Open-Source AI-Driven Solution for Supply Chain Purchase Order Data Management

Executive Summary

This report outlines a robust, open-source AI-driven solution designed for supply chain companies to automate the extraction, transformation, and management of purchase order (PO) data from PDF manifest orders. By leveraging a strategic combination of specialized PDF parsing libraries, local Large Language Models (LLMs) for intelligent data extraction, a structured relational database, and a user-friendly interface, this solution aims to significantly reduce manual effort, improve data accuracy, and enhance operational efficiency. The proposed architecture emphasizes modularity, scalability, and maintainability, ensuring a future-proof system that minimizes reliance on paid services for critical supply chain data processing.

Understanding the Supply Chain Data Challenge

Purchase order data, frequently embedded within diverse PDF manifest formats, presents a substantial challenge for conventional data entry and management processes. These documents can vary widely, ranging from digitally native, structured PDFs to scanned images that feature complex layouts, intricate tables, and inconsistent data presentation. The current manual extraction process is not only prone to errors but also consumes considerable time, thereby impeding real-time data analysis and informed decision-making. The primary objective of this custom software solution is to automate this laborious process, ensuring high accuracy, structured data storage, and efficient retrieval based on internal PO numbers.

A critical aspect to consider is the inherent operational cost associated with manual data processing. While the explicit request is to avoid paid software services, the

implicit expenses of human labor, the time consumed, the frequency of errors, and the delays in obtaining actionable insights represent a significant financial burden. Automating document processing, as evidenced in similar applications for invoices, has been shown to reduce costs by up to 80% and decrease processing time from minutes to mere seconds, leading to substantial efficiency gains.[1] Therefore, framing this open-source solution not merely as a way to circumvent software license fees, but as a strategic investment to drastically cut operational overhead and unlock the strategic value of data currently confined within unstructured PDF manifests, is essential. This approach transforms the project from a simple cost-cutting exercise into a pivotal enabler for advanced data utilization within the supply chain.

Phase 1: Robust PDF Data Extraction

The initial and most critical phase of this solution involves the reliable extraction of data from PDF manifest orders. This requires a nuanced approach that differentiates between digitally native PDFs, where text can be directly accessed, and scanned documents, which necessitate Optical Character Recognition (OCR).

PDF Parsing Techniques: Native Text Extraction vs. OCR

For digitally native PDFs, libraries can directly access and extract text along with associated metadata. This method is generally faster and yields higher accuracy. Conversely, for scanned documents or image-based PDFs, OCR technology is indispensable for converting images of text into machine-readable characters. A hybrid approach is frequently necessary because manifest orders often contain a mix of digital text and scanned elements, such as stamps, signatures, or handwritten annotations.

PDF structures are notoriously complex, often lacking the semantic tags that facilitate easy machine interpretation, making accurate extraction inherently challenging.[2] Furthermore, OCR models like Tesseract, while powerful and widely used, can encounter difficulties with complex layouts, multiple columns, tables, or low-quality input images, which can compromise accuracy and performance.[3]

A deeper examination of PDF parsing reveals a fundamental trade-off between speed and simplicity on one hand, and the preservation of document layout and structure on the other. Libraries such as PyPDF2 offer rapid, basic text extraction but often discard critical formatting and structural information.[5] In contrast, tools like PDFplumber and Camelot are specifically designed to excel at extracting structured data, particularly tables.[6] The inherent lack of semantic tags in many PDFs means that basic text extraction methods often strip away valuable layout cues, making subsequent interpretation difficult.[2] This characteristic of PDF documents strongly suggests that a single, all-encompassing tool is unlikely to be sufficient. Instead, a multi-tool pipeline will be required, where different libraries handle distinct aspects of the PDF (e.g., one for raw text, another for tables, and OCR for scanned portions). The extracted fragments would then be fed into an LLM for unification and intelligent interpretation, necessitating robust error handling and validation at each stage of this pipeline.

Open-Source Python Libraries for PDF Handling

A variety of open-source Python libraries are available to address the diverse challenges of PDF data extraction:

- **General Text Extraction:**
  - PyPDF2 (now integrated into PyPDF) is a pure Python library suitable for basic operations like splitting, merging, cropping, transforming, and extracting text and metadata.[6] It is lightweight and efficient for direct PDF parsing.[9] However, it is less effective for structured data and struggles with hyperlink metadata.[5]
  - pdfminer.six offers robust capabilities for extracting text data and performing layout analysis.[6] It excels at representing PDF content in text blocks with correct reading sequences but provides limited semantic information extraction.[2]
  - pypdfium2 is notable for its exceptional speed in basic text extraction, making it suitable for high-volume processing where the primary concern is speed over detailed structure preservation.[5]
  - PyMuPDF (also known as fitz) provides Python bindings to the MuPDF C library, allowing for the extraction of text, tables, and images, with optional Tesseract OCR support.[7] While proficient at mapping text to bounding boxes, its native hyperlink extraction capabilities require significant customization.[9]
  - Unstructured is a versatile library for preprocessing and ingesting various document types, including images and text. It supports element-wise text

extraction and performs layout analysis, offering both OCR and table extraction features.[7]

- **Table Extraction:**
  - Camelot is specifically designed for extracting tables from PDFs and converting them into pandas DataFrames.[6] It supports both 'stream' and 'lattice' modes to accommodate different table structures.[7]
  - PDFplumber, built upon pdfminer.six, excels at extracting structured data such as tables and forms, and includes a valuable visual debugging tool to aid in the extraction process.[6]
  - Open Parse is tailored for visually discerning document layouts and chunking them effectively for LLM processing. It offers high-precision table support, converting tables into clean Markdown formats, and can integrate with machine learning table detection models like unitable.[10] It also supports OCR functionality via Tesseract.[10]
- **OCR for Scanned Documents:**
  - Tesseract OCR is a widely recognized, highly accurate, cost-effective, and customizable open-source OCR engine.[3] It is excellent for converting scanned documents and image-based PDFs into editable text.[3] However, its performance can be challenged by complex layouts (e.g., multiple columns, tables, non-standard text) and may face accuracy or speed issues with large volumes or low-quality images.[3] It requires the pdf2image library for direct PDF processing.[4]

The selection of appropriate PDF parsing tools will necessitate a careful evaluation of document types and complexity. The following table provides a comparative overview of these libraries, highlighting their strengths and limitations for purchase order data extraction.

Table 1: Comparison of Open-Source PDF Extraction Libraries

Library	Primary Function	Strengths	Weaknesses/Limitations	Best Use Case for PO Data
PyPDF2 (PyPDF)	General PDF manipulation, text/metadata extraction	Fast, pure Python, lightweight, efficient for basic text [6]	Struggles with structured data, no table extraction, poor hyperlink metadata [5]	Simple text content, merging/splitting PDF pages

pdfminer.six	Text extraction, layout analysis	Robust text extraction, analyzes text layout, provides text blocks with reading sequences [6]	Primitive semantic information extraction, no direct table extraction [2]	Detailed text content analysis, understanding document flow
pypdfium2	Basic text extraction	Extremely fast, high-volume processing [5]	No formatting preservation, no table structure [5]	High-speed, basic text extraction for simple content indexing
PyMuPDF (fitz)	Text, table, image extraction; OCR support	Python bindings to fast C library, good for bounding boxes, optional OCR [7]	Hyperlink extraction requires customization [9]	General extraction, integrating OCR for mixed digital/scanned content
Unstructured	Preprocessing, element-wise text, layout analysis, OCR, table extraction	Supports diverse document types, layout analysis with detectron2, OCR and table features [7]	May require additional configuration for specific layouts	Pre-processing documents for LLM input, complex document types
Camelot	Table extraction	Specialized for tables, outputs to Pandas DataFrames, offers 'stream' and 'lattice' modes [6]	Primarily for tables, not general text extraction	Extracting structured tables of line items from POs
PDFplumber	Structured data extraction (tables, forms)	Built on pdfminer.six, excels at tables/forms, visual debugging tool [6]	Can have spacing quirks in basic text [5]	Extracting structured tables and form-like data from POs

Open Parse	Visually-driven document parsing for LLMs	Visually analyzes layouts, high-precision tables (Markdown), basic Markdown support, extensible [10]	Requires Tesseract for OCR, ML table detection can be subpar [10]	Optimizing PDF content for LLM input, complex layouts, accurate table conversion to Markdown
Tesseract OCR	Optical Character Recognition (OCR)	High accuracy for printed text, cost-effective, customizable, multilingual [3]	Struggles with complex layouts, performance issues with low quality/volume, needs pdf2image for PDFs [3]	Converting scanned POs or image-based text sections into machine-readable text

This table highlights that a combination of these tools will likely yield the best results. For instance, Tesseract OCR would handle scanned images, while PDFplumber or Camelot would extract tables from digital PDFs. pdfminer.six or PyPDF2 could handle general text extraction, and Open Parse could provide a unified, LLM-optimized representation of the document's layout.

