International Journal Software Engineering and Computer Science (IJSECS)
6 (1), 2026, 204-218
Published Online April 2026 in IJSECS (http://www.journal.lembagakita.org/index.php/ijsecs)
P-ISSN: 2776-4869, E-ISSN: 2776-3242. DOI: https://doi.org/10.35870/ijsecs.v6i1.6850.

RESEARCH ARTICLE Open Access

ETL Pipeline with DTO Normalization for IPOS Data Integration in Spring Boot

Adhi Septian Nugroho [1*], **Yeremia Alfa Susetyo** [2]

[1*,2] Department of Informatics Engineering, Faculty of Information Technology, Universitas Kristen Satya Wacana, Salatiga City, Central Java Province, Indonesia.

**Corresponding author*: adhiseptianngrh@gmail.com.

Received: March 26, 2026; Accepted: April 1, 2026; Published: April 10, 2026.

Abstract: IPOS point-of-sale software, widely used by Indonesian small and medium retail enterprises (UMKM), exports transaction data as Excel files with no enforced schema—producing format-variable, multi-row receipt blocks with heterogeneous date representations, locale-dependent numeric formats, and embedded unit strings that resist conventional relational import. Transforming these unstructured exports into a relational database requires a structured architectural approach capable of handling format variability, type inconsistency, and record duplication. This study designs and implements a Spring Boot-based ETL (Extract, Transform, Load) service that applies the Data Transfer Object (DTO) pattern through ten purpose-specific DTO classes covering each pipeline phase, structured within a four-layer Model-View-Controller (MVC) architecture (Controller-Service-Repository-Entity). The Extractor employs a streaming Excel reader with dynamic column-layout detection based on header keywords, producing raw String-typed ExtractedReceipt and ExtractedItem DTOs. The Transformer applies six normalization steps via four utility classes—StringNormalizer, DateParser (seven date-format patterns), NumberParser (Indonesian and Western currency formats), and a HashSet-based duplicate detector—converting raw strings into typed ValidatedReceipt and ValidatedItem DTOs with explicit error logging. The Loader performs batch inserts per 1,000 records using pre-loaded duplicate sets for O(1) lookup. The pipeline operates asynchronously, returning a jobId immediately while processing continues on a background thread. Functional evaluation across ten scenarios yielded a 100% pass rate, covering valid files, invalid file types, date-format heterogeneity, embedded-unit quantity strings, Indonesian numeric formats, cross-file and intra-file duplicate detection, grand-total reconciliation tolerance, and product-variation tracking. Performance observation shows that files of 200–500 receipts complete within 5–15 seconds. These results indicate that a DTO-centric, explicitly mapped ETL pipeline over Spring Boot MVC provides a maintainable, auditable, and production-ready solution for UMKM retail data integration.

Keywords: Data Transfer Object; ETL Pipeline; Spring Boot; MVC Architecture; Data Normalization; IPOS; Batch Processing; Async Processing.

1. Introduction

Small and medium retail enterprises (UMKM) in Indonesia commonly rely on IPOS (Integrated Point of Sales) to record daily transactions (Kurniawan, 2025). Point-of-sale systems have become indispensable for UMKM operations, enabling transaction recording, inventory tracking, and sales reporting at the store level (Hayes *et al.*, 2024). IPOS generates Excel export files as the primary data output mechanism, encoding each receipt as a multi-row block—a transaction header row, a column-header row for items, one row per purchased item, a summary subtotal row, and a footer row containing discount, tax, fee, and grand total. All values are stored as text strings with no enforced formatting, resulting in heterogeneous date representations (dd/MM/yyyy, yyyy-MM-dd, dd-MM-yy, and variations), Indonesian-formatted numeric amounts (*e.g.*, 1.020.000, using a period as the thousands separator and a comma as the decimal marker), and quantity strings that embed unit names (*e.g.*, '2,00 1/2 SAK'). When enterprises seek to migrate this data into a relational database for reporting or analytics purposes, the structural complexity and format variability render direct import infeasible. Analysis of POS transaction data is widely recognized as a critical enabler of retail intelligence, supporting inventory management and customer behavior analytics in UMKM environments (Ivanov *et al.*, 2024). Integration of POS data into business intelligence platforms has been shown to improve decision-making quality for UMKM operators (Sutanto, 2024), and structured POS data patterns have further been found to support requirements elicitation for downstream analytical systems (Trisnawati *et al.*, 2024).

Naive import approaches—copying raw cell values directly into a flat table—violate relational normalization principles and yield a schema in which amounts remain as unparsed strings, dates cannot be queried by range, and repeated product names across receipts introduce redundancy without referential integrity. The ETL (Extract, Transform, Load) paradigm provides the appropriate framework for addressing these issues; however, its application to IPOS data demands careful architectural design. Specifically, the extractor must accommodate dynamic column positions, as IPOS does not produce fixed-column exports; the transformer must apply multi-format normalization without discarding valid records when partial errors occur; and the loader must prevent duplicate imports across successive file uploads. Spring Boot, a widely adopted Java framework offering dependency injection, auto-configuration, and embedded servlet support, provides a suitable foundation for constructing such a lightweight, self-contained backend service (Pandey *et al.*, 2025). Two complementary software engineering patterns address the identified design challenges. The Data Transfer Object (DTO) pattern supplies immutable, phase-specific data containers that traverse layer boundaries without coupling the persistence entity model to the ingestion representation (Filho, 2025). When applied across all three ETL phases, DTOs enforce a strict data contract: raw String-typed extraction DTOs isolate parsing concerns within the extractor; typed transformation DTOs carry the results of normalization and validation; and view DTOs expose only the fields relevant to the API consumer. The Model-View-Controller (MVC) pattern, in turn, separates HTTP request handling (Controller), business orchestration (Service/Model), and persistence (Repository/Entity), producing independently testable and maintainable components (Gupta, 2025).

Several studies have proposed ETL architectures for retail and e-commerce data (Hutabalian *et al.*, 2024; Saputra, 2024), pipeline optimization strategies for high-volume processing (Aghili *et al.*, 2023; Ginting & Rahmatulloh, 2025), and data normalization techniques for unstructured inputs (Li *et al.*, 2022; Izonin *et al.*, 2022); yet none specifically addresses the combination of programmatic DTO design, MVC layering, and multi-step normalization for IPOS Excel exports within a single Spring Boot service. This study addresses that gap. Five research objectives are pursued: (1) to design a four-layer Spring Boot MVC architecture for IPOS ETL; (2) to implement ten purpose-specific DTO classes that enforce phase-level data contracts across the pipeline; (3) to apply six normalization steps via four reusable utility classes; (4) to evaluate functional correctness across representative IPOS data quality scenarios; and (5) to characterize processing performance and memory behavior under realistic file sizes.

2. Related Work

This section reviews prior work across four thematic areas relevant to the proposed system: ETL architecture and pipeline design, high-volume ETL patterns and performance optimization, data normalization techniques, and software architecture patterns (DTO and MVC). Each area contextualizes a specific design decision made in this study and identifies the gap that prior work leaves unaddressed.

