Monitoring Jobs via the Dashboard." Accessed: Feb. 26, 2026. [Online]. Available: <https://www.jobrunr.io/en/documentation/#monitoring-jobs-via-the-dashboard>
- [29] "Quartz - Example 6 - Dealing with Job Exceptions." Accessed: Feb. 26, 2026. [Online]. Available: <https://www.quartz-scheduler.org/documentation/quartz-2.5.x/examples/Example6.html>
- [30] "Jobrunr - Dealing with exceptions." Accessed: Feb. 26, 2026. [Online]. Available: <https://www.jobrunr.io/en/documentation/background-methods/dealing-with-exceptions/>
- [31] "Quartz - Lesson 5 - SimpleTrigger." Accessed: Feb. 26, 2026. [Online]. Available: <https://www.quartz-scheduler.org/documentation/quartz-2.5.x/tutorials/tutorial-lesson-05.html>
- [32] "Dedupe." [Online]. Available: <https://dedupe.io/>
- [33] "Python Record Linkage Toolkit." [Online]. Available: <https://recordlinkage.readthedocs.io/en/latest/about.html>
- [34] "Python Record Linkage Toolkit - Performance." [Online]. Available: <https://recordlinkage.readthedocs.io/en/latest/performance.html#performance>
- [35] "Splink - Key Features." Accessed: Mar. 15, 2026. [Online]. Available: <https://moj-analytical-services.github.io/splink/index.html?h=unsupervised#key-features>
- [36] "Splink - DuckDB support." Accessed: Mar. 15, 2026. [Online]. Available: https://moj-analytical-services.github.io/splink/topic_guides/splink_fundamentals/backends/backends.html#duckdb
- [37] "DuckDB - In-Process Database Management System." Accessed: Mar. 15, 2026. [Online]. Available: <https://duckdb.org/docs/lts/guides/glossary#in-process-database-management-system>
- [38] "DuckDB - Data Sources." Accessed: Mar. 15, 2026. [Online]. Available: https://duckdb.org/docs/current/data/data_sources
- [39] "Lazy Loading." Accessed: Mar. 15, 2026. [Online]. Available: <https://www.imperva.com/learn/performance/lazy-loading/>
- [40] "Python Packaging User Guide." Accessed: Mar. 15, 2026. [Online]. Available: <https://packaging.python.org/en/latest/tutorials/packaging-projects/>
- [41] "Amtsblattportal - API shared structure." Accessed: Feb. 19, 2026. [Online]. Available: https://www.amtsblattportal.ch/docs/api/#_content_data_attributes
- [42] "Refactoring Guru - Strategy Pattern." Accessed: May 22, 2026. [Online]. Available: <https://refactoring.guru/design-patterns/strategy>
- [43] "SHAB-Publication HR02-1006660991: Mutation F.S Food GmbH, Wildhaus-Alt St. Johann." Accessed: May 28, 2026. [Online]. Available: <https://www.shab.ch/#!/search/publications/detail/ca532729-3bc8-47d8-a137-8bdc84e6c33d>
- [44] "Liquibase." Accessed: June 10, 2026. [Online]. Available: <https://www.liquibase.com/>
- [45] "SpringDoc OpenAPI - Library for Spring Boot." Accessed: Mar. 29, 2026. [Online]. Available: <https://springdoc.org/>
- [46] "Swagger UI." Accessed: Mar. 29, 2026. [Online]. Available: <https://swagger.io/tools/swagger-ui>
- [47] "SHAB Monitoring OpenAPI Definition." [Online]. Available: <http://localhost:8080/swagger-ui/index.html>
- [48] "Splink - Waterfall Chart." Accessed: June 10, 2026. [Online]. Available: https://moj-analytical-services.github.io/splink/charts/waterfall_chart.html
- [49] "Mantine template." Accessed: Apr. 28, 2026. [Online]. Available: <https://github.com/mantinedev/vite-vanilla-extract-template/>