Phase 2: Intelligent Information Extraction with Open-Source LLMs

Following the initial PDF parsing, Large Language Models (LLMs) become indispensable for intelligently extracting specific, structured data points such as PO numbers, individual line items, and vendor details from the potentially semi-structured or unstructured text.

Leveraging LLMs for Structured Data: Principles and Capabilities

LLMs possess the capability to process substantial volumes of unstructured text and convert it into structured formats.[11] They are particularly adept at identifying

semantically significant details, including names, dates, and numerical figures, and presenting them in a consistent structure.[11] This ability is highly valuable for transforming natural language into formats that computer programs can readily consume, often by intelligently disregarding extraneous information.[11] Furthermore, LLMs can be effectively employed for tasks such as named entity recognition and the identification of crucial dates within documents.[12]

The application of LLMs elevates the extraction process beyond simple keyword or regular expression matching to a more robust, semantic understanding of the document content. This capability is critically important for handling purchase orders, where variations in phrasing, layout, and even synonyms for line items or vendor names are common. Traditional rule-based extraction methods often fall short in such scenarios. Documents like invoices and contracts frequently lack explicit semantic tags, making their interpretation challenging for automated systems.[2] LLMs, with their advanced Natural Language Processing (NLP) capabilities, can infer meaning and structure even from text that appears unstructured, a common characteristic of POs originating from diverse suppliers. This semantic understanding is key to managing the inherent variability and "noise" found in real-world supply chain documents, leading to significantly higher accuracy and a reduced need for manual correction compared to systems relying solely on predefined rules.

LLM Coding Structures: Prompt Engineering and Pydantic

The effectiveness of LLMs in data extraction is profoundly influenced by the quality of the prompts provided.

- **Prompt Engineering:** Designing and refining the inputs given to LLMs is crucial for achieving desired outputs.[14]
    - **Best Practices:** Instructions should be clear, concise, and specific, ideally placed at the beginning of the prompt.[15] Using delimiters, such as ### or """, to separate instructions from the main context is highly recommended.[16]
    - **Few-Shot Examples:** Supplying a few high-quality input-output examples helps the model grasp the desired data type and ensures consistency in its responses.[15] These examples serve as a template for the model to follow.[15]
    - **Schema Definition:** Explicitly defining the structure of the data to be extracted, often in a JSON schema format, is paramount for maintaining

consistency and accuracy in the LLM's output.[15]
    - **Iterative Refinement:** Prompt design is an iterative process. It is advisable to start with simple prompts and gradually refine them by adding more context and elements as needed.[18]
- **Pydantic for Structured Outputs:**
    - Pydantic is a Python library that enables runtime data validation using type annotations.[11] It facilitates the definition of data models using standard Python type hints, ensuring that all incoming data conforms to the specified schema.[11]
    - A key feature of Pydantic is its ability to convert Python classes into JSON-serialized schema objects. These schemas are then passed to LLMs, providing explicit instructions on how to format and return the extracted data.[11]
    - The benefits of using Pydantic include enhanced consistency, improved reliability, stronger security through validation, automatic data parsing and serialization, and seamless integration with Python's type hinting system.[21]
    - Pydantic models can be nested, allowing for the creation of highly complex and hierarchical data structures.[11]
    - Including natural-language descriptions via docstrings for classes and Field objects for individual fields further enhances LLM accuracy by providing additional context about the expected data.[11]
    - Pydantic is also crucial for validating AI-generated data and can be integrated into retry mechanisms for handling failed validation attempts, thereby improving the robustness of LLM-based pipelines.[19]

Pydantic functions as a formal contract between the LLM and the downstream system, not merely a validation tool. This architectural pattern is vital for the reliable integration of LLMs, particularly when dealing with sensitive data like purchase orders. It enables robust error handling and ensures data quality before data is committed to storage. The use of Pydantic ensures that even if an LLM occasionally generates malformed or inconsistent output, the system can detect these issues, potentially trigger a retry with refined instructions, or flag the data for human review. This proactive validation prevents erroneous data from corrupting the database, which is paramount for maintaining data integrity in a supply chain context.

Local LLM Integration and Frameworks

Running LLMs locally offers significant advantages, primarily by avoiding cloud API

costs and keeping sensitive supply chain data securely within the company's infrastructure. Ollama is a prominent open-source tool that facilitates running various LLMs, such as Llama 3.2 and Granite 3.2, on a local machine.[19] Notably, Ollama supports structured outputs, which is critical for programmatic data extraction.[17]

Several open-source frameworks are available to streamline LLM-based extraction:

- ContextGem: This free, open-source LLM framework is designed to simplify structured data extraction from documents with minimal code. It offers built-in abstractions for automated dynamic prompts, data modeling, validators, granular reference mapping, and a unified extraction pipeline, significantly reducing development overhead.[23]
- LangExtract: An open-source Python library that processes large volumes of unstructured text into structured information using various LLMs, including open-source on-device models.[24] It provides precise source grounding by mapping extracted entities back to their exact character offsets, ensuring reliable structured outputs through few-shot examples and controlled generation. It is also optimized for handling long contexts in documents.[24]
- LlamaIndex: This framework enables LLMs to ingest vast amounts of unstructured data and return it in structured formats.[11] It integrates seamlessly with Pydantic for defining output schemas and supports various LLM backends, including local ones. LlamaIndex can also be effectively utilized within Retrieval Augmented Generation (RAG) pipelines.[11]
- Unstract (Open-Source Edition): A no-code, AI-powered platform that combines intelligent document processing with robotic process automation, enhanced by LLMs and advanced OCR techniques. The open-source edition can be deployed locally using Docker and Docker Compose.[25] It includes features like LLMChallenge for comparing LLM performance, SinglePass Extraction for optimizing prompts by combining them, and LLMWhisperer for document preprocessing to ensure LLMs can effectively understand varied document designs and formats.[25]

The availability of multiple open-source frameworks for LLM-based extraction, coupled with local LLM runners like Ollama, signifies a maturing ecosystem. This abundance of tools reduces vendor lock-in and allows for greater customization and cost control, directly addressing the requirement to minimize reliance on paid services. The existence of distinct frameworks, each with specialized strengths—for instance, ContextGem for reducing boilerplate, LangExtract for grounding and traceability, LlamaIndex for RAG and Pydantic integration, and Unstract for comprehensive Intelligent Document Processing workflows—indicates that the

open-source community is actively developing tailored solutions for various aspects of LLM-powered document processing. This rich ecosystem means that a supply chain company can construct a highly customized solution without building from scratch, leveraging specialized tools for each segment of the extraction pipeline (parsing, LLM interaction, validation, and potential RAG). This also lowers the barrier to entry for in-house development compared to previous years.

Fine-tuning Smaller LLMs (LoRA): When and How to Fine-tune for Domain-Specific Accuracy

Fine-tuning involves providing additional training to a pre-existing LLM using a smaller, domain-specific dataset. This process aims to enhance the model's accuracy for a particular task.[12] For purchase order data, this is particularly valuable for tasks such as named entity recognition and the precise identification of dates within the specific context of the supply chain industry.[13]

LoRA (Low-Rank Adaptation) is a cost-effective fine-tuning method that significantly reduces the number of trainable parameters, sometimes to as little as 0.01% of the full model's parameters.[12] Instead of adjusting all the weights of a large pre-trained model, LoRA fine-tunes two smaller matrices that approximate the larger weight matrix.[12] The outcome of this process is a compact "LoRA adapter" that can be combined with the original LLM during inference. This architecture allows multiple LoRA adapters to reuse the same base LLM, thereby reducing overall memory requirements when handling diverse tasks.[12]

The fine-tuning process typically involves selecting a suitable pre-trained model (e.g., Llama 3.1), gathering a high-quality, relevant, and domain-specific dataset (such as labeled purchase orders), preprocessing this data, and then performing the fine-tuning using frameworks like Hugging Face's Transformers library, often with tools like Axolotl.[12] This method can yield substantial performance gains, with reported improvements ranging from +36% to +67% in financial domain tasks.[27] It enables the model to better comprehend specialized terminology and domain-related nuances, thereby minimizing the generation of imprecise information.[26] However, it is important to note that this approach requires high-quality, representative data and, while LoRA makes it more feasible, it remains computationally intensive.[26] Continuous monitoring and evaluation are essential to ensure the fine-tuning process is successful and the

refined LLM performs as expected.[13]

While prompt engineering with few-shot examples can achieve satisfactory results for many extraction tasks, fine-tuning, especially with LoRA, offers a strategic pathway to attain higher, domain-specific accuracy for the unique terminology, abbreviations, and layouts prevalent in purchase orders. This represents a trade-off: initial setup complexity versus long-term accuracy and reduced post-processing effort. For a critical business function like PO data extraction, this level of precision can provide a distinct competitive advantage. The ability of fine-tuned models to better understand specialized terminology and nuances [26] means they can accurately interpret specific fields, common abbreviations, and even typical errors unique to a company's suppliers. LoRA makes this level of precision computationally feasible without incurring the prohibitive costs associated with training a model from scratch.[12] This enables the achievement of "enterprise-grade" accuracy through an open-source strategy. Ultimately, high accuracy in PO data directly translates to fewer errors in inventory management, payment processing, and supply chain planning, thus enhancing overall operational integrity.