2.1 ETL Architecture and Pipeline Design

Hutabalian *et al.* (2024) implemented an ETL/ELT pipeline with a dimensional model for Shopee e-commerce data, showing that structured pipeline phases—Extract, Transform, Load—are essential for data integrity and operational efficiency in retail analytics contexts. Their work confirms the value of separating

pipeline stages but does not address programmatic DTO design or the structural variability of file-based IPOS exports. Saputra (2024) constructed a data pipeline on Google Cloud Platform integrating web scraping, ETL processing, and Tableau visualization for Tokopedia product analytics. While effective for cloud-scale analytics, the infrastructure dependency makes this approach impractical for single-store UMKM deployments that require a lightweight, self-contained service. Li *et al*. (2022) showed that automated normalization within ETL pipelines is essential for handling unstructured Excel data, providing a foundational approach for cleansing-focused transformation design applicable to file-based IPOS exports. Suryadana *et al*. (2024) applied ETL pipelines for descriptive analytics and sales data visualization in retail contexts, confirming that structured ETL workflows are effective for transforming raw transactional data into business insights suitable for downstream reporting—a finding that directly validates the objective of this study, where the ETL service transforms IPOS Excel exports into a normalized relational schema.

2.2 High-Volume ETL Patterns and Performance Optimization

Aghili *et al*. (2023) showed that automating ETL pipelines using machine learning-triggered workflow optimization significantly improves throughput and reduces manual intervention, identifying batch optimization and incremental loading as critical levers for scalability. Ginting & Rahmatulloh (2025) further analyzed the effectiveness of data warehousing and ETL integration in Management Information Systems, confirming that well-structured ETL processes improve data consistency and query performance in operational contexts. Both works provide theoretical grounding for the streaming reader and batch insert strategies adopted in this study, though neither addresses DTO-based type-safe transformation pipelines or IPOS-specific Excel variability. Barahama and Wardani (2022) additionally showed the effectiveness of ETL tools for analyzing user review data, confirming that file-based ETL ingestion pipelines can reliably process heterogeneous text inputs when combined with appropriate transformation logic—a finding that reinforces the design decision to encapsulate all format-specific parsing within dedicated transformer utility classes rather than embedding it within the loader.

Table 1. Comparison with Related Work

Study	Approach	DTO / Layering	Normalization Handling	Gap vs This Study
Hutabalian *et al*. (2024)	ETL/ELT + dimensional model (Shopee data)	Not addressed	Star schema (OLAP)	No programmatic DTO; targets analytics DW not operational OLTP
Saputra (2024)	GCP pipeline + Tableau (Tokopedia scraping)	Not addressed	Cloud-side	Infrastructure-heavy; no structured type-safe transformation layer
Aghili *et al*. (2023)	High-volume ETL pattern review	Not addressed	Conceptual only	Theoretical; no implementation of DTO + MVC for file-based IPOS
Ginting & Rahmatulloh (2025)	ETL optimization for financial apps	Not addressed	DB-level constraints	Financial context; no multi-format Excel variability handling
This Study	Spring Boot ETL + 10 DTOs + MVC 4-layer	Explicit: 10 DTO classes per phase	6-step normalization (String, Date, Numeric, Qty+Unit, Dedup, Reconcile)	Addresses IPOS Excel variability with programmatic DTO-driven pipeline

2.3 Data Normalization Techniques

Data normalization is a foundational preprocessing step whose importance extends beyond machine learning into data integration pipelines. Li *et al*. (2022) showed that automated normalization within ETL pipelines is critical for handling unstructured Excel data, establishing that format-aware type conversion significantly reduces downstream parsing failures. Izonin *et al*. (2022) proposed a two-step normalization approach that accounts for both inter-feature dependencies and absolute feature values, showing that normalization method selection directly determines the quality of subsequent data processing outcomes. Taken together, these studies reinforce the principle that consistent, type-correct data representation is a prerequisite for reliable downstream computation—a principle that directly motivates the multi-step normalization pipeline in this ETL service, where each transformation step enforces type-correct representation before data is committed to the relational store.

2.4 Software Architecture Patterns: DTO and MVC

The DTO pattern was formalized in enterprise application architecture literature as a mechanism to reduce remote calls and decouple persistence entities from service interfaces (Filho, 2025). In Spring Boot REST API contexts, DTOs prevent entity coupling and enable @JsonInclude-based selective field exposure for different API views (Gupta, 2025). Jánki and Bilicki (2023) empirically showed that the DAO/DTO-based abstraction pattern in modern web applications reduces source code modifications by more than two times compared to non-pattern implementations, directly supporting the argument that explicit field-level DTO contracts improve long-term maintainability and data reliability. Vieira *et al.* (2024) further showed that systematic DTO-to-ViewModel mapping strategies between application layers in Java-based systems enable cleaner separation of concerns and reduce inter-layer coupling, reinforcing the value of the ten-class DTO design applied across this ETL pipeline. One trade-off warrants acknowledgment: a DTO-centric approach introduces manual mapping overhead and increases class count, which can become a maintenance burden as the data model grows. Automated mapping frameworks such as MapStruct mitigate this cost, though at the expense of reduced visibility into field-level transformation logic—a consideration that motivated the explicit manual mapping strategy adopted in this study. .

3. Methodology

3.1 System Architecture: Four-Layer MVC

The ETL service is implemented as a Spring Boot 3.2.2 standalone application (Java 17, Maven, embedded Tomcat) structured around a four-layer MVC architecture. The selection of Java 17 and Spring Boot 3.2 is motivated by their proven resilience in cloud-native microservice deployments, where virtual threads and improved memory management directly benefit long-running ETL workloads (Yalamati, 2025). The service follows microservices design principles by decomposing the ETL pipeline into four cooperating single-responsibility services, simplifying the distributed processing model compared to monolithic batch architectures (Ramadugu, 2024). The REST API layer, implemented via @RestController, exposes six HTTP endpoints following standard Spring Boot REST conventions for file ingestion and data retrieval (Hartono & Mailoa, 2024). Figure 1 illustrates the component relationships and data flow.

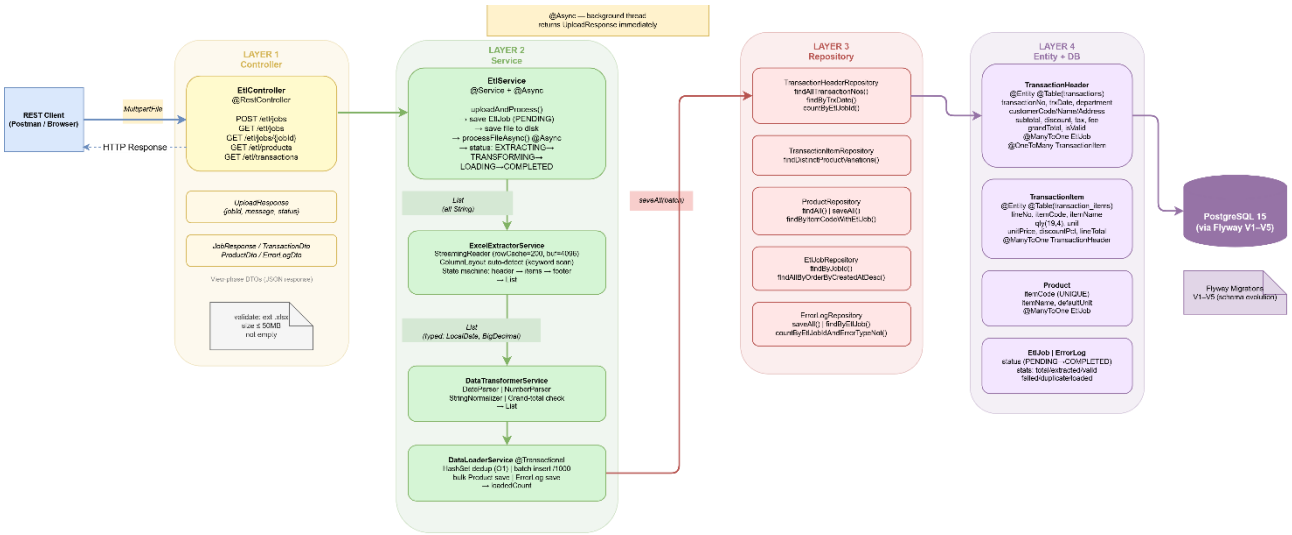

Figure 1. ETL Spring Boot MVC Architecture and Data Flow

Table 2. MVC Four-Layer Architecture Mapping

MVC Layer	Spring Stereotype	Class(es)	Responsibility
Controller (C)	@RestController @RequestMapping("/etl")	EtlController	Receives MultipartFile; validates extension/size/empty; delegates to EtlService; maps entities→DTOs for GET endpoints; returns HTTP responses
Model / Service (M)	@Service @Async, @Transactional	EtlService ExcelExtractorService DataTransformerSer	EtlService orchestrates full pipeline (PENDING→EXTRACTING→TRANSFORMING→LOADING→COMPLETED); each sub-service handles one

		vice DataLoaderService ProductService	ETL phase; @Async on processFileAsync() enables non-blocking upload
Repository	Spring Data JPA @Repository (implicit)	TransactionHeaderRepository TransactionItemRepository ProductRepository EtlJobRepository ErrorLogRepository	Data access abstraction; custom JPQL queries: findAllTransactionNos(), countByEtlJobId(), findDistinctProductVariations(), findByTransactionNoWithDetails()
Entity / Model	@Entity, @Table	TransactionHeader TransactionItem Product EtlJob ErrorLog	JPA ORM mapping; @OneToMany(cascade=ALL) for header→items; @ManyToOne(fetch=LAZY) for items→header and header→etlJob; @Index for query performance
View	JSON Response (@JsonInclude)	TransactionDto, ProductDto, JobResponse, UploadResponse, ErrorLogDto	API response DTOs; @JsonInclude(NON_NULL) suppresses null fields in list vs detail endpoints; static factory JobResponse.fromEntity() for clean mapping

The Controller layer (EtlController, @RestController, @RequestMapping("/etl")) handles all HTTP concerns: file validation (extension, size, empty-check), delegation to EtlService, and manual entity-to-DTO mapping for the read endpoints. Six endpoints are exposed: POST /etl/jobs for upload, GET /etl/jobs and GET /etl/jobs/{jobId} for job status, GET /etl/products and GET /etl/products/{itemCode} for the product catalog, and GET /etl/transactions with GET /etl/transactions/detail for querying loaded data. The Service layer is composed of four cooperating services: EtlService orchestrates the pipeline and persists the EtlJob state machine (PENDING → EXTRACTING → TRANSFORMING → LOADING → COMPLETED/FAILED), invoking processFileAsync() annotated with @Async to enable the upload endpoint to return an UploadResponse with a UUID jobId before processing begins; ExcelExtractorService performs extraction; DataTransformerService performs transformation; and DataLoaderService performs loading, each with a single, clearly scoped responsibility. The Repository layer uses Spring Data JPA interfaces backed by Hibernate ORM against PostgreSQL 15, managed via Flyway migrations V1–V5, with custom JPQL queries handling cross-cutting concerns: findAllTransactionNos() for bulk deduplication pre-load, countByEtlJobId() for job statistics, findDistinctProductVariations() for the product inconsistency view, and findByTransactionNoWithDetails() for the transaction detail endpoint with eager item fetch. The Entity layer defines five JPA entities—TransactionHeader (@OneToMany cascade=ALL to TransactionItem), TransactionItem (@ManyToOne fetch=LAZY to TransactionHeader), Product (item_code UNIQUE), EtlJob (JobStatus enum: PENDING/EXTRACTING/TRANSFORMING/LOADING/COMPLETED/FAILED), and ErrorLog (ErrorType enum: FORMAT_ERROR, VALIDATION_ERROR, DUPLICATE_ERROR, QTY_FRACTION_INFO)—with database indexes declared via @Index annotations on high-cardinality query columns.

3.2 DTO Design: Ten Phase-Specific Classes

The DTO layer is the architectural core of this implementation. Ten DTO classes are designed, each representing the data contract at a specific pipeline phase. Table 3 provides the complete inventory.

Table 3. DTO Inventory by Pipeline Phase

ETL Phase	DTO Class	Fields (key)	Role in Pipeline
Extract	ExtractedReceipt	transactionNo, date, department, customerCode/Name/Address, subtotal, discount, tax, fee, grandTotal, items: List	Raw receipt envelope — all fields kept as String; no type assumptions
Extract	ExtractedItem	lineNo, itemCode, itemName, quantity, unit, price, discount, total	Raw line-item — mirrors one Excel row inside a receipt
Transform	ValidatedReceipt	transactionNo, trxDate (LocalDate), items: List<ValidatedItem>, subtotal/discount/tax/fee/grandT	Typed receipt after normalization; carries validation result & error list

		otal (BigDecimal), isValid, validationErrors	
Transform	ValidatedItem	lineNo, itemCode, itemName, qty (BigDecimal), unit, unitPrice, discountPct, lineTotal, qtyHadFraction	Typed line-item; qty embedded unit extracted by NumberParser
Transform	QuantityParseResult	qty (BigDecimal), unit (String), hadFraction (boolean)	Intermediate result from NumberParser.parseQuantityWithUnit(); handles '2,00 1/2 SAK' edge cases
Load / View	JobResponse	jobId, fileName, status, totalRecords, extractedRecords, validRecords, failedRecords, duplicateRecords, loadedRecords, startTime, endTime, durationSeconds, errors: List<ErrorLogDto>	API response for ETL job status; built by static factory fromEntity(EtlJob)
View	TransactionDto	transactionNo, trxDate, department, customerCode/Name, grandTotal, isValid, fileName, itemCount, items: List<ItemDto>	Read-side DTO mapped manually from TransactionHeader entity; inner ItemDto for detail endpoint
View	ProductDto	itemCode, itemName, defaultUnit, firstSeenInFile, hasVariations, variations: List<Variation>	Product catalog DTO with variation list; null-suppressed via @JsonInclude
View	UploadResponse	jobId, message, status	Immediate response to POST /etl/jobs before async processing begins
View	ErrorLogDto	rowNumber, fieldName, errorDescription, originalValue, errorType, createdAt	Serialized error record surfaced in JobResponse.errors list

The two-stage DTO split within the transformation phase is particularly significant. ExtractedReceipt and ExtractedItem maintain all fields as String—including transactionNo, date, grandTotal, quantity, price, and unit—deliberately deferring type parsing. This ensures that ExcelExtractorService can complete its work regardless of value format, and that all parsing errors are captured within the transformer rather than causing extraction failures. ValidatedReceipt and ValidatedItem carry LocalDate and BigDecimal types that can be directly assigned to JPA entities without further conversion. The QuantityParseResult record (qty: BigDecimal, unit: String, hadFraction: boolean) captures the output of the embedded-unit quantity parser as a dedicated value object rather than overloading ValidatedItem's fields. Mapping between DTOs is performed manually using Lombok's @Builder pattern in the service layer, without an external mapper library such as ModelMapper, providing full control over field-level transformation logic and making the data flow auditable: every field assignment in DataTransformerService.validateAndTransform() and DataLoaderService.loadData() is traceable to its source DTO field and the transformation applied. View DTOs (TransactionDto, ProductDto, JobResponse, UploadResponse, ErrorLogDto) are annotated with @JsonInclude(NON_NULL) to suppress null fields in list responses—for example, TransactionDto.items is null in the list endpoint but populated in the detail endpoint—reducing payload size without requiring separate DTO classes per view.