- [50] "Phosphor Icons." Accessed: Apr. 28, 2026. [Online]. Available: <https://phosphoricons.com/>
- [51] "Introducing Testing Domain - localtest.me." Accessed: May 31, 2026. [Online]. Available: <https://weblogs.asp.net/owscott/introducing-testing-domain-localtest-me/>
- [52] "JUnit 5 - The Next Generation of JUnit." [Online]. Available: <https://junit.org/>
- [53] "Testcontainers." [Online]. Available: <https://testcontainers.com/>
- [54] "WireMock." [Online]. Available: <https://wiremock.org/>
- [55] "Mockito." [Online]. Available: <https://site.mockito.org/>
- [56] "AssertJ." [Online]. Available: <https://assertj.github.io/doc/>
- [57] "JaCoCo." [Online]. Available: <https://www.jacoco.org/>
- [58] "GitHub - Spotless: Keep your code spotless." Accessed: May 23, 2026. [Online]. Available: <https://github.com/diffplug/spotless>
- [59] "Gemma 4 model card." Accessed: June 05, 2026. [Online]. Available: <https://huggingface.co/unsloth/gemma-4-31B-it-GGUF>
- [60] "Gemma 4 model card." Accessed: June 05, 2026. [Online]. Available: <https://huggingface.co/unsloth/gemma-4-E4B-it-GGUF>
- [61] L. Long *et al.*, "On LLMs-Driven Synthetic Data Generation, Curation, and Evaluation: A Survey," 2024. Accessed: Mar. 15, 2026. [Online]. Available: <https://arxiv.org/abs/2406.15126>
- [62] "Faker." Accessed: Apr. 19, 2026. [Online]. Available: <https://github.com/joke2k/faker>
- [63] OpenAI, "OpenAI pricing." Accessed: June 09, 2026. [Online]. Available: <https://developers.openai.com/api/docs/pricing>

Chapter IV

List of Illustrations

Figure 1	Daily publication distribution per calendar week 2026 (Min/Max/Median)	4
Figure 2	Rubric distribution across imported publications	4
Figure 3	Language distribution across imported publications	5
Figure 4	C4 - Context Diagram	11
Figure 5	C4 - Container Diagram	12
Figure 6	C4 - Component Diagram - SHAB Monitoring	13
Figure 7	C4 - Component Diagram - Inference API	13
Figure 8	C4 - Component Diagram - User Interface	14
Figure 9	SHAB Monitoring - Database Schema	21
Figure 10	Inference API - Database Schema	22
Figure 11	Sequence diagram - Importer Job	27
Figure 12	Sequence diagram - Enrichment Job	29
Figure 13	Sequence diagram - Classification	30
Figure 14	Sequence diagram - Customer Matching	31
Figure 15	Swagger UI overview of the <i>SHAB Monitoring</i> REST API	36
Figure 16	Swagger UI overview of the <i>Inference API</i>	41
Figure 17	Example waterfall chart from Splinks official documentation [48]	43
Figure 18	Publications View	44
Figure 19	Publication Detailed View	45
Figure 20	Publication Detailed View - Match Verification Modal	46
Figure 21	Publication Detailed View - Classification Section	46
Figure 22	Evaluation - Classification - Combined Macro Statistics	54
Figure 23	Evaluation - Classification - GPT Confusion Matrix	55
Figure 24	Evaluation - Classification - Existing System Confusion Matrix	56
Figure 25	Matching - Person Model - Confusion Matrix	57