Phase 3: Data Transformation and Normalization

Raw extracted data frequently contains inconsistencies, formatting issues, or variations that require standardization before storage and analysis. This phase focuses on cleaning and normalizing the data to ensure its utility and reliability.

Data Cleaning and Manipulation

Common issues encountered in extracted text include extraneous whitespaces, inconsistent capitalization, special characters, and duplicate entries.[28] Python offers robust libraries for addressing these challenges.

- Pandas: This powerful open-source library is a cornerstone for data analysis, manipulation, and cleaning. It provides DataFrames, which are highly efficient for handling tabular data, and offers extensive tools for reading and writing data across various formats, including CSV, Excel, and SQL databases. Pandas also

includes robust capabilities for grouping and transforming data.[28]

- **Techniques:** Standard string manipulation methods are employed for cleaning. This includes converting text to lowercase using .lower(), removing numerical digits with regular expressions (re.sub), eliminating punctuation (re.sub), and stripping unwanted whitespaces (.strip(), re.sub).[28] Duplicate rows can be efficiently identified and removed using the .drop_duplicates() function.[28]

The quality of downstream database operations and search functionality is heavily dependent on the cleanliness and normalization of the data, even when utilizing advanced LLM extraction. This phase functions not merely as a technical step but as a critical quality control gate. Inconsistent data, such as variations in how a purchase order number is represented (e.g., "PO No." versus "Purchase Order #"), will inevitably hinder search capabilities and analytical processes, regardless of the LLM's extraction prowess. The process of normalizing text before storage or further processing ensures consistency in the input, which is vital for subsequent operations.[29] If critical identifiers like PO numbers, vendor names, or item descriptions are not standardized, searching for "ABC Corp" will fail to retrieve documents referencing "ABC Corporation" or "ABC Co.", and database joins will produce incomplete results. This phase ensures that the valuable data extracted by the LLM is truly usable and reliable for the database and search functionalities, thereby preventing "garbage in, garbage out" scenarios that would undermine the entire system's value.

String Normalization and Fuzzy Matching

The purpose of string normalization and fuzzy matching is to identify and reconcile near-matches or variations in text data, such as different spellings or abbreviations for vendor names like "Supplier Inc." versus "Supplier Incorporated."

- **Fuzzy String Matching:** This technique determines the approximate similarity between two strings, often by calculating the Levenshtein distance—the minimum number of single-character edits (insertions, deletions, or substitutions) required to transform one string into another.[31]
- **Python Library:** TheFuzz (formerly FuzzyWuzzy) is a popular open-source Python library for fuzzy string matching.[31] It provides methods such as ratio(), partial_ratio(), and token_sort_ratio() to compute similarity scores between strings, allowing for flexible matching.[31]

- **Applications:** Fuzzy matching is widely used for deduplication, linking data from disparate sources, and spell checking.[31]
- **NLP for Entity Resolution:** For more intricate cases, Natural Language Processing (NLP) techniques, leveraging libraries like spaCy or NLTK, can be employed for tokenization, stemming (e.g., token.lemma_), and identifying common multi-word phrases. More advanced methods, such as word2vec embeddings or even LLMs, can further assist in identifying semantic synonyms or conceptually similar entities.[33]

Fuzzy matching and NLP for entity resolution are crucial because human data entry, or even variations from source systems, often introduce minor differences that machines interpret as distinct entities. This phase directly addresses the inherently messy nature of real-world supply chain data, ensuring that "Acme Corp" and "Acme Corporation" are recognized as the same entity. This capability is vital for accurate reporting and search functions. Standard deduplication methods, which rely solely on exact matches, are insufficient for these scenarios.[31] For instance, a purchase order number might appear as "PO-XYZ-001" on one document and "P.O. XYZ001" on another. Without fuzzy matching, the search system would fail to retrieve all relevant documents. This capability directly impacts the usability and completeness of the search system. If a user searches for a PO number and does not retrieve all matching documents due to minor variations, the system's credibility and efficiency gains are compromised. This is a key enabler for the user's requirement of a "simple UI for searching and retrieving matching PDF files based on internal PO numbers."

Pattern Matching and Data Extraction (Regex)

Regular expressions (Regex) are invaluable for extracting specific, highly structured patterns that adhere to predictable formats, such as dates, currency amounts, or specific product codes.

- **Python Library:** Python's built-in re module is the standard library for working with regular expressions.[36]
- **Functions:** The re module offers a suite of functions including re.search(), re.match(), re.findall(), and re.sub(), which enable searching, matching, and manipulating strings based on defined patterns.[36]
- **Use Cases:** Regex is indispensable for tasks like email validation, phone number extraction, and identifying specific numerical configurations.[36]

- **Integration:** It can be seamlessly integrated with Pandas for column-wise data cleaning and filtering, allowing for targeted transformations on structured dataframes.[36]

While LLMs excel at understanding context and extracting information from unstructured text, regex remains an invaluable tool for enforcing strict formats and precisely extracting highly predictable data points, either as a post-processing step on LLM output or as a pre-processing step. It serves as a "precision scalpel" to clean and validate specific fields where strict adherence to a pattern is required, such as a specific PO number format or a date format. LLMs, despite their power, can occasionally produce slightly off-format outputs or "hallucinate" data. Regex, when integrated into Pydantic validators or used independently, can serve as a robust validation layer for fields that must conform to a specific pattern.[21] This combination of the semantic understanding provided by LLMs and the deterministic precision of regex creates a highly robust extraction pipeline that balances flexibility with the strict data quality requirements essential for critical fields in a supply chain context.

Phase 4: Structured Database Storage

Storing the extracted and transformed purchase order data in a structured database is fundamental for efficient querying, comprehensive reporting, and seamless integration with other internal business systems.

Choosing an Open-Source Relational Database: PostgreSQL vs. SQLite

The choice of database is a critical architectural decision that impacts scalability, performance, and data integrity.

- **SQLite:**
  - **Architecture:** SQLite operates as a serverless, self-contained database engine, where the entire database resides in a single .sqlite file.[38]
  - **Setup:** It offers zero-configuration, requiring no installation or complex user permissions, making it exceptionally easy to set up.[38]
  - **Concurrency:** SQLite has limited write concurrency, typically supporting only

a single writer at a time, and is not designed for high concurrency environments.[39]
    - **Performance:** It performs quickly for simple read operations and is suitable for small to medium-sized websites, capable of handling up to 100,000 hits per day.[38]
    - **Use Cases:** Ideal for embedded systems, mobile applications, lightweight workloads, file archives, and local data storage scenarios.[38]
- **PostgreSQL:**
    - **Architecture:** PostgreSQL is an object-relational database management system that operates on a client-server model.[38]
    - **Setup:** It requires more extensive installation and setup procedures compared to SQLite.[38]
    - **Concurrency:** PostgreSQL employs a Multi-Version Concurrency Control (MVCC) architecture, which allows for multiple concurrent reads and writes without locking, ensuring high concurrency.[38]
    - **Performance:** It offers superior speed for complex queries and is highly scalable, making it well-suited for large workloads and high-traffic web applications.[38]
    - **Features:** PostgreSQL boasts a rich set of advanced SQL features, including recursive queries, window functions, and native JSON handling (JSONB). It also provides advanced indexing capabilities and robust disaster recovery mechanisms such as Write-Ahead Logging (WAL) and replication, along with checksums for data page corruption detection. Its extensibility allows for custom functions, procedural languages, and extensions, and it offers full-text search capabilities.[38]
    - **Data Integrity:** It enforces strong data integrity through various mechanisms, including strict data types, comprehensive constraints (PRIMARY KEY, FOREIGN KEY, NOT NULL, CHECK, UNIQUE), and triggers that automate tasks in response to database events.[40]
    - **Use Cases:** PostgreSQL is the preferred choice for enterprise applications, complex web services, data warehousing and analytics, Geographic Information Systems (GIS), telecommunications, and database consolidation.[39]

The decision between SQLite and PostgreSQL for an "in-house, custom software solution for a supply chain company" represents a fundamental choice between simplicity and future-proofing. While SQLite offers unparalleled ease of use for initial prototyping or very small-scale applications, PostgreSQL is the unequivocally superior choice for a production-grade system due to its robust concurrency, advanced features, and inherent scalability. Supply chain data volumes can escalate rapidly, and

multiple users or integrated systems will likely need to access PO data concurrently. SQLite's limitations, such as its "limited write concurrency (single writer)" and its design not being for "high concurrency" [39], make it an unsuitable option for a growing enterprise system. In contrast, PostgreSQL's MVCC architecture [38] and its proven capability for "highly scalable for large workloads" [39] directly address these future demands. Opting for PostgreSQL from the outset, despite its slightly more involved setup, strategically future-proofs the data layer against common scalability bottlenecks and provides a richer feature set for complex queries and ensuring data integrity, which are all critical for efficient supply chain operations.

The following table provides a detailed comparison to guide the database selection for a purchase order management system:

Table 2: Recommended Open-Source Relational Databases

Feature	SQLite	PostgreSQL	Relevance to PO System
Architecture	Serverless, embedded, single .sqlite file [38]	Client-Server, object-relational [38]	PostgreSQL's client-server model is essential for multi-user, networked access in a supply chain environment.
Setup/Admin	Zero-configuration, minimal setup [38]	Extensive installation and setup [38]	While SQLite is simpler to start, PostgreSQL's robust setup is a necessary investment for long-term stability and features.
Concurrency	Limited write concurrency (single writer) [39]	Multi-Version Concurrency Control (MVCC), supports multiple concurrent reads/writes [38]	High concurrency from PostgreSQL is vital as multiple users/systems will interact with PO data simultaneously.
Scalability	Not designed for high	Highly scalable for	PostgreSQL's

	concurrency, limited for large workloads [39]	large workloads and high-traffic applications [39]	scalability is crucial for handling growing volumes of PO data and increasing user demands in a dynamic supply chain.
Performance	Fast for simple reads, moderate workloads [38]	Superior speed for complex queries, optimized for large datasets [38]	Complex queries for analytics (e.g., aggregate spend, supplier performance) will benefit significantly from PostgreSQL's performance.
Data Integrity	ACID compliant, basic features [38]	ACID compliant, advanced constraints (PK, FK, NOT NULL, CHECK, UNIQUE), triggers [38]	PostgreSQL's strong data integrity features ensure accuracy and consistency of critical PO data, enforcing business rules at the database level.
Advanced SQL Features	Full-featured SQL, but less advanced [38]	Recursive queries, window functions, JSONB data type, full-text search [38]	JSONB support is useful for semi-structured PO details, and advanced SQL enables complex reporting.
Use Cases for PO System	Local data storage, small-scale prototyping	Enterprise applications, data warehousing, web services, analytical workloads [39]	PostgreSQL is the appropriate choice for a central, reliable, and scalable PO management system.

Database Schema Design for Purchase Orders

Designing a well-structured database schema is foundational for efficient querying,

robust reporting, and seamless integration of the extracted PO data. The process typically involves defining conceptual, logical, and physical schemas.[42]

- **Conceptual Schema:** This provides a high-level overview, capturing business rules and system requirements without delving into specific database technologies or syntax.[42]
- **Logical Schema:** This translates the conceptual view into concrete database elements, defining tables, columns, data types, and constraints.[42]
- **Key Entities:** For a purchase order management system, essential entities and their attributes would include:
    - PurchaseOrders: PO_ID (Primary Key), OrderDate, SupplierID (Foreign Key), Status (e.g., Pending, Fulfilled, Canceled), TotalAmount, and PDF_FilePath (to link to the original document).
    - LineItems: LineItemID (Primary Key), PO_ID (Foreign Key), ProductID (Foreign Key), Quantity, UnitPrice, and LineTotal.
    - Suppliers: SupplierID (Primary Key), SupplierName, and ContactInfo.
    - Products: ProductID (Primary Key), ProductName, Description, and SKU.
- **Relationships:** One-to-many relationships are common and crucial for purchase order data, such as one purchase order having multiple line items, or one supplier having many purchase orders.[42]
- **Best Practices:** Normalizing the database schema is a best practice to eliminate data redundancy and enhance data integrity.[40] Using appropriate data types for each column is also essential for data quality and efficient storage.[40]

The database schema is not merely a technical detail; it serves as the blueprint for how the extracted PO data will be utilized, queried, and integrated. A meticulously designed, normalized schema directly influences the efficiency of search operations, the accuracy of reports, and the viability of future analytical capabilities, ensuring that the data is not just stored but is truly manageable and actionable. As stated, a well-designed schema "helps surface issues early, identify redundancy, and reduce future storage or compute waste".[42] For purchase orders, this means ensuring that a search for a PO number swiftly retrieves all its associated line items, or that aggregating spending by supplier is straightforward. Normalization is fundamental to achieving this.[40] The schema directly supports the user's requirement for "searching and retrieving matching PDF files based on internal PO numbers" by providing the structured backbone necessary for these queries. It also lays the groundwork for more advanced supply chain analytics in the future.

Storing PDF References

For the storage of PDF files, the recommended best practice is to store the file path to the PDF in the database rather than embedding the entire PDF file as a BLOB (Binary Large Object).[44]

- **Reasons for Storing File Paths:**
  - **Performance:** Storing large files directly within the database can significantly decrease database performance, as databases are optimized for structured data, not large binary objects.[45]
  - **Scalability:** File systems are inherently more efficient and scalable for storing large binary files compared to relational databases.
  - **Accessibility:** Storing file paths allows third-party tools and external applications to directly view or access the PDF files without needing to extract them from the database first.[45]
  - **Management:** It simplifies the management and access of files for other processes, such as archiving or integration with external viewing applications.
- **Data Type:** VARCHAR or TEXT data types are suitable for storing the file path string in the database.[44]
- **Considerations:** It is crucial to ensure that the file share where the PDFs are stored is secure and that the file system is backed up consistently and synchronously with the database. This parallel backup strategy is vital to maintain data integrity and prevent inconsistencies during restore operations, where a database restore might otherwise point to non-existent or outdated PDF files.[45] Additionally, implementing a clear archiving strategy for older PDF files is important for managing storage costs over time.[45]

The decision to store file paths versus BLOBs is not solely about performance; it also concerns managing system complexity and leveraging the strengths of different storage mechanisms. Storing paths implies a distributed storage model, where the database manages structured metadata and the file system manages the large binary files. This distributed approach necessitates careful consideration of backup, recovery, and consistency across these two distinct layers. The operational risk of inconsistent backups or accidental file deletion outside of database control must be explicitly managed. This requires a strong IT operations component to the solution, extending beyond just the development aspects, to ensure the long-term reliability and integrity of the system.

Ensuring Data Integrity

Data integrity is paramount for any robust information system, ensuring that the data contained within the database is consistently reliable, accurate, and consistent.[40] PostgreSQL offers a comprehensive suite of features to enforce this.

- **Importance:** Data integrity prevents invalid data from entering the system, maintains consistent relationships between data entities, and ensures the accuracy of information over time.[40]
- **PostgreSQL Features:**
  - **Data Types:** PostgreSQL allows for precise definition of data types for each column, which inherently limits the kind of data that can be stored, preventing type-related errors.[41]
  - **Constraints:** Constraints are rules enforced on data columns to prevent invalid data from being stored, thereby ensuring that all data adheres to predefined rules.[41] Key types include:
    - PRIMARY KEY: Ensures a unique identifier for each record, preventing duplicate entries.[41]
    - FOREIGN KEY: Maintains referential integrity between two tables by ensuring that values in a column or group of columns match existing values in another table, implementing relationships.[41]
    - NOT NULL: Guarantees that a column cannot contain empty (NULL) values, ensuring completeness of critical data.[41]
    - CHECK: Allows for custom validation rules to be applied to column values (e.g., a quantity must be positive).[41]
    - UNIQUE: Ensures that all values in a column (or set of columns) are distinct across the table.[41]
  - **Triggers:** These are special database functions that are automatically executed by the Database Management System (DBMS) when specific events occur, such as INSERT, UPDATE, or DELETE operations on a table.[40] Triggers are highly useful for enforcing complex business rules or maintaining data consistency across multiple tables.[41]
  - **ACID Compliance:** PostgreSQL strictly adheres to ACID (Atomicity, Consistency, Isolation, Durability) principles.[38] This compliance ensures that database transactions are either fully completed (atomic) or entirely rolled back, maintaining data consistency even in the event of system crashes or

unexpected interruptions.[38]

Beyond simply preventing basic errors, the robust data integrity features within PostgreSQL function as an automated enforcer of business rules. For purchase order data, this ensures, for example, that every line item correctly links to a valid purchase order, that quantities are always positive, and that all supplier IDs correspond to existing supplier records. This capability significantly reduces the need for complex application-level validation logic, centralizing data quality enforcement within the database itself. As noted, triggers are particularly useful for enforcing complex business rules or maintaining consistency across tables.[41] For POs, this could involve automatically updating a

TotalAmount field on the PurchaseOrders table whenever LineItems are added or modified, or ensuring that a Status field can only transition through valid states (e.g., from 'Pending' to 'Fulfilled', but not directly to 'Canceled' without specific conditions). By embedding these business rules directly into the database schema and leveraging features like constraints and triggers, the system becomes more resilient to application-level bugs, ensures consistent data quality regardless of the input source, and simplifies future development by offloading validation logic.

Phase 5: Building the Search and Retrieval System

The ability to quickly search and retrieve purchase order data and their associated PDF files is a core requirement for the solution. This necessitates a dual approach, incorporating both traditional keyword search and more advanced semantic search capabilities.

Keyword-Based Search with Whoosh

For precise, exact, or near-exact keyword matching on extracted text (e.g., searching for a specific PO number, vendor name, or item description), a keyword-based search engine is essential.

- **Python Library:** Whoosh is a fast, feature-rich, pure Python library for full-text

indexing and searching.[46]

- **Features:**
  - **Pure Python:** It requires no compilation or binary packages, making it highly portable and easy to deploy in any Python environment.[46]
  - **Fielded Indexing and Search:** Whoosh allows for the definition of a schema with various field types. For instance, an ID field type is suitable for exact matches like PO numbers, TEXT for searchable body content, KEYWORD for space- or comma-separated terms, and STORED for data that needs to be returned with results but not indexed.[49]
  - **Pluggable Scoring:** It uses the Okapi BM25F ranking function by default, which can be easily customized to suit specific relevance needs.[46]
  - **Powerful Query Language:** Whoosh supports a query language similar to Lucene's, enabling complex searches with operators like AND, OR, NOT, grouping terms with parentheses, and performing range, prefix, and wildcard queries across different fields.[49]
  - **Small Indexes:** It creates relatively compact indexes, optimizing storage usage.[46]
- **Workflow:** The typical workflow involves defining a schema, creating an index, adding documents (containing the extracted PO data), and then performing searches using a Searcher object and a QueryParser.[49]

While semantic search offers advanced capabilities, a robust keyword search system, such as one built with Whoosh, is essential as a reliable fallback, especially for exact matches like purchase order numbers. It provides predictable and precise results for direct queries and handles cases where semantic understanding might be unnecessary or less accurate for specific identifiers. Whoosh's ID field type [49] is perfectly suited for indexing and retrieving exact PO numbers. A hybrid search approach, combining keyword and semantic search, offers the best of both worlds: precision for exact identifiers and conceptual understanding for broader queries, thereby ensuring comprehensive document retrieval for supply chain users.

Semantic Search with Vector Embeddings

Semantic search aims to understand the meaning and context of user search queries, providing more relevant results than traditional exact keyword matches.[50] This is particularly useful for finding POs related to abstract concepts like "urgent deliveries" or "damaged goods," even if those exact phrases are not explicitly present in the

document.

- **Process:**
  - **Generating Embeddings:** The first step involves converting extracted text (from POs, line items, or associated notes) into numerical representations known as vectors. These vectors capture the key features and meaning of the unstructured data.[50] The distance between two vectors quantifies their semantic relatedness.[52]
    Sentence Transformers is an open-source library capable of generating these vectors using various pre-trained models.[51] Alternatively, Ollama's embeddings endpoint can be used with local LLMs to generate these vectors.[53]
  - **Storing Embeddings:** These high-dimensional vectors need to be stored in a specialized vector database or locally for efficient retrieval.[50]
    - OpenSearch Vector Engine: This is an open-source vector database specifically designed for indexing and storing complex unstructured data. It enables fast similarity searches using techniques like k-nearest neighbors.[50] It integrates with leading LLM frameworks and supports real-time data ingestion and indexing.[50]
    - **Local Storage:** For smaller-scale applications or initial prototyping, embeddings can be stored locally, for example, in a CSV file or a NumPy array.[51]
  - **Similarity Search:** When a user submits a search query, that query is also converted into a vector. The system then performs a similarity search (e.g., using cosine similarity or the k-nearest neighbors (KNN) algorithm) to retrieve the most relevant results from the embedding database.[51]

Semantic search fundamentally transforms the search experience from a literal keyword match to a conceptual understanding. For a supply chain context, this means that users can find purchase orders related to a *problem* or a *category* rather than being limited to an exact PO number. This capability is invaluable for investigative tasks or broader analytical needs, moving beyond simple retrieval to intelligent retrieval. For instance, a user might search for "late shipments from Vendor X" or "issues with quality of widgets." A traditional keyword search would struggle unless those exact phrases were present in the document. Semantic search, by understanding the underlying meaning of "late" or "quality issue," can retrieve relevant POs or related documents (such as delivery manifests or inspection reports) that discuss these concepts, even if the phrasing differs. This represents a significant enhancement for operational problem-solving. This capability directly addresses a common pain point in enterprise search: finding documents based on *what they are about* rather than just *what words they contain*. It makes the search UI more powerful

and intuitive for non-technical users, leading to faster problem identification and resolution within the supply chain.

Phase 6: Developing the User Interface (UI)

A simple, intuitive user interface (UI) is essential to enable supply chain personnel to effectively interact with the system, search for purchase orders, view extracted data, and retrieve the original PDF files.

Open-Source Python UI Frameworks

The choice of UI framework depends on the desired balance between rapid development and extensive customization.

- **Rapid Prototyping and Data Dashboards (Streamlit):**
  - **Features:** Streamlit is an open-source Python library specifically designed for building interactive, data-rich web applications using pure Python, eliminating the need for HTML, CSS, or JavaScript knowledge.[54] It provides a variety of built-in interactive widgets (buttons, sliders, dropdowns), supports real-time application updates, and integrates seamlessly with common data visualization libraries like Seaborn and Plotly.[54]
  - **Advantages:** It is exceptionally easy to use, facilitates rapid prototyping, and is ideal for creating data dashboards and internal tools.[54] Its low-code nature makes it highly accessible for data scientists and analysts who primarily work with Python.[54]
  - **Disadvantages:** Streamlit offers less flexibility for developing complex, highly customized web applications compared to full-fledged web frameworks.
- **Full-Fledged Web Applications (Flask or Django with Vue.js or React):** For more complex and customizable web applications, combining a Python backend with a JavaScript frontend is a robust approach.
  - **Backend (Python):**
    - Flask: A lightweight micro-framework known for its flexibility. It is well-suited for smaller projects, offers good scalability, and integrates easily with databases.[56] However, it requires more manual setup for

features not built into its core.[56]
        - Django: A comprehensive, full-featured framework that includes robust security measures, adapts well to heavy-load projects, adheres to a "Don't Repeat Yourself" (DRY) philosophy, and provides built-in components like an admin interface, Object-Relational Mapper (ORM), URL routing, and form handling.[56] Django facilitates faster Minimum Viable Product (MVP) development due to its extensive built-in features.[56] Its learning curve is steeper than Flask's.[56]
    - **Frontend (JavaScript Frameworks):**
        - Vue.js / React: These are popular JavaScript frameworks for building rich, dynamic, and interactive web applications.[58] They promote a component-based architecture, allowing for the creation of reusable UI elements.
        - **Integration:** Python backends (Flask or Django) can expose APIs that the JavaScript frontend consumes via standard fetch requests, enabling seamless data exchange.[60]
    - **Advantages:** This approach offers maximum flexibility, customization, and scalability for applications requiring complex user interactions, multi-page layouts, and intricate workflows.
    - **Disadvantages:** It necessitates knowledge of both Python backend frameworks and JavaScript frontend frameworks, resulting in a steeper learning curve and generally longer development times.[56]

The choice of UI framework directly impacts development speed versus the level of customization and long-term maintainability. Streamlit provides rapid prototyping capabilities for internal tools, which aligns with the initial "simple UI" requirement. However, for a more robust, enterprise-grade application with evolving complex workflows and diverse user roles within a supply chain context, a Flask/Django backend paired with a JavaScript frontend offers superior flexibility and control, albeit requiring more development resources. While Django's steeper learning curve [56] is a factor, for a long-term in-house solution, the benefits of its comprehensive features and robust security might justify the initial investment. A recommended approach could involve starting with Streamlit for a quick MVP to validate the core functionality, then potentially migrating to or building out a more robust Flask/Django + JS frontend as requirements mature or if the "simple UI" needs to evolve into a full-fledged operational portal.

The following table summarizes the characteristics of various open-source UI frameworks for internal tools:

Table 3: Open-Source UI Frameworks for Internal Tools

Framework	Type	Ease of Use/Learning Curve	Interactivity/Features	Suitability for Data Apps/Dashboards	Suitability for General Web Apps	Pros	Cons
Streamlit	Python Web App Library	Very Easy, Low Code [54]	Interactive widgets, real-time updates, strong data visualization [54]	Excellent [54]	Limited for complex, multi-page apps	Rapid prototyping, Python-centric, no web dev knowledge needed [54]	Less flexible for highly customized UIs, not for complex multi-page apps
Flask (with JS Frontend)	Python Micro-framework (Backend)	Moderate	API-driven, flexible backend for custom logic	Good (as API backend)	Excellent (with JS frontend) [60]	Lightweight, flexible, easy database integration, scalable [56]	No built-in UI, requires separate frontend development (JS) [56]
Django (with JS Frontend)	Python Full-stack Framework (Backend)	Moderate to Steep [56]	Built-in admin, ORM, security, form handling [57]	Good (as API backend)	Excellent (with JS frontend) [58]	Robust, secure, rapid MVP development for complex projects, "DRY" principle [56]	Steeper learning curve, less suited for very small projects [56]

PyQt/PySide	Python Desktop GUI Framework	Moderate	Native-looking widgets, extensive Qt features [62]	Good	Good (desktop only)	Cross-platform desktop apps, access to Qt tools (QtCreator) [62]	Desktop-focused, not web-native
Tkinter	Python Desktop GUI Framework	Easy	Basic widgets, part of standard library [62]	Fair	Fair (desktop only)	Standard Python GUI, vast resources, simple to get started [62]	Less native look/feel across OS, limited modern features [62]
Kivy	Python Cross-platform GUI Framework	Moderate	Touch-enabled, custom design language [62]	Fair	Fair (desktop/mobile)	Cross-platform (Linux, Windows, Mac, Android), good for mobile apps [62]	No visual layout program, specific design language

Designing a Simple UI

The user interface should prioritize simplicity and functionality to facilitate efficient interaction for supply chain personnel.

- **Core Elements:** The UI should feature a prominent search bar for entering PO numbers or other identifiers. A clear display area, ideally in a table format, will present the extracted structured data. Crucially, clickable links to the original PDF files must be provided, allowing users to retrieve the source document from its stored path.
- **User Flow:** The typical user interaction would involve:

1. The user inputs a PO number (or other search criteria).
2. The system queries the database for matching PO data.
3. The extracted structured data is displayed in a user-friendly format.
4. A direct link is provided to open the corresponding PDF file.

- **Interactivity:** The UI should allow users to refine their searches, sort results based on various criteria, and potentially provide feedback on the accuracy of the data extraction.
- **Visuals:** The design should be clean and uncluttered, with a strong focus on core functionality. Data visualization libraries, such as Seaborn or Plotly (especially when using Streamlit), can enhance the display of extracted data, making it more digestible and actionable.[54]

The "simple UI" is more than just a display mechanism; it is the critical interface for the human-in-the-loop validation and correction process, especially during the initial stages of system deployment. Incorporating a straightforward feedback mechanism, such as a "Is this extraction correct? Yes/No/Edit" option, directly into the UI can enable continuous improvement of the LLM's performance, particularly if fine-tuning is pursued. As noted in the context of OCR invoice processing, "Exception Handling and Human Review" is a vital step.[1] Furthermore, the ability for users to "Review and confirm AI predictions" and "make adjustments to guide the AI toward better accuracy" [18] highlights the importance of this interactive component. By designing the UI to support human review and feedback, the company can actively "train" its LLM-based extraction system on its specific purchase order documents, leading to compounding accuracy improvements over time and fostering greater trust in the automated process.

Overall System Architecture and In-House Deployment

A well-designed architecture and a robust deployment strategy are paramount for the long-term success, maintainability, and scalability of an in-house software solution.

Modular Design and Containerization with Docker

- **Modular Design:** The solution should be broken down into smaller, independently

deployable services. Examples include a PDF processing service, an LLM extraction service, a data storage service, and a UI service.[63] This modularity enables horizontal scaling of specific components based on demand, allowing for efficient resource allocation.

- **Containerization (Docker):** Docker is a key technology for packaging applications and their dependencies into single, lightweight, executable units called containers.[64]
  - **Benefits:**
    - **Portability:** Containers create executable software packages that are abstracted from the host operating system, ensuring they run uniformly and consistently across any platform or cloud environment. This effectively eliminates the common "it works on my machine" problem.[65]
    - **Environment Consistency:** Docker guarantees that the application's environment remains consistent across development, testing, and production stages, preventing unexpected behavior caused by environmental discrepancies.[66]
    - **Simplified Deployment:** Once an application is containerized, it can be easily moved and executed on any system that supports Docker, significantly streamlining the deployment pipeline.[67]
    - **Isolation:** Each containerized application operates independently, ensuring that the failure of one container does not affect the continued operation of others, thereby enhancing fault isolation.[65]
    - **Efficiency:** Containers share the host operating system kernel, leading to better resource efficiency. They have lower overhead than traditional Virtual Machines (VMs) and faster start times, allowing more applications to run on the same compute capacity.[65]
  - **Implementation:** Dockerfile is used to define the container image, specifying the application's environment and dependencies. docker compose is then used to define and run multi-service Docker applications, orchestrating the interaction between different components.[68]
  - **CI/CD:** Docker is ideally suited for integration into Continuous Integration/Continuous Deployment (CI/CD) pipelines, automating the testing and deployment processes and improving code quality.[64]

Docker is not merely a deployment tool; it is a fundamental enabler for standardizing the development, testing, and deployment environments for complex AI applications. This consistency is paramount for an in-house solution, as it significantly reduces debugging time, facilitates team collaboration, and ensures the reliability of AI models that often have specific and intricate dependency requirements. AI/ML models,

particularly LLMs and their underlying dependencies (such as PyTorch, Transformers for fine-tuning, or even Tesseract for OCR), frequently come with complex and precise dependency specifications. Docker ensures that the exact environment in which a model was developed and tested is faithfully replicated in production, thereby preventing "dependency hell" or the "it works on my machine" conundrum.[67] This consistency is critical for the reliable operation of the AI components. For an in-house solution, Docker streamlines updates, rollbacks, and the scaling of individual components (e.g., updating the LLM without affecting the PDF parser), which significantly reduces operational risk and maintenance overhead, making the custom solution viable in the long term.

Scalability Considerations

Scalability refers to an application's ability to handle an increasing amount of work or its potential to accommodate growth without sacrificing performance.[63] For a supply chain solution, proactive design for scalability is crucial to prevent performance bottlenecks.

- **Horizontal Scalability:** This is the ability to add more nodes or instances of a service to handle increased workload.[63] A microservices architecture, as described above, inherently facilitates this by allowing individual services to be scaled independently based on their specific demand.[63]
- **Concurrency and Parallelism:**
  - **Multiprocessing:** For CPU-bound tasks (those that heavily utilize the processor), using Python's multiprocessing package allows the workload to be spread across multiple CPU cores, effectively utilizing available hardware.[69]
  - **Asynchronous Programming (asyncio):** For I/O-bound tasks (those that involve waiting for external operations like reading from files, database interactions, or LLM API calls), asyncio enables high concurrency using a single thread or event loop.[69] This is particularly beneficial for operations that spend significant time waiting for responses rather than performing computations.
- **Performance Monitoring:** Implementing monitoring tools, such as Prometheus or Grafana, is essential to continuously track key metrics like CPU usage, memory consumption, response times, and error rates.[63] This allows for proactive identification of performance bottlenecks before they impact users.
- **Caching:** Utilizing caching mechanisms, such as Redis or Memcached, for

frequently accessed data can significantly reduce the load on the database and improve overall response times.[63]

Scalability is not merely about handling larger data volumes; it is about designing the system from the ground up to anticipate and prevent performance bottlenecks. For a supply chain, unpredictable spikes in purchase order volume, such as seasonal peaks or end-of-quarter rushes, necessitate a system design that can gracefully scale without manual intervention or system collapse. The choice of asyncio for I/O-bound tasks, like reading PDFs or making LLM API calls, is particularly important here, as these operations frequently involve waiting for external resources.[69] A system that cannot scale horizontally or handle concurrent requests will inevitably become a bottleneck during critical periods. Therefore, proactive design for scalability ensures that the system remains performant and reliable under varying loads, preventing operational disruptions and maximizing the value derived from automated PO processing.

Dependency Management

Effective dependency management is crucial for building and maintaining robust Python projects, especially in collaborative environments and for reliable deployments.[70]

- **Importance:** Proper dependency management ensures consistency across development environments, facilitates collaboration among team members, and enables fearless deployment of the application.[70] Without it, projects can fall into "dependency hell," making updates and future development exceedingly difficult.
- **Tools:**
  - pip and requirements.txt: These are the standard tools for installing Python packages and listing project dependencies.[68]
  - venv (virtual environments): Highly recommended for isolating project dependencies, preventing conflicts between different Python projects on the same machine.[61]
  - poetry and pyproject.toml: For larger, more serious projects, poetry combined with pyproject.toml is increasingly preferred over setup.py for its richer metadata capabilities and more robust dependency resolution and handling.[71]
  - pipx: Useful for installing and running Python applications in isolated environments, particularly for command-line tools.[71]

    - shiv: Allows for the creation of self-contained, isolated Python applications, simplifying distribution.[71]
- **Containerization Role:** Docker plays a significant role in dependency management by packaging the application along with all its specific dependencies into a single container image. This ensures environmental consistency from development to production, abstracting away underlying system configurations.[67]

Poor dependency management is a silent impediment to the success of in-house software projects, frequently leading to "dependency hell" and rendering updates or future development excruciatingly difficult. Proactive use of tools like poetry and Docker prevents this technical debt, ensuring the system remains maintainable and adaptable throughout its lifecycle. The statement that "You have to think about dependencies if you want to build and maintain a serious Python project that you can collaborate on with multiple people and that you can deploy fearlessly" [70] underscores that dependency issues are not mere inconveniences; they can halt development, introduce bugs, and make future upgrades impossible. While Docker encapsulates the environment, good Python packaging practices, such as those facilitated by

poetry, ensure that the *internal* dependency graph of the application is healthy and well-defined. Investing in robust dependency management practices from the outset is thus crucial for the long-term health and cost-effectiveness of an in-house solution, preventing it from becoming a brittle, unmaintainable legacy system.

Conclusion and Recommended Roadmap

This report has detailed a comprehensive open-source architecture for an in-house purchase order data extraction and management system. By strategically combining specialized PDF parsing tools, intelligent LLM-based extraction with robust prompt engineering and Pydantic validation, a scalable PostgreSQL database, and a user-friendly UI, a supply chain company can achieve significant automation and data leverage without relying on costly proprietary services.

Key Synergy of Tools

The strength of this proposed solution lies in the synergistic combination of various open-source tools, each addressing a specific facet of the data processing pipeline:

- **PDF Parsing (e.g., PDFplumber, Camelot, Open Parse, Tesseract):** These tools collectively provide the capability to extract raw text and tabular data from diverse PDF formats, handling both digitally native and scanned documents.
- **LLM Frameworks (e.g., LlamaIndex, ContextGem, LangExtract, Unstract with Ollama):** These frameworks intelligently extract structured PO data from the parsed text, leveraging their semantic understanding capabilities to interpret and normalize varied input.
- **Data Transformation (Pandas, TheFuzz, re):** This suite of libraries is essential for cleaning, normalizing, and reconciling extracted data, ensuring consistency and accuracy before storage.
- **Database (PostgreSQL):** Selected for its robustness and scalability, PostgreSQL provides a highly reliable platform for storing the structured PO data and the corresponding PDF file paths with strong data integrity.
- **Search (Whoosh, Sentence Transformers, OpenSearch Vector Engine):** This combination enables both precise keyword-based search for direct identifiers and advanced semantic retrieval for conceptual queries, ensuring comprehensive document access.
- **UI (Streamlit or Flask/Django + Vue.js/React):** The chosen UI framework provides the necessary interface for user interaction, data display, and crucial human-in-the-loop validation.
- **Deployment (Docker):** Docker ensures consistent, portable, and scalable deployment of all system components, simplifying management and reducing environmental discrepancies.

Phased Implementation Strategy

A phased approach to implementation is recommended to manage complexity and deliver value incrementally:

1. **Phase 1 (Minimum Viable Product - Core Extraction & Storage):**
    - Focus on establishing a reliable PDF text and table extraction pipeline, potentially using PDFplumber for digital documents and Tesseract for OCR on scanned inputs.

    - Implement basic LLM extraction capabilities leveraging a local LLM runner like Ollama integrated with LlamaIndex and Pydantic for structured output.
    - Set up data storage in PostgreSQL with a foundational schema.
    - Develop a Streamlit UI for basic document upload, display of extracted data, and simple keyword search.
    - Crucially, containerize all components using Docker from the outset to ensure environmental consistency and simplify future scaling.
2. **Phase 2 (Refinement & Enhancement):**
    - Improve extraction accuracy through iterative prompt engineering.
    - Evaluate the benefits of fine-tuning a local LLM with LoRA on a small, labeled dataset of actual purchase orders to achieve higher domain-specific accuracy.
    - Enhance the data cleaning process with fuzzy matching techniques (TheFuzz) to reconcile variations in vendor names or product descriptions.
    - Implement semantic search capabilities using vector embeddings (Sentence Transformers) and a vector store (e.g., OpenSearch Vector Engine or local storage) to enable more contextual queries.
3. **Phase 3 (Scalability & Productionization):**
    - Optimize system performance through profiling, caching (e.g., Redis), and leveraging Python's asyncio for I/O-bound tasks and multiprocessing for CPU-bound tasks.
    - Implement robust error handling, comprehensive monitoring (e.g., Prometheus, Grafana), and automated backup and recovery strategies for both the database and the PDF file storage.
    - If the initial "simple UI" becomes a bottleneck or if more complex workflows, user roles, or integrations are required, consider migrating to a more robust Flask/Django backend with a Vue.js or React frontend.

Considerations for Ongoing Maintenance and Future Enhancements

For the long-term success and sustainability of this in-house solution, several ongoing considerations are vital:

- **Continuous Monitoring:** Implement robust monitoring tools to track application health, resource usage (CPU, memory), response times, and error rates. This proactive monitoring allows for the early identification and resolution of potential issues.[63]

- **Feedback Loop:** Establish a systematic process for end-users to provide feedback on the accuracy of data extraction. This feedback is invaluable for iteratively improving LLM prompts, refining fine-tuning datasets, and enhancing the overall system performance.
- **Model Updates:** Regularly evaluate and update the chosen LLMs and embedding models as new, more performant, or more efficient open-source options become available. The field of AI is rapidly evolving, and staying current can yield significant benefits.
- **Security:** Continuously implement secure coding practices, ensure environment isolation for different services, and utilize encrypted communication channels (e.g., HTTPS) for all data in transit.
- **Archiving:** Develop a clear and efficient long-term archiving strategy for older PDF files to manage storage costs effectively while ensuring compliance with data retention policies.

Works cited

1. OCR Invoice Processing: How It Works & Benefits [2025 Guide] - DocuClipper, accessed July 30, 2025, https://www.docuclipper.com/blog/ocr-invoice-processing/
2. PDFDataExtractor: A Tool for Reading Scientific Text and Interpreting Metadata from the Typeset Literature in the Portable Document Format, accessed July 30, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC9049592/
3. Tesseract OCR: Features, Applications, and Limitations - Folio3 AI, accessed July 30, 2025, https://www.folio3.ai/blog/tesseract-ocr/
4. A Guide to Optical Character Recognition (OCR) With Tesseract - Unstract, accessed July 30, 2025, https://unstract.com/blog/guide-to-optical-character-recognition-with-tesseract-ocr/
5. I Tested 7 Python PDF Extractors So You Don't Have To (2025 Edition) | by Aman Kumar, accessed July 30, 2025, https://onlyoneaman.medium.com/i-tested-7-python-pdf-extractors-so-you-dont-have-to-2025-edition-c88013922257
6. Python, Open-Source Libraries for Efficient PDF Management - DZone, accessed July 30, 2025, https://dzone.com/articles/python-open-source-libraries-pdf-management
7. A Comparative Study of PDF Parsing Tools Across Diverse Document Categories - arXiv, accessed July 30, 2025, https://arxiv.org/html/2410.09871v1
8. dzone.com, accessed July 30, 2025, https://dzone.com/articles/python-open-source-libraries-pdf-management#:~:text=PyPDF2,also%20extract%20text%20and%20metadata.
9. Is PyPDF2 the Best Option? Extracting Hyperlinks from PDFs: A Deep Dive - Medium, accessed July 30, 2025,

https://medium.com/@uaqureshi_61924/is-pypdf2-the-best-option-extracting-hyperlinks-from-pdfs-a-deep-dive-026fa9cd2518
10. Filimoa/open-parse: Improved file parsing for LLM's - GitHub, accessed July 30, 2025, https://github.com/Filimoa/open-parse
11. Structured Data Extraction - LlamaIndex, accessed July 30, 2025, https://docs.llamaindex.ai/en/stable/use_cases/extraction/
12. Fine Tune Large Language Model (LLM) on a Custom Dataset with QLoRA | by Suman Das, accessed July 30, 2025, https://dassum.medium.com/fine-tune-large-language-model-llm-on-a-custom-dataset-with-qlora-fb60abdeba07
13. How to Fine Tune LLMs for Your Documents and Data, accessed July 30, 2025, https://blog.bisok.com/general-technology/how-to-fine-tune-llm
14. Prompt engineering - OpenAI API, accessed July 30, 2025, https://platform.openai.com/docs/guides/prompt-engineering
15. Effective Prompt Engineering for Data Extraction with Large Language Models | by DanShw, accessed July 30, 2025, https://medium.com/@kofsitho/effective-prompt-engineering-for-data-extraction-with-large-language-models-331ee454cbae
16. Best practices for prompt engineering with the OpenAI API, accessed July 30, 2025, https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api
17. Structured Outputs in Ollama - What's Your Recipe for Success? - Reddit, accessed July 30, 2025, https://www.reddit.com/r/ollama/comments/1jflnxl/structured_outputs_in_ollama_whats_your_recipe/
18. How to write effective prompts for data extraction in Cradl AI | Apr 07, 2025, accessed July 30, 2025, https://www.cradl.ai/post/how-to-write-great-prompt-for-data-extraction-in-cradl-ai
19. Structured Data Extraction from Unstructured Text Python LLMs Ollama Pydantic Llama 3.2 Granite 3.2 - YouTube, accessed July 30, 2025, https://www.youtube.com/watch?v=dJsTkj9xCKA
20. Output - Pydantic AI, accessed July 30, 2025, https://ai.pydantic.dev/output/
21. A Practical Guide on Structuring LLM Outputs with Pydantic - DEV Community, accessed July 30, 2025, https://dev.to/devasservice/a-practical-guide-on-structuring-llm-outputs-with-pydantic-50b4
22. On-premise structured extraction with LLM using Ollama - HackerNoon, accessed July 30, 2025, https://hackernoon.com/on-premise-structured-extraction-with-llm-using-ollama
23. shcherbak-ai/contextgem: ContextGem: Effortless LLM ... - GitHub, accessed July 30, 2025, https://github.com/shcherbak-ai/contextgem
24. Introducing LangExtract: A Gemini powered information extraction ..., accessed

July 30, 2025, https://developers.googleblog.com/en/introducing-langextract-a-gemini-powered-information-extraction-library/
25. Intelligent Document Processing with No-Code LLM Platform Unstract, accessed July 30, 2025, https://adasci.org/intelligent-document-processing-with-no-code-llm-platform-unstract/
26. Custom Fine-Tuning for Domain-Specific LLMs - MachineLearningMastery.com, accessed July 30, 2025, https://machinelearningmastery.com/custom-fine-tuning-for-domain-specific-llms/
27. FinLoRA: Benchmarking LoRA Methods for Fine-Tuning LLMs on Financial Datasets - GitHub, accessed July 30, 2025, https://github.com/Open-Finance-Lab/FinLoRA
28. Data cleaning in Python (Beginner's guide for 2025) - Apify Blog, accessed July 30, 2025, https://blog.apify.com/data-cleaning-python/
29. Normalizing Textual Data with Python - GeeksforGeeks, accessed July 30, 2025, https://www.geeksforgeeks.org/python/normalizing-textual-data-with-python/
30. Top 26 Python Libraries for Data Science in 2025 | DataCamp, accessed July 30, 2025, https://www.datacamp.com/blog/top-python-libraries-for-data-science
31. Fuzzy String Matching in Python: Introduction to FuzzyWuzzy - Built In, accessed July 30, 2025, https://builtin.com/data-science/fuzzy-matching-python
32. Fuzzy String Matching in Python Tutorial - DataCamp, accessed July 30, 2025, https://www.datacamp.com/tutorial/fuzzy-string-python
33. spaCy 101: Everything you need to know, accessed July 30, 2025, https://spacy.io/usage/spacy-101
34. How to normalise keywords extracted with Named Entity Recognition - Stack Overflow, accessed July 30, 2025, https://stackoverflow.com/questions/75654562/how-to-normalise-keywords-extracted-with-named-entity-recognition
35. How to Make a Search Engine in Python: Step-by-Step Tutorial - Meilisearch, accessed July 30, 2025, https://www.meilisearch.com/blog/python-search-engine
36. Regex : Data Extraction using Python, Pattern Detection for files. Fundamental Overview, accessed July 30, 2025, https://medium.com/@kapoorchinmay231/regex-data-extraction-using-python-pattern-detection-for-files-fundamental-overview-e0f1342ddc9c
37. Pattern matching in Python with Regex - GeeksforGeeks, accessed July 30, 2025, https://www.geeksforgeeks.org/python/pattern-matching-python-regex/
38. SQLite Vs PostgreSQL - Key Differences - Airbyte, accessed July 30, 2025, https://airbyte.com/data-engineering-resources/sqlite-vs-postgresql
39. PostgreSQL vs SQLite: Ultimate Database Showdown - Astera Software, accessed July 30, 2025, https://www.astera.com/knowledge-center/postgresql-vs-sqlite/
40. Complete guide to PostgreSQL: Features, use cases, and tutorial - Instaclustr, accessed July 30, 2025,

https://www.instaclustr.com/education/postgresql/complete-guide-to-postgresql-features-use-cases-and-tutorial/
41. Understanding PostgreSQL Data Integrity - DbVisualizer, accessed July 30, 2025, https://www.dbvis.com/thetable/understanding-postgresql-data-integrity/
42. 4 Database Schema Examples for Various Applications | Airbyte, accessed July 30, 2025, https://airbyte.com/data-engineering-resources/database-schema-examples
43. How to Design Database Inventory Management Systems - GeeksforGeeks, accessed July 30, 2025, https://www.geeksforgeeks.org/dbms/how-to-design-database-inventory-management-systems/
44. How do I store PDF files in a database and retrieve them? - Quora, accessed July 30, 2025, https://www.quora.com/How-do-I-store-PDF-files-in-a-database-and-retrieve-them
45. blob - Store .pdf in SQL or path? - Stack Overflow, accessed July 30, 2025, https://stackoverflow.com/questions/5251101/store-pdf-in-sql-or-path
46. Introduction to Whoosh — Whoosh 2.7.4 documentation, accessed July 30, 2025, https://whoosh.readthedocs.io/en/latest/intro.html
47. mchaput/whoosh: Pure-Python full-text search library - GitHub, accessed July 30, 2025, https://github.com/mchaput/whoosh
48. Whoosh is a fast, featureful full-text indexing and searching library implemented in pure Python. - GitHub, accessed July 30, 2025, https://github.com/whoosh-community/whoosh
49. Quick start — Whoosh 2.7.4 documentation - Read the Docs, accessed July 30, 2025, https://whoosh.readthedocs.io/en/latest/quickstart.html
50. Vector Search - OpenSearch, accessed July 30, 2025, https://opensearch.org/vector-search/
51. A simple semantic search with Python and FastAPI | by Udit Rawat | AgentsOps - Medium, accessed July 30, 2025, https://medium.com/agentsops/a-simple-semantic-search-with-python-and-fastapi-f80f2b785086
52. Vector embeddings - OpenAI API, accessed July 30, 2025, https://platform.openai.com/docs/guides/embeddings
53. Semantic Search with KNN and LLM - Explained with a Poem - DEV Community, accessed July 30, 2025, https://dev.to/mcharytoniuk/semantic-search-with-knn-and-llm-explained-with-a-poem-13gi
54. What is Streamlit: All Why's and How's Answered | UI Bakery Blog, accessed July 30, 2025, https://uibakery.io/blog/what-is-streamlit
55. Building a dashboard in Python using Streamlit, accessed July 30, 2025, https://blog.streamlit.io/crafting-a-dashboard-app-in-python-using-streamlit/
56. Flask vs Django: Let's Choose Your Next Python Framework - Kinsta®, accessed July 30, 2025, https://kinsta.com/blog/flask-vs-django/
57. Django vs Flask: Which Python Framework Should You Choose? - GreenGeeks,

accessed July 30, 2025, https://www.greengeeks.com/blog/django-vs-flask-python-framework/
58. Vue.js - Full Stack Python, accessed July 30, 2025, https://www.fullstackpython.com/vuejs.html
59. A curated list of awesome things related to Vue.js - GitHub, accessed July 30, 2025, https://github.com/vuejs/awesome-vue
60. Python + JavaScript - Full Stack App Tutorial - YouTube, accessed July 30, 2025, https://www.youtube.com/watch?v=PppsIXOR7TA&pp=0gcJCfwAo7VqN5tD
61. This repo is base of how to connect python backend and javascript frontend. - GitHub, accessed July 30, 2025, https://github.com/0PrashantYadav0/Python-Javascript
62. 5 open source Python GUI frameworks | Opensource.com, accessed July 30, 2025, https://opensource.com/resources/python/gui-frameworks
63. Building Scalable Applications with Python: Best Practices for Outsourced Teams - Ellow.io, accessed July 30, 2025, https://ellow.io/building-scalable-applications-with-python-best-practices-for-outsourced-teams/
64. Deploying Python Applications in the Cloud - A Comprehensive Step-by-Step Guide, accessed July 30, 2025, https://moldstud.com/articles/p-deploying-python-applications-in-the-cloud-a-comprehensive-step-by-step-guide
65. What Is Containerization? | IBM, accessed July 30, 2025, https://www.ibm.com/think/topics/containerization
66. What is Containerization? - Harness, accessed July 30, 2025, https://www.harness.io/harness-devops-academy/what-is-containerization
67. How to Deploy Python Using Docker the Easy Way - Inedo Blog, accessed July 30, 2025, https://blog.inedo.com/python/development-and-cicd/
68. Containerize a Python application - Docker Docs, accessed July 30, 2025, https://docs.docker.com/guides/python/containerize/
69. An Introduction to Scaling Distributed Python Applications | by The Educative Team, accessed July 30, 2025, https://learningdaily.dev/an-introduction-to-scaling-distributed-python-applications-7a87da2d868f
70. A complete-ish guide to dependency management in Python - Reddit, accessed July 30, 2025, https://www.reddit.com/r/Python/comments/1gphzn2/a_completeish_guide_to_dependency_management_in/
71. How do you deploy Python applications? - Reddit, accessed July 30, 2025, https://www.reddit.com/r/Python/comments/r6aqji/how_do_you_deploy_python_applications/