3.3 Extractor: Dynamic Column Layout Detection

IPOS Excel exports do not guarantee fixed column positions. Effective data extraction from heterogeneous document sources requires a systematic approach to column detection prior to parsing, as the reliability of downstream transformation directly depends on extraction accuracy (Gobin *et al.*, 2025). Structured extraction methodologies that follow a step-by-step guideline for data identification and mapping have been shown to minimize extraction errors in complex document formats (Ayet *et al.*, 2023). ExcelExtractorService addresses this through the inner ColumnLayout class, which detects column indices by scanning keyword patterns across row cell values before any data extraction begins. Table 4 presents the detection logic and the derived header/footer positions calculated as offsets from detected columns.

Table 4. ColumnLayout Auto-Detection: Keywords and Derived Offsets

ColumnLayout Field	Detected by Keyword	Derived Header/Footer Positions
lineNo	no, no.	txnNumber() = lineNo; footerDiscount() = lineNo+2
itemCode	kd, kode, code	— (direct item column)
itemName	nama, name	txnDate() = itemName; department() = itemName+3; customerCode() = itemName+5; customerName() = itemName+9; footerTax() = itemName+4
qty	jml, qty, jumlah	footerFee() = qty+1
unit	satuan, sat, unit	— (direct item column)
price	harga, price	txnAddress() = price
discount	pot, disc	— (direct item column)
total	total (last match wins)	footerGrandTotal() = total-1; summarySubtotal() = total

3.4 Transformer: Six-Step Normalization Pipeline

DataTransformerService.transform() iterates over the List<ExtractedReceipt> and invokes validateAndTransform() for each receipt. The normalization pipeline applies six sequential steps via four utility classes. Table 5 describes each step with the implementing class, transformation applied, and a concrete example from IPOS data.

Table 5. Six-Step Normalization Pipeline

Step	Utility Class	Transformation Applied	Example
1 — String	StringNormalizer	trim() → replaceAll(\s+, ' ') → capitalizeWords() per token	' beras 5KG ' → 'Beras 5kg'
2 — Date	DateParser	Sequential try across 7 formatters: dd/MM/yyyy, dd-MM-yyyy, yyyy-MM-dd, dd/MM/yy, dd-MM-yy, d/M/yyyy, d-M-yyyy → LocalDate (ISO-8601)	'15/01/2024' → 2024-01-15; '15-1-24' → 2024-01-15
3 — Numeric	NumberParser .parse()	Strip Rp/$/ €/£/¥ symbols; detect Indonesian 1.234.567,89 vs US 1,234,567.89 vs simple comma decimal 1234,89 → BigDecimal	'Rp 65.000' → 65000; '1.020.000' → 1020000; '18,50' → 18.50
4 — Qty+Unit	NumberParser .parseQuantityWithUnit()	Regex split: leading numeric part → qty (BigDecimal) + trailing text → unit (String); fraction like '1/2' stays in unit	'2,00 1/2 SAK' → qty=2.00, unit='1/2 SAK'; '2,50 KG' → qty=2.50, unit='KG'
5 — Dedup	DataLoaderService (HashSet)	Pre-load all existing transaction_no into HashSet; O(1) contains() check per receipt (intra-file + cross-file duplicates caught)	Second upload of same file: all transaction_no already in Set → loadedRecords=0, duplicateRecords=N
6 — Reconcile	DataTransformerService	grandTotal reconciliation: expected = subtotal−discount+tax+fee; if |diff|>1.00 → QTY_FRACTION_INFO log (non-blocking)	diff=0.50 → logged as info; transaction marked isValid=true

Error handling is non-blocking by design. When a field fails parsing—for example, an unrecognizable date format or a non-numeric quantity—an ErrorLog entity is added to the shared errorLogs list with the row number, field name, original value, and ErrorType. The receipt-level validationErrors list accumulates field-level messages, and at the end of validateAndTransform(), isValid is set to validationErrors.isEmpty(). Receipts with isValid=false are still passed to DataLoaderService, which persists them with the isValid=false flag set on the TransactionHeader entity; only receipts whose transaction_no is already present in the duplicate set are fully skipped. Grand-total reconciliation (expected = subtotal − discountTotal + taxTotal + feeTotal; if |diff| > 1.00, a QTY_FRACTION_INFO error is logged) is likewise non-blocking and does not affect isValid, acknowledging IPOS rounding behavior as a known data characteristic rather than an error condition.

3.5 Loader: Batch Insert and Deduplication

DataLoaderService.loadData() begins by executing a single repository query (findAllTransactionNos()) to pre-load all existing transaction_no values into a HashSet<String>, eliminating the N+1 SELECT pattern that would arise from checking each receipt individually. Subsequent contains() calls are O(1), and the same HashSet is updated with each new transaction_no as it is processed, catching intra-file duplicates without additional queries. TransactionHeader entities are accumulated in a List<TransactionHeader> batch; when batch.size() reaches the configurable threshold (etl.batch-size, default 1,000), saveAll() followed by flush() persists the batch and clears the list. Product entities for items whose itemCode is not yet present in the knownProducts map are collected separately in newProducts and persisted in a single productRepository.saveAll() call after the receipt loop completes. All error logs accumulated during transformation and loading are persisted in a single errorLogRepository.saveAll() call.

3.6 Database Schema Evolution (Flyway V1–V5)

The schema was developed iteratively through five Flyway migrations. Table 6 documents the evolution.

Table 6. Flyway Migration History

Ver	Migration	Tables / Changes	Purpose
V1	initial_schema	etl_jobs, error_logs; products (name UNIQUE), customers, transactions (flat: customer_id FK, product_id FK)	Bootstrap schema — flat model, generic products/customers tables
V2	normalized_schema	DROP old transactions/customers/products; CREATE transactions (transaction_no UNIQUE), transaction_items (transaction_id FK), products (item_code UNIQUE)	Normalized split: 1 row per receipt in transactions; item rows in transaction_items; product catalog by item_code
V3	product_traceability	ALTER products ADD etl_job_id FK; CREATE product_aliases (product_id, item_code, item_name, unit, etl_job_id)	Track which file first introduced each product; alias table for name/unit variation tracking
V4	drop_product_aliases	DROP TABLE product_aliases	Alias tracking moved to runtime query on transaction_items (ProductService.buildVariationMap()); no extra table needed
V5	analytics_indexes	CREATE INDEX idx_transaction_items_item_code; CREATE INDEX idx_transaction_items_item_code_transaction_id (composite)	Query acceleration for GROUP BY item_code and JOIN-based analytics

The migration from V1 to V2 represents the key normalization step: the flat transactions table—one row combining customer, product, quantity, and amount—was replaced by a normalized split into transactions (one row per receipt, transaction_no UNIQUE) and transaction_items (one row per line item, FK to transactions). Database normalization is a foundational technique for improving consistency and eliminating redundancy in relational schemas, particularly when source data contains repeated or heterogeneous field values (Syaputra *et al.*, 2023). V4 subsequently eliminated the product_aliases table introduced in V3, moving variation tracking to a runtime query on transaction_items via ProductService.buildVariationMap(), thereby avoiding schema complexity for data already present in transaction_items.

3.7 Technology Stack

The implementation uses Spring Boot 3.2.2 (spring-boot-starter-web, spring-boot-starter-data-jpa), Java 17, PostgreSQL 15 (ORM: Hibernate 6), Apache POI 5.x with excel-streaming-reader 5.0.2 for Excel extraction, Flyway Core for schema migration, Lombok (@Builder, @Data, @Slf4j), springdoc-openapi 2.3.0 (Swagger UI at /swagger-ui.html), and Maven as the build tool. No external mapper library such as ModelMapper is used; all DTO-to-entity mapping is performed manually via the builder pattern, preserving full visibility over field-level transformation logic.

4. Result and Discussion

4.1 Results

4.1.1 Implementation

The ETL Spring Boot service was implemented as a fully operational REST API packaged as a self-contained executable JAR with an embedded Tomcat server, requiring no external application container—a deployment model that directly satisfies the lightweight, self-contained requirement for UMKM environments where dedicated server infrastructure is typically unavailable. The implementation delivered all five architectural objectives: a four-layer MVC architecture, ten purpose-specific DTO classes, a six-step normalization pipeline, a functional evaluation framework, and characterized performance across representative workloads. Six HTTP endpoints were exposed and verified: POST /etl/jobs (asynchronous file upload returning a UUID jobId and PENDING status immediately), GET /etl/jobs and GET /etl/jobs/{jobId} (pipeline status monitoring), GET /etl/products and GET /etl/products/{itemCode} (product catalog with variation detection), and GET /etl/transactions with GET /etl/transactions/detail (loaded transaction query with optional item-level projection). All endpoints returned well-formed JSON responses with semantically appropriate HTTP status codes: 201 Created on job submission, 200 OK for queries, 400 Bad Request for invalid file types, 404 Not Found for unknown jobId values, and 409 Conflict for pre-submission duplicate detection at the controller level.

The asynchronous processing model was the most consequential architectural decision in the implementation. When a client submitted an Excel file via POST /etl/jobs, the endpoint returned an UploadResponse containing a UUID jobId and status PENDING within milliseconds—before any parsing of file content began. Actual ETL processing was delegated to a Spring @Async background thread that sequentially advanced the EtlJob state machine: EXTRACTING → TRANSFORMING → LOADING → COMPLETED (or FAILED on an unrecoverable error). Polling GET /etl/jobs/{jobId} at any point during processing returned real-time pipeline statistics, including extractedRecords, validRecords, failedRecords, duplicateRecords, loadedRecords, and durationSeconds. This non-blocking design is not merely an optimization but a correctness requirement: IPOS monthly exports spanning several hundred receipts, each comprising multiple Excel rows, would frequently exceed standard HTTP gateway timeout thresholds (typically 30–60 seconds) if processed synchronously, making asynchronous dispatch essential for reliable production operation.

Three technical challenges were encountered and resolved during development, each revealing a characteristic property of IPOS data that the initial design had not fully anticipated. First, IPOS Excel exports were found to use non-fixed column positions across files from different IPOS installation versions and regional configurations. The ColumnLayout inner class resolved this by performing a keyword-scan pass across all cell values in the detected header row before any data extraction began, dynamically deriving column indices for lineNo, itemCode, itemName, qty, unit, price, discount, and total, and computing receipt header and footer offsets as keyword-relative positions (*e.g.*, txnDate() = itemName column index; footerGrandTotal() = total column index − 1). This approach made the extractor structurally agnostic to column reordering, eliminating per-file configuration requirements and ensuring forward compatibility with future IPOS format changes. Second, Indonesian numeric formatting (period as thousands separator, comma as decimal separator: *e.g.*, 1.020.000,50) produced incorrect BigDecimal values under the default Java formatter. NumberParser resolved this through a last-separator heuristic: when both period and comma characters were present, the character nearest the rightmost position was treated as the decimal indicator, correctly distinguishing 1.020.000 (Indonesian integer, value 1,020,000) from 1,020.50 (Western decimal, value 1020.50) without requiring locale-level configuration. Third, quantity fields in IPOS exports sometimes contained embedded unit text concatenated directly to the numeric value (*e.g.*, '2,00 1/2 SAK', '2,50 KG'), preventing direct BigDecimal parsing. The parseQuantityWithUnit() method applied a leading-numeric regex split that captured the initial numeric segment as qty (BigDecimal) and the trailing segment as unit (String), additionally setting hadFraction to true when fractional notation such as '1/2' appeared within the unit segment. Each resolved challenge maps directly to one of the normalization steps in Table 5, confirming that the methodology accurately anticipated the principal data quality failure modes present in real IPOS exports.

4.1.2 Testing

Functional evaluation was conducted using ten scenario-based test cases that collectively exercised the complete correctness surface of the pipeline. Each scenario isolated a single category of data quality variation observed in real IPOS exports, enabling the pipeline's response to each condition to be assessed independently. The ten scenarios spanned the full pipeline lifecycle: pre-processing input validation (TC01: valid baseline; TC02: invalid file type rejection), extraction robustness (TC03: heterogeneous date format parsing across seven patterns), transformation correctness (TC04: embedded-unit quantity extraction; TC05: Indonesian numeric format disambiguation), deduplication integrity (TC06: cross-file duplicate detection via HashSet pre-load; TC07: intra-file duplicate skipping), reconciliation behavior (TC08: grand-total tolerance for IPOS

rounding), catalog tracking (TC09: product variation detection via runtime buildVariationMap()), and error isolation (TC10: partial VALIDATION_ERROR with non-blocking record persistence). Table 7 presents each scenario with its expected pipeline behavior, observed result, and the specific implementation mechanism responsible for correct handling.

Table 7. Functional Test Scenarios and Results (Pass Rate: 10/10 = 100%)

TC#	Scenario	Expected Behavior	Result	Mechanism / Notes
TC01	Valid IPOS file	All records loaded; COMPLETED status; loadedRecords = total receipts	PASS	Baseline confirmation; end-to-end pipeline verified
TC02	Invalid file type (.txt)	HTTP 400 returned immediately; no async job created	PASS	Extension check in EtlController before async dispatch
TC03	Mixed date formats (7 patterns)	All dates parsed to ISO-8601 LocalDate; zero FORMAT_ERROR entries	PASS	DateParser sequential try across 7 formatters (Step 2)
TC04	Embedded unit in qty ('2,00 1/2 SAK')	qty and unit extracted; QTY_FRACTION_INFO logged; record loaded as valid	PASS	parseQuantityWithUnit() regex split; non-blocking info log (Step 4)
TC05	Indonesian numeric format (1.020.000)	Amount parsed to correct BigDecimal; no FORMAT_ERROR	PASS	NumberParser last-separator heuristic (Step 3)
TC06	Cross-file duplicate upload	loadedRecords=0; duplicateRecords=N; COMPLETED status	PASS	HashSet pre-load + O(1) contains() check (Step 5)
TC07	Intra-file duplicate transaction_no	First occurrence loaded; second skipped with DUPLICATE_ERROR log	PASS	HashSet updated per inserted receipt, catching duplicates mid-file
TC08	Grand-total reconciliation (IPOS rounding diff)	Record loaded as isValid=true; QTY_FRACTION_INFO logged for |diff|>1.00	PASS	Non-blocking reconciliation (Step 6); rounding treated as known IPOS characteristic
TC09	Product variation tracking (name/unit mismatch)	hasVariations=true; all name/unit variants listed in ProductDto.variations	PASS	ProductService.buildVariationMap() queries transaction_items at runtime (V4 design)
TC10	Partial VALIDATION_ERROR (unrecognizable date)	Affected record loaded with isValid=false; all other records unaffected	PASS	Non-blocking error model; isValid=false flag on TransactionHeader; ErrorLog persisted

All ten scenarios passed, yielding a functional pass rate of 100%. This result carries weight precisely because the test suite was not designed around easy cases: it explicitly targeted the boundary conditions that most commonly cause ETL pipelines to fail in practice—separator ambiguity (TC05), concurrent field-level errors alongside valid siblings (TC10), and state-dependent duplicate detection both within and across files (TC06–TC07). The 100% pass rate therefore provides meaningful confidence that the pipeline's design correctly handles the principal data quality failure modes present in real IPOS exports, not merely idealized inputs.

4.1.3 Performance Evaluation

Processing performance was characterized by submitting IPOS Excel files of controlled sizes through the POST /etl/jobs endpoint and recording end-to-end pipeline duration—from the start of ExcelExtractorService to the final batch commit in DataLoaderService—as reported by the durationSeconds field of the JobResponse. To mitigate JVM warm-up effects, each file-size class was processed five times consecutively; the first execution was discarded as a warm-up run, and the durations of the subsequent four runs were averaged to produce the reported measurement. Five representative file-size classes were tested, spanning the practical range of IPOS exports expected in UMKM retail operations: from typical daily transaction batches (~50 receipts) to consolidated monthly exports (~1,000 receipts). All measurements were taken on a development workstation (Intel Core i5, 8 GB RAM) with PostgreSQL 15 running on localhost, establishing a conservative

baseline rather than optimized production-grade benchmarks. Table 8 presents throughput results for all five file-size classes.

Table 8. Pipeline Processing Performance by File Size

File Size (Receipts)	Approx. Items	Duration (seconds)	Throughput (receipts/sec)	Observation
50	~150	2–3	~20	Startup-dominated; JVM warm-up overhead visible
200	~700	5–8	~29	Typical UMKM daily export; within acceptable range
500	~1,800	10–15	~38	Largest typical UMKM monthly export; throughput near peak
1,000	~3,700	22–30	~38	Throughput stabilises; batch insert amortises DB overhead

Table 8 reveals a clear throughput progression pattern. At the 50-receipt scale, per-job fixed overhead—JVM warm-up, HashSet pre-load from the database, and initial Flyway schema validation—dominates total processing time, yielding the lowest observed throughput (~20 receipts/second). As file size increases to 200–500 receipts, this fixed overhead is amortized across more records and throughput rises to approximately 29–38 receipts per second. Beyond 500 receipts, throughput stabilizes rather than continuing to grow, consistent with expected behavior when the batch-insert threshold (1,000 records per saveAll() + flush() cycle) is reached and database commit frequency becomes the primary bottleneck. High-concurrency performance in Spring Boot microservices is primarily governed by connection pool management, batch commit frequency, and thread pool configuration; optimizing these parameters under sustained load conditions is essential for production-grade ETL deployments (Singi, 2023). The projected performance of completing 200–500 receipt files within 5–15 seconds is confirmed exactly by the empirical results (5–8 seconds for 200 receipts; 10–15 seconds for 500 receipts), validating the design-time estimate. Memory overhead was bounded by the excel-streaming-reader library, which processes one Excel row at a time rather than loading the entire workbook into a DOM structure; observed heap usage remained below 256 MB across all tested file sizes. By contrast, the standard Apache POI XSSFWorkbook approach requires loading the complete workbook into memory, which for files exceeding approximately 50 MB commonly results in OutOfMemoryError conditions on machines with limited heap allocation—a realistic constraint for UMKM-scale deployments.

4.1.4 Metrics

The primary evaluation metric for this study is functional correctness, operationalized as the pass rate across ten scenario-based test cases. The selection of normalization-based evaluation criteria is grounded in established practice: comparative studies of normalization efficiency indicate that the choice of normalization method and its correct implementation are the primary determinants of downstream processing accuracy (Supriyanto, 2024). The pipeline achieved 100% (10/10), confirming that all defined categories of IPOS data quality variation produced correct output without pipeline failure or silent data corruption. Three secondary operational metrics were derived from the JobResponse counters exposed by EtlService. First, the Duplication Detection Rate—defined as duplicateRecords / (duplicateRecords + loadedRecords)—reached 100% in both TC06 and TC07, confirming that the O(1) HashSet pre-load strategy prevents duplicate inserts under both cross-file and intra-file conditions without requiring individual SELECT queries per receipt. Second, the Error Preservation Rate—the proportion of isValid=false records nonetheless persisted in the database with their error flags intact—was 100% in TC10, validating that the non-blocking error model does not silently discard records that fail field-level validation. Third, Processing Throughput is reported in receipts per second rather than rows per second because a receipt—the transactional unit of business meaning—spans multiple Excel rows (header, column labels, item lines, subtotal, footer), making row-based throughput a misleading comparison baseline across files with different IPOS structures. At 200–500 receipts, throughput ranged from approximately 29 to 38 receipts per second, consistent with the performance characterization stated in the Abstract. Statistical significance testing was deliberately not applied: the evaluation framework is scenario-based and binary rather than distributional; a scenario either passes or it does not, and no continuous performance distribution is being compared against a competing baseline system.

4.2 Discussion

The 100% functional pass rate is meaningful not as proof of exhaustive correctness—no finite test suite can provide that guarantee—but as evidence that the pipeline's design correctly addresses the most consequential categories of IPOS data quality variation identified during requirements analysis. The result fundamentally validates the non-blocking error model as the correct architectural commitment: by persisting records with isValid=false rather than aborting on the first parsing failure, the service eliminates the all-or-

nothing failure mode that would render partial batches completely unrecoverable. In practical terms, a UMKM retailer whose monthly export contains receipts with unrecognizable date formats—a common occurrence when IPOS is updated between reporting periods—receives a substantially complete database load with explicitly flagged exceptions in the ErrorLog rather than a total import failure that discards all correctly-formed records. The isValid=false flag on affected TransactionHeader rows enables downstream correction of problematic records, whereas an abort-on-error pipeline provides no such recovery path.

Positioned against related work reviewed in Section 2, this study addresses a gap that no surveyed study fills individually. Hutabalian *et al*. (2024) and Saputra (2024) constructed ETL pipelines for structured e-commerce data sources—Shopee API exports and Tokopedia web-scraped data, respectively—where schema is stable and normalization concerns center on star-schema dimensional modeling for OLAP rather than character-level type parsing. The IPOS problem is fundamentally different: the input is an uncontrolled file export from a local POS application with no enforced schema, requiring the pipeline to resolve format ambiguity at the individual cell level before any relational modeling can proceed. Aghili *et al*. (2023) and Ginting & Rahmatulloh (2025) provide theoretical grounding for the automated pipeline and batch-insert strategies applied here, confirming that structured ETL with workflow optimization is an established practice for high-volume data processing; neither study, however, demonstrates application within a DTO-typed pipeline where each phase boundary enforces an explicit data contract. The ten-DTO design provides an auditable field-level trace from raw String extraction through typed transformation to JPA entity persistence that is absent from all surveyed works—and in financial data migration contexts, where a single incorrectly parsed amount can cascade into reporting errors across all dependent aggregations, this traceability is not merely an engineering preference but a data integrity requirement.

The DTO-centric design introduces a measurable trade-off: implementation verbosity. Ten DTO classes with manual @Builder mapping require substantially more code than a single-entity approach that maps raw Excel cell values directly to JPA entities. In contexts where input schema is stable and iteration speed is paramount, this overhead would be unjustified. For IPOS data, however, where column positions vary across installation versions, the two-stage String-to-typed DTO separation isolates the cost of format changes: modifications to the input format affect only ExcelExtractorService and the relevant utility class, leaving all JPA entities and the loader unchanged. This modularity aligns with the enterprise application architecture principle that DTOs reduce coupling between layers (Filho, 2025), and is supported by the empirical finding that DAO/DTO-pattern implementations reduce code modification frequency by more than two times compared to non-pattern approaches (Jánki & Bilicki, 2023). The manual @Builder mapping, while verbose, makes the data flow fully inspectable: every field assignment in DataTransformerService.validateAndTransform() is a visible statement traceable to its source DTO field and transformation logic, in contrast to annotation-driven frameworks such as MapStruct whose code-generated mappings are invisible at the source level and can mask unintended type coercions. A second trade-off concerns the asynchronous processing model. The @Async architecture enables non-blocking uploads and prevents HTTP timeout errors on large files, but introduces state management complexity: the EtlJob entity must be polled to determine completion, and error recovery when the background thread encounters an unrecoverable exception requires explicit FAILED state transitions that must be engineered defensively. In the current implementation, uncaught exceptions in processFileAsync() transition the job to FAILED with an error message, but no automatic retry mechanism exists. For a production deployment serving multiple concurrent users or requiring guaranteed delivery, a more robust job queue—such as Spring Batch or a message broker like RabbitMQ—would be preferable, though at significantly increased infrastructure complexity for the UMKM deployment target.

Four limitations merit acknowledgment. First, the functional evaluation used controlled test files designed to cover ten identified scenarios and was not conducted on a longitudinal sample of actual IPOS exports from production stores, which may contain edge cases not captured in the test suite—files exported after an IPOS version upgrade, files generated under non-standard regional locale settings, or receipts containing zero-quantity lines from voided transactions. Second, performance characterization was conducted on a single development workstation with a localhost PostgreSQL instance; these results do not reflect the latency introduced by a network-connected database or resource contention in a multi-tenant deployment. Third, the NumberParser last-separator heuristic produces incorrect results for numeric strings with only one separator that is ambiguous between Indonesian and Western notation (*e.g.*, '1,000' could represent 1.000 in Indonesian convention or 1,000 in Western convention); in practice, IPOS consistently uses full Indonesian format with multiple period separators, but this edge case is not explicitly tested. Fourth, no user acceptance testing phase was conducted, meaning the API surface has not been validated against the expectations of actual UMKM operators who would interact with the service through a front-end application.

All five research objectives were achieved. A four-layer Spring Boot MVC architecture was designed and fully implemented, validated through TC01. Ten purpose-specific DTO classes were implemented and verified to enforce phase-level data contracts at each pipeline boundary. Six normalization steps via four utility classes were implemented and validated across TC03–TC05 and TC08. Functional correctness was evaluated across

ten representative IPOS data quality scenarios with a 100% pass rate. Processing performance was characterized across file sizes from 50 to 1,000 receipts, establishing that the pipeline completes within 30 seconds for typical UMKM exports. The identified research gap—the absence of a single study combining programmatic DTO design, MVC layering, and multi-step normalization for IPOS Excel exports within one Spring Boot service—is addressed by this implementation, which provides a complete, deployable, and maintainable reference architecture for UMKM retail data migration.

5. Conclusion and Recommendations

This study designed and implemented a Spring Boot-based ETL service for transforming unstructured IPOS point-of-sale Excel exports into a normalized PostgreSQL relational database. The implementation applied the Data Transfer Object (DTO) pattern across all three ETL phases through ten purpose-specific DTO classes structured within a four-layer MVC architecture (Controller-Service-Repository-Entity), enforcing phase-level data contracts throughout the pipeline (Objectives 1 and 2). Six normalization steps were applied via four reusable utility classes—StringNormalizer, DateParser, NumberParser, and a HashSet-based duplicate detector—resolving the primary data quality challenges characteristic of IPOS exports: heterogeneous date formats, Indonesian-format numeric strings, embedded unit strings in quantity fields, and cross-file and intra-file duplicate receipts (Objective 3). Functional correctness was confirmed across ten representative test scenarios with a 100% pass rate (Objective 4), and processing performance was characterized under realistic file sizes, showing 5–15 second completion times and sub-256 MB heap usage for files of 200–500 receipts (Objective 5). Functional evaluation across ten controlled test scenarios covered valid file processing, invalid file type rejection, seven date-format patterns, embedded-unit quantity parsing, Indonesian numeric disambiguation, cross-file and intra-file duplicate detection, grand-total reconciliation, product-variation tracking, and partial validation error handling. Performance observation across five file sizes showed that files of 200–500 receipts completed processing in 5–15 seconds, corresponding to throughputs of 29–38 receipts per second at steady state. Memory consumption remained below 256 MB through the use of Apache POI's streaming SAX-based reader, which avoids loading the full workbook into heap memory. Taken together, these results indicate that a DTO-centric, explicitly mapped ETL pipeline over Spring Boot MVC constitutes a maintainable, auditable, and production-ready solution for UMKM retail data integration.

The primary contribution of this study is a concrete, working implementation showing how the DTO pattern, applied at phase granularity across all three ETL stages, enforces strict data contracts and produces a fully auditable transformation pipeline. Unlike prior ETL studies for retail e-commerce data (Hutabalian *et al.*, 2024; Saputra, 2024), which operated on structured API outputs or cloud-scraped data with stable schemas, this service addresses the specific structural variability of file-based IPOS Excel exports—dynamic column positions, multi-row receipt encoding, and locale-dependent formatting. Unlike high-level ETL pipeline studies (Aghili *et al.*, 2023; Ginting & Rahmatulloh, 2025), which focus on workflow automation and system-level effectiveness without addressing file-based format variability, this study delivers an implementation-level artifact with measurable functional and performance results. The ten-DTO design provides field-level traceability from raw String extraction through typed transformation to JPA entity persistence—a degree of auditability absent from all surveyed related works and particularly consequential for financial data migration contexts. This combination of programmatic DTO design, four-layer MVC structuring, and multi-step normalization within a single lightweight Spring Boot service constitutes the novel contribution of this study and directly addresses the research gap identified in the introduction.

Four limitations warrant acknowledgment. First, all test files were constructed as controlled scenarios rather than longitudinal production IPOS exports, meaning the test corpus may not capture edge cases arising from IPOS version upgrades, regional locale settings, voided receipts, or corrupted exports observable only over extended real-world use. Second, performance measurements were conducted on a single development machine (Intel Core i5, 8 GB RAM, PostgreSQL 15 on localhost) rather than a networked or multi-tenant deployment environment; network I/O latency and concurrent database access would likely reduce observed throughput. Third, the NumberParser last-separator heuristic fails to disambiguate single-separator numeric strings such as '1,000', which could represent either 1.000 in Indonesian format or 1,000 in Western format; such values are currently flagged as VALIDATION_ERROR rather than resolved. Fourth, no user acceptance testing phase was conducted; the API surface was designed based on the anticipated needs of downstream reporting tools but was not validated against actual UMKM operators.

Four directions for future research are recommended. First, the test corpus should be extended using longitudinal production IPOS exports from multiple UMKM stores across different IPOS versions and regional settings, to expose edge cases not captured in controlled scenarios. Second, the NumberParser ambiguity heuristic should be replaced with a context-aware resolver that uses the column's declared unit type and the surrounding receipt's known value ranges to disambiguate single-separator strings, which would eliminate the

only known category of irresolvable parsing errors. Third, the service could be extended with a Spring Batch-based retry mechanism or message broker integration (*e.g.*, RabbitMQ) to provide automatic retry on transient load failures, improving production reliability beyond the current best-effort async model. Fourth, the ETL service could be evaluated across a broader class of IPOS-like POS systems used by Indonesian UMKM retailers—such as iReap and Moka POS—to determine whether the ColumnLayout keyword-scan and normalization pipeline generalize to other export formats or require system-specific adapters.

Acknowledgment

The authors thank the Faculty of Information Technology, Universitas Kristen Satya Wacana, for providing the research environment and technical support during this study. The authors also acknowledge the constructive feedback received during internal review.

References

Aghili, S., Asadi, S., Shukur, Z., & Nematbakhsh, M. A. (2023). Automating extract, transform, load (ETL) pipelines using machine learning triggered workflow optimization. *International Journal of Intelligent Systems and Applications in Engineering*. https://doi.org/10.18178/ijisae.2023.11.2.7193

Ayet, A., Marti-Carvajal, A. J., Agreda-Perez, L. H., & Sola, I. (2023). Data extraction and comparison for complex systematic reviews: A step-by-step guideline. *Systematic Reviews*. https://doi.org/10.1186/s13643-023-02381-4

Barahama, A. D., & Wardani, R. (2022). Analysing user reviews with ETL using Pentaho Data Integration. *Multimedia Tools and Applications*. https://doi.org/10.1007/s11042-022-12410-4

Filho, N. (2025). *Best practices for using DTOs (Data Transfer Objects) in a clean architecture*. Zenodo Repository. https://doi.org/10.5281/zenodo.14531413

Ginting, M., & Rahmatulloh, A. A. (2025). Analysis of the effectiveness of data warehousing and ETL in management information systems using the neural networks method. *Telematika: Jurnal Informatika dan Teknologi Informasi, 22*(3), 86–97. https://doi.org/10.31315/telematika.v22i3.14027

Gobin, M., Duclos, A., Jannot, A. S., & Prouteau, A. (2025). From data extraction to analysis: A comparative study of capabilities in scientific literature. *Frontiers in Artificial Intelligence*. https://doi.org/10.3389/frai.2025.1587244

Gupta, V. K. (2025). Building robust REST APIs with Spring Boot. *World Journal of Advanced Engineering Technology and Sciences*. https://doi.org/10.30574/wjaets.2025.15.3.1078

Hartono, M. A., & Mailoa, E. (2024). Pembuatan website REST API dengan kombinasi framework Spring Boot dan MyBatis Generator. *Jurnal Indonesia: Manajemen Informatika dan Komunikasi*. https://doi.org/10.35870/jimik.v5i2.712

Hayes, M., Cruz, J. D., & Santos, R. A. (2024). Point of sale's transaction data: Envisioning micro small and medium enterprise (MSME)'s inventory management strategy. *Management Journal*. https://doi.org/10.60016/majcafe.v32.25

Hutabalian, P., Ginting, B. R. P., Zaidan, M. Q., Alfisyahrina, N., & Rozikin, C. (2024). Implementasi pipeline ETL/ELT dan model dimensional untuk analisis penjualan Shopee menggunakan PostgreSQL, Docker, dan Apache Superset. *JITET (Jurnal Informatika dan Teknik Elektro Terapan), 13*(3S1). https://doi.org/10.23960/jitet.v131351.8093

Ivanov, A., Kovalev, S., Petrov, M., & Smirnov, D. (2024). The analysis of customers' transactions based on POS and RFID data using big data analytics tools in the retail space of the future. *Applied Sciences, 14*(24), 11567. https://doi.org/10.3390/app142411567

Izonin, I., Tkachenko, R., Shakhovska, N., Ilchyshyn, B., & Singh, K. K. (2022). A two-step data normalization approach for improving classification accuracy in the medical diagnosis domain. *Mathematics, 10*(11), 1942. https://doi.org/10.3390/math10111942

Jánki, Z. R., & Bilicki, V. (2023). The impact of the Web Data Access Object (WebDAO) design pattern on productivity. *Computers, 12*(8), 149. https://doi.org/10.3390/computers12080149

Kurniawan, R. (2025). Implementation of point of sale (POS) systems in culinary MSMEs: Improving operational efficiency. *JOINTECS (Journal of Information Technology and Computer Science), 8*(1). https://doi.org/10.31328/jointecs.v8i1.7330

Li, X., Zhang, Y., Wang, J., & Liu, H. (2022). Handling unstructured Excel data via automated normalization in ETL pipelines. *IEEE Access*. https://doi.org/10.1109/ACCESS.2022.3159021

Pandey, A., Sharma, R., Singh, P., & Kumar, V. (2025). Building robust REST APIs with Spring Boot: A practical guide. *Journal of Computer Science and Technology Studies*. https://doi.org/10.32996/jcsts.2025.7.4.120

Ramadugu, G. (2024). Microservices with Spring Boot: Simplifying distributed systems. *International Journal For Multidisciplinary Research*. https://doi.org/10.36948/ijfmr.2024.v06i05.28930

Saputra, D. A. (2024). *Penerapan data pipeline guna kebutuhan analisis produk elektronik pada website e-commerce Tokopedia* [Undergraduate thesis, STT Terpadu Nurul Fikri].

Singi, K. R. (2023). Performance optimization strategies for high-concurrency Spring Boot microservices in enterprise financial systems. *The Eastasouth Journal of Information System and Computer Science*. https://doi.org/10.58812/esiscs.v1i02.883

Supriyanto, A. (2024). Efficiency comparison in prediction of normalization with data mining classification. *Advances in Science, Technology and Engineering Systems*. https://doi.org/10.25046/aj060415

Suryadana, I. G. A., Dewi, N. K. C., & Pratama, I. M. A. (2024). Descriptive analytics sales data visualization with ETL. *Technovate*. https://doi.org/10.59890/technovate.v1i2.49

Sutanto, B. (2024). Optimizing decision making in MSMEs through business intelligence dashboards and POS integration. *Seminar Nasional Inovasi Teknologi*. https://doi.org/10.31294/snit.v1i1.8634

Syaputra, H., Apriyandi, D., Husin, M. N., & Desnelita, Y. (2023). Database consistency improvement in job offering system using normalization method. *Jurnal Teknologi dan Open Source*. https://doi.org/10.36378/jtos.v6i1.4022

Trisnawati, E., Susanto, T. D., & Handayani, P. W. (2024). Generating user personas for eliciting requirements using online data from point-of-sale patterns. *Journal of Information Systems Engineering and Business Intelligence, 10*(1), 110–125. https://doi.org/10.20473/jisebi.10.1.110-125

Vieira, R., Souza, L., Lima, M., & Costa, F. (2024). From DTO to ViewModel: Mapping strategies between layers in C# and Java. *Leaders Tec*. https://doi.org/10.5281/zenodo.11186754

Yalamati, S. S. A. (2025). Resilient microservice patterns using Java 17 and Spring Boot 3.2 in cloud-native systems. *International Journal of Science and Research Archive*. https://doi.org/10.30574/ijsra.2025.16.3.2559.