Figure 26 Matching - Company Model - Confusion Matrix 58

Chapter V

List of Tables

Tab. 1	SHAB publication volume by day of week (observation period: 12.05.2026 to 02.06.2026)	3
Tab. 2	FR-1: Persistent storage of original and extracted publication data	6
Tab. 3	FR-2: Traceability of publication ingestion time	6
Tab. 4	FR-3: Transparency of ingestion status and error cases	6
Tab. 5	FR-4: Manual re-execution of ingestion processes	6
Tab. 6	FR-5: Automated publication classification	6
Tab. 7	FR-6: Explainability of classification decisions	6
Tab. 8	FR-7: Manual correction of classification results	7
Tab. 9	FR-8: Automated customer assignment process	7
Tab. 10	FR-9: Explainability of customer matching decisions	7
Tab. 11	FR-10: Visibility of unmatched/ overmatched publications	7
Tab. 12	FR-11: Availability of matching confidence scores	7
Tab. 13	FR-12: Manual customer assignment capability	7
Tab. 14	FR-13: User notification of required customer-related actions	7
Tab. 15	FR-14: Auditability of automated and manual decisions	8
Tab. 16	FR-15: Export capability for quality and performance metrics	8
Tab. 17	FR-16: Filtering of historical decision records	8
Tab. 18	NFR-1: Performance - Publication processing	9
Tab. 19	NFR-2: Reliability - Publication classification accuracy	9
Tab. 20	NFR-3: Reliability - Customer matching accuracy	9
Tab. 21	NFR-4: Reliability - System recovery	10
Tab. 22	NFR-5: Maintainability - Code quality measures	10
Tab. 23	Decision Matrix - Job Scheduling	17
Tab. 24	Publication processing states	23
Tab. 25	Idempotency keys used by the <i>Importer-Job</i> and <i>Enrichment-Job</i>	24

Tab. 26 Audit trail event types	24
Tab. 27 Parameters for SHAB API bulk publications endpoint	33
Tab. 28 Local LLM Parameters	42
Tab. 29 nginx routing configuration	48
Tab. 30 JaCoCo code coverage summary	49
Tab. 31 TC-01: Date range filter	49
Tab. 32 TC-02: Sub-rubric and sidebar rubric filter	50
Tab. 33 TC-03: Text search	50
Tab. 34 TC-04: Quick filters	50
Tab. 35 TC-05: Pagination	50
Tab. 36 TC-06: Metadata display, original publication link and Enriched badge	51
Tab. 37 TC-07: Audit trail	51
Tab. 38 TC-08: Matched customer display, verify, remove and add match	51
Tab. 39 TC-09: Status badge update from NO MATCH to MATCHED	52
Tab. 40 TC-10: Classification update	52
Tab. 41 General ML evaluation procedure	53
Tab. 42 Classification comparison on labels including structured SHAB attributes	54
Tab. 43 Person matching evaluation metrics	57
Tab. 44 Company matching evaluation metrics	58
Tab. 45 Threshold sensitivity for the match class	58
Tab. 46 False negative pattern analysis	59
Tab. 47 Estimated classification cost by model tier (1'050 daily publications, worst case)	59
Tab. 48 SHAB Sub-rubrics	72
Tab. 49 HR02 Classification Codes	73
Tab. 50 List of aids	74

Chapter VI

Code-Listings

Listing 1	Simplified package structure of <i>SHAB Monitoring</i>	32
Listing 2	SHAB API bulk response DTOs	33
Listing 3	Classification request and response: HR02-1006601515 <u>[15]</u>	34
Listing 4	Matching request and response: HR02-1006601515 <u>[15]</u>	34
Listing 5	Simplified structure of the shared matching package	38
Listing 6	Simplified package structure of the <i>Inference API</i>	41
Listing 7	Example field-level explanation data for a person match candidate	43

Chapter VII

Appendix

8 SHAB Sub-rubrics

Code	Description
BH00	Aufforderung nach Handelsregisterverordnung (HregV)
ES03	Rechnungsruf zur Aufnahme des öffentlichen Inventars
HR02	Handelsregistermeldung
HR02 / 06	Handelsregistermeldung / Firma/Name
HR02 / 12	Handelsregistermeldung / Kapital
HR02 / 13	Handelsregistermeldung / Sitz
HR02 / 29	Handelsregistermeldung / Zweck HR
HR02 / 36	Handelsregistermeldung / Rechtsform
HR02 / 37	Handelsregistermeldung / Löschung
HR02 / 38	Handelsregistermeldung / Werdegang / Fusion
HR02 / 49	Handelsregistermeldung / Amtliche Meldungen
HR02 / 54	Handelsregistermeldung / Vernetzung Entscheider (Unterschriftenwechsel)
HR02 / 55	Handelsregistermeldung / Vernetzung / weitere Zeichnungsberechtigte (Unterschriftenwechsel)
HR02 / 56	Handelsregistermeldung / Revisionsstelle
HR02 / 73	Handelsregistermeldung / Domizil / Adresse Handelsregister
HR02 / 80	Handelsregistermeldung / Amtliche Meldung Inhaber
KK01	Vorläufige Konkursanzeige
KK02	Konkurspublikation / Schuldenruf
KK03	Einstellung des Konkursverfahren
KK04	Kollokationsplan und Inventar
KK05	Verteilungsliste und Schlussrechnung
KK06	Schluss des Konkursverfahrens
KK07	Widerruf des Konkurses
KK08	Konkursamtliche Grundstücksteigerung
KK09	Lastenverzeichnisse
KK10	Verschiedenes / Spezialliquidation
LS01	Liquidationsschuldenruf einer Aktiengesellschaft
LS02	Liquidationsschuldenruf einer Gesellschaft mit beschränkter Haftung
LS03	Liquidationsschuldenruf einer Genossenschaft
LS04	Liquidationsschuldenruf eines Vereins
LS05	Liquidationsschuldenruf einer Stiftung
LS06	Liquidationsschuldenruf einer Kollektivgesellschaft
LS07	Liquidationsschuldenruf einer Kommanditgesellschaft
LS08	Liquidationsschuldenruf einer Kommanditaktiengesellschaft/anderer Rechtsform
NA01	Provisorische Nachlassstundung
NA02	Definitive Nachlassstundung

NA03	Schuldenruf im Nachlassverfahren
NA04	Verlängerung der Nachlassstundung
NA05	Konkurs im Nachlassverfahren
NA06	Verhandlung über die Bestätigung des Nachlassvertrages
NA07	Bestätigung des Nachlassvertrages
NA08	Aufhebung der Nachlassstundung infolge Sanierung
NA09	Kollokationsplan im Nachlassverfahren mit Vermögensabtretung
NA10	Rechenschaftsbericht im Nachlassverfahren mit Vermögensabtretung
NA11	Verschiedenes
NA12	COVID-19-Stundung
SB01	Betreibungsamtliche Grundstücksteigerung
SR01	Schuldenruf infolge Fusion nach FusG
SR02	Aufforderung an die Gläubiger infolge Spaltung
SR03	Umwandlung einer AG in eine GmbH
SR04	Sitzverlegung der Gesellschaft ins Ausland
SR05	Schuldenruf AG: Herabsetzung des Aktienkapitalschuldenruf GmbH: Herabsetzung des Stammkapitals und Aufforderung an die Gläubiger Schuldenruf Genossenschaft: Aufhebung der Anteilscheine und Aufforderung an die Gläubiger
SR06	Schuldenruf infolge Fusion nach Altrecht (OR)

Tab. 48: SHAB Sub-rubrics

9 HR02 Classification Codes

Code	Meaning
06	Company name
12	Capital
13	Registered seat
29	Company purpose
36	Legal form
37	Deletion, dissolution, bankruptcy or liquidation event
38	Corporate history or merger
49	Official register notice
54	Decision-maker or owner signature change
55	Additional signing-authority change
56	Auditor
73	Domicile or address
80	Official notice concerning proprietor

Tab. 49: HR02 Classification Codes

10 List of aids

Area of Responsibility	Tools
Typst Template	https://gitlab.com/Yutubi/typst-template-ost
Literature research and management	Google, Google Scholar, arxiv, GitHub, huggingface, ChatGPT, Claude, Google Gemini
Collaboration and project management	Microsoft Office, Notion, GitLab, YouTrack
Visualization	Excel, Python, Figma, Claude, ChartDB, Plant UML
Coding	VS Code, IntelliJ IDEA, PyCharm, Bruno, Mockoon, GitHub Copilot, ChatGPT, Claude, Google Gemini
DevOps	Docker, GitLab, MySQL, DuckDB, Java, NodeJS, Python, UV, Liquibase
Text creation, text optimization, spelling and grammar checking	Typst, VS Code, Deepl, ChatGPT
Image Creation (title image)	ChatGPT

Tab. 50: List of aids