ChatGPT

Tech Stack Proposal: Batch Invoice Processing System (2026)

Introduction

This proposal outlines a **Python-based tech stack** for a batch invoice processing system, tailored for development on a MacBook Pro M3 and deployment to the cloud with minimal costs. The system will handle **heterogeneous invoice formats** (CSV, JSON, PDF, scanned images), perform OCR on PDFs/images, extract key fields (invoice number, date, totals), and execute on a schedule via a lightweight orchestrator. We emphasize **high-performance single-machine tools** over heavy distributed frameworks like Spark, to efficiently process a few hundred files per batch without the overhead of a cluster [1]. The architecture is cloud-ready, containerized, and designed for low runtime costs (leveraging pay-per-use infrastructure). Below, we detail the tech stack by layer—**ingestion, processing, OCR/parsing, storage, orchestration, and deployment**—including tool choices, integration points, trade-offs, and future scalability considerations.

Ingestion Layer

The ingestion layer handles reading invoices from various sources and formats, preparing them for processing. We will use simple, robust Python tools to ingest each file type without heavy infrastructure:

- **File Access**: If invoice files reside in cloud storage (e.g. AWS S3 or GCP Storage), use Python SDKs (`boto3` for S3, etc.) or file-system abstractions (like **fsspec**) to stream files. For local or mounted files, standard file I/O is sufficient. This avoids complex ingestion frameworks since hundreds of files are easily listed and read with Python.
- **Structured Data (CSV/JSON)**: Use **Pandas** to read CSVs/JSONs (`pandas.read_csv`, `pandas.read_json`). Pandas efficiently loads structured data into DataFrames for immediate use [2]. This covers invoices already in a structured form (e.g. exported from an ERP). Pandas is mature, easy to use, and handles moderate data volumes well. *(Alternative:* **Polars** DataFrame library can also ingest CSV/JSON with higher performance and lower memory overhead, useful if datasets grow larger [3].)
- **PDF Documents**: Use a PDF parsing library like **PDFPlumber** or **PyMuPDF** (fitz) to ingest PDF files. These libraries extract text from PDFs that contain embedded text. They are lightweight and Pythonic—no need for external services if the PDF is text-based. If a PDF is a scanned image (no text layer), we fall back to the OCR layer (described below).
- **Images (Scanned Invoices)**: Ingest image files (JPEG, PNG, TIFF, or PDF pages rendered to images) using **Pillow** (PIL) or **OpenCV**. These libraries can open images and prepare them for OCR. OpenCV is particularly useful for image pre-processing (rescaling, converting to grayscale, denoising) to improve OCR results [4].

Justification & Trade-offs: This ingestion approach uses simple Python calls (no heavy connectors or distributed ingest needed). It trades off the advanced scaling of tools like Apache Nifi or Spark streaming

for direct control and low overhead. Given the moderate file count, Python's I/O and Pandas are sufficient. Polars could be introduced if performance profiling shows Pandas as a bottleneck, since Polars can utilize multiple cores and handle larger-than-RAM data efficiently on a single machine [3]. Overall, this layer prioritizes readability and simplicity, as these files can be loaded quickly without specialized frameworks.

Processing & Transformation Layer

Once data is ingested, the processing layer performs in-memory transformations and prepares structured outputs. The system will use high-performance DataFrame libraries and Python logic instead of Spark. This yields fast single-machine processing and avoids the startup and memory overhead of a distributed engine [5]. Key components and libraries include:

- **DataFrame Engine**: **Pandas** will be the primary tool for data transformations (cleaning data, normalizing fields, merging datasets). It provides a rich API to manipulate DataFrames in memory and is well-suited for aggregating results from a few hundred invoices. For example, multiple CSV/JSON inputs can be concatenated into one DataFrame and filtered for the required fields. Pandas is widely supported and easy to debug. *(Consideration:* **Polars** can be used as a DataFrame alternative for performance. Polars, built in Rust, executes operations in parallel and can outperform Pandas by an order of magnitude on multi-core machines [3]. It also uses Apache Arrow memory format for efficiency. Using Polars would reduce runtime and memory usage for larger batches, at the cost of introducing a new library. We can start with Pandas for familiarity, and later switch to Polars if processing becomes a bottleneck.)*
- **Structured Transformations**: The system will handle both **in-memory** and **file-based** transformations. In-memory, we use DataFrames to unify schema and compute any derived values (e.g. convert all dates to a standard format, calculate tax if needed, etc.). File-based transformations involve converting or serializing data to formats—e.g. writing a consolidated output CSV or Parquet file. We will leverage **Apache Arrow/Parquet** through libraries like **PyArrow** or Pandas built-in Parquet support to write efficient, compressed output files if needed. Parquet is columnar and compresses data, which saves storage and speeds up later read times. This is a trade-off against simple CSV: Parquet adds a dependency but yields smaller, faster-to-read outputs for larger datasets.
- **Performance Considerations**: By confining processing to a single machine, we eliminate the overhead of cluster setup. A single modern machine (the M3 or a cloud VM) with multi-core can handle moderate data volumes quickly. For example, Polars has been shown to crunch tens of gigabytes of data on one machine in seconds [6] [7]. The cold-start delay and resource complexity of Spark are avoided – **Polars or Pandas jobs can finish in less time than it takes to even provision a Spark cluster** [5]. This design ensures **minimal runtime cost**, as we use only one instance's resources for a short period.
- **Data Validation & Schema**: Python's flexibility allows adding simple validation steps (e.g. ensure totals are numeric, dates parse correctly). We can use Pydantic or simple schema checks as needed. These checks run in-memory and are lightweight, providing assurance of data quality before output.

Justification: This layer's stack (Pandas/Polars + Arrow) is chosen for **speed and cost-efficiency**. It avoids Spark's heavy context (no need for JVM or cluster scheduling on small jobs), aligning with the requirement to avoid unnecessary overhead. Single-machine processing with efficient libraries can postpone the need for any cluster – indeed, Polars case studies show postponing clusters reduces complexity and cost [1]. The trade-off is that extremely large data (beyond a single machine's RAM) would require careful handling (streaming or chunk processing). Given the current scale (hundreds of files, moderate size), this approach is

ideal. Should data scale up, we can introduce chunked processing or a switch to Polars' streaming mode, which can handle datasets even larger than memory on one node [8] (Polars streaming has been used for inputs up to 1 TiB on a single machine). For now, we leverage in-memory transformations for their simplicity and speed, writing out results in efficient formats to bridge to the storage layer.

OCR & Parsing Layer

The OCR (Optical Character Recognition) and parsing layer is crucial for PDF and image invoices. This component extracts textual content from scanned documents and then parses that text to identify key fields. We propose an open-source OCR solution to keep costs low, coupled with Python parsing logic:

- **OCR Engine**: **Tesseract OCR** (via the **Pytesseract** Python library) will perform OCR on scanned PDFs and images. Tesseract is a proven, powerful open-source engine maintained by Google [9]. Using Pytesseract, we can call Tesseract from Python seamlessly [10]. This engine supports multiple languages and can handle typical invoice print fonts. It's well-suited for invoices/receipts, as it can **automate extraction of key details from financial documents** [11]. The choice of Tesseract keeps the solution cost-effective (no OCR API fees) and runs locally within our container. Trade-off: accuracy may not match expensive cloud OCR APIs on tricky layouts, but for moderate-quality scans it's generally reliable. We can improve accuracy by training Tesseract on any custom fonts if needed or adjusting its configuration (e.g. page segmentation modes).
- **Image Pre-processing**: We incorporate **OpenCV** (with **Pillow**) to pre-process images before OCR. This may include converting to grayscale, thresholding (binarization), deskewing, or noise reduction. Pre-processing can significantly boost OCR accuracy by cleaning up the image (for example, removing background noise or increasing contrast on faded text) [4]. OpenCV also allows detecting regions of interest (for example, if we know roughly where totals or dates appear, though with heterogeneous formats this might be less deterministic). These steps are fast on CPU and occur in-memory. (Trade-off: adds complexity to tune these steps; however, we can start with basic steps like grayscale and adaptive threshold which often yield immediate improvements in text recognition.)
- **PDF Text Extraction**: For PDFs that are not scanned (i.e., contain text layer), use **PDFPlumber or PyMuPDF** to extract text directly without OCR. This is faster and error-free compared to OCR for digital PDFs. The system can automatically decide: attempt text extraction, and if the result is empty or non-sensical (indicating a scanned image PDF), then fall back to OCR on that PDF's pages (we can convert PDF pages to images via PyMuPDF or **pdf2image** for OCR input).
- **Key Field Parsing**: Once we have the raw text content of an invoice, we need to extract structured fields (invoice number, date, total, etc.). This will be done via **Python text parsing**:
- We will define regex patterns and heuristics to locate key fields. For example, a regex for invoice number might look for patterns like `Invoice\s*#?:?\s*([\w\-]+)`, dates can be identified with date regex patterns, and totals by looking for currency symbols or keywords like "Total". The parsing logic can be implemented in Python functions that search the OCR text for these patterns.
- To assist this, we can leverage an open-source utility like **invoice2data** (which uses templates and regex rules to extract data [12]). Invoice2data provides a template-based approach: it can use OCR or PDF text and then apply multiple regexes from a template to find fields (e.g., different templates for different vendors) [13] [14]. For a heterogeneous set of invoices, we might not have a template for every layout, but we can start with general patterns and refine over time as we see recurring invoice formats. Using invoice2data's approach or library saves development time by reusing community-contributed patterns for common invoice layouts.

- Another approach is using **Named Entity Recognition or ML models** for field extraction (for example, training a small model to identify "InvoiceDate" in text). However, given the scale and budget, a rule-based approach is initially preferred for simplicity and transparency. It's faster to implement and runs entirely locally. The trade-off is that maintaining regexes for many formats can become cumbersome. For now, we focus on the key fields mentioned and assume they appear with identifiable labels in the text (which is common – e.g., "Invoice No:", "Date:", "Total:").
- **Validation**: After parsing, basic validation ensures the fields make sense (e.g., date is a valid date, total is a number). We can cross-verify totals if line items are present or at least ensure non-empty values. Any failed extraction can be logged for manual review.

Justification: The OCR/parsing layer uses **open-source tools for cost control** and flexibility. Tesseract via Pytesseract is a natural choice as it is *"one of the most popular and powerful open-source OCR tools"* and easily integrates with Python [9] [10]. It is explicitly noted as suitable for **invoices and receipts processing**, which aligns perfectly with our use case [11]. By using Tesseract locally, we avoid per-page costs of cloud OCR services (like AWS Textract or Google Vision), which is crucial for a tight budget. The trade-off is potentially slightly lower accuracy on complex layouts, but with pre-processing and tuning we mitigate that. Moreover, for key fields (which are often in predictable areas of an invoice, like headers or footers), Tesseract is usually sufficient.

For parsing, leveraging Python's text-processing strength keeps the solution simple. We cited invoice2data as it exemplifies our approach: extract text (via PDF or OCR), then apply regex templates to get structured data [12]. This architecture is proven in practice and supports output to common formats like CSV/JSON [15]. We have the flexibility to start with generic patterns and later add specific ones as needed without altering the core pipeline. In sum, this layer's stack (Tesseract + Python parsing) provides a **cost-effective, controllable solution** for data extraction, with the option to integrate more advanced ML or external APIs later if requirements evolve.

Storage & Output Layer

This layer covers how we store both the raw inputs (if needed) and the processed output data, as well as any intermediate artifacts. The goal is to use **cheap, scalable storage** and avoid running heavy databases unless necessary, given the moderate data volume and batch nature:

- **Input Storage**: Raw invoice files can be stored in a cloud object storage service (e.g., **Amazon S3**, **Google Cloud Storage**, or Azure Blob Storage) for durability and easy access by the processing job. Object storage is low-cost and can scale to hold all invoices. Our pipeline can fetch the files at runtime. (If the files are currently just on the MacBook for development, uploading them to a private cloud storage bucket before processing would make the solution cloud-ready.) This decouples storage from compute and allows the processing container to be stateless (it pulls inputs when it runs).
- **In-Memory vs File-Based Processing**: During processing, we aim to keep data in memory to speed up transformations (using Pandas/Polars DataFrames). This avoids unnecessary I/O overhead. However, if intermediate results need to be saved (for checkpointing or memory constraints), we can write them to ephemeral storage. For example, a large combined dataset could be written as a Parquet file on a temporary storage volume and read back in chunks. In practice, given the batch size, we might not need this, but the pipeline supports it.

- **Output Data Storage**: The extracted structured data (invoice fields) will be saved in a format suitable for downstream use and easy querying:
- For simplicity, we can output a **CSV or Excel file** summarizing the invoices in each batch (each row representing one invoice with its key fields). This is human-readable and can be loaded into tools like Excel or imported into databases. CSV is universal, but not efficient for large scale. Given moderate volume, CSV is acceptable here.
- For a more scalable approach, output to **Parquet or JSON** files stored on cloud storage. Parquet is preferable if the output is later processed by analytical tools or SQL engines (e.g., AWS Athena or Google BigQuery can query Parquet in-place). Parquet's efficient encoding means minimal storage cost and faster reads [15]. We can partition output by date (e.g., a folder per processing date) if accumulating many records over time.
- If immediate querying or integration is needed, we could load results into a **lightweight database**. Options include a **SQLite** database file (if a single-user scenario for quick lookups) or a small **PostgreSQL** instance (if multi-user or integration with an app). However, running a database 24/7 adds cost. An alternative with zero runtime cost is to use a serverless query engine: for instance, store data in CSV/Parquet on S3 and use AWS Athena to query it on demand. This avoids having a live DB server. For the initial implementation, we propose using files (CSV/Parquet) on cloud storage as the output, and only move to a database if query needs become complex.
- **Metadata & Logging**: We will also store logs or run metadata (possibly in CloudWatch Logs or Stackdriver if using cloud, or as a simple log file). This helps trace processing history. Additionally, if needed, we can store a copy of the OCR text outputs for each invoice (perhaps in a separate folder, or as part of a JSON output) for debugging extraction issues. This is optional but can assist in verifying OCR results without rerunning the job.

Justification: This storage approach is aligned with **minimal cost and simplicity**. Object storage is extremely cheap (often fractions of a cent per GB) and has no server upkeep. By writing outputs to files in object storage, we avoid the overhead of a continuously running database server, which fits the budget-conscious requirement. Also, using open formats (CSV, Parquet, JSON) ensures the data is not locked in a proprietary system and can be easily consumed by other tools or migrated to a database in the future if needed. The system effectively treats storage in a **pipeline fashion**: raw files in, processed files out. This keeps the architecture stateless and easy to maintain.

The decision to not use a big data lake or Hadoop/Hive is intentional—such tools would be overkill here. Instead, a simple data lake on S3/GCS with Parquet files achieves the goal at negligible cost and complexity. Should the need arise to analyze many invoices over time, those Parquet files can be loaded in bulk by Pandas or queried with Athena/BigQuery on a pay-per-query basis, which is cost-efficient. In summary, the storage layer is **cloud-native and frugal**: pay only for storage space and the tiny I/O costs when reading/writing during the batch job, with no always-on infrastructure.

Orchestration & Scheduling Layer

To run the batch processing on a schedule without manual intervention, we need a lightweight orchestration solution. The orchestrator's role is to **trigger the invoice processing jobs** (e.g., daily or weekly), manage dependencies (if any), and handle retries or notifications on failure. We aim to avoid

heavyweight orchestrators like Airflow due to their overhead in setup and maintenance. Instead, we have two pragmatic approaches:

- **Cloud Scheduling (Serverless Trigger)**: Leverage cloud-managed scheduling to invoke our processing pipeline. For example:
- On AWS, use **Amazon EventBridge (CloudWatch Schedule)** to trigger an AWS Lambda or Fargate Task that runs the processing container on a cron-like schedule (e.g., every night at 2 AM). This approach requires no dedicated orchestrator service – the cloud's native scheduler calls the code directly.
- On GCP, use **Cloud Scheduler** to send an HTTP trigger to a Cloud Run service or Cloud Function that runs the batch. Similarly, Azure has **Logic Apps/Azure Scheduler** for timed triggers.
  Using these services, we achieve orchestration with essentially zero constant cost (schedulers cost pennies or are free for a few jobs) and no servers to manage. The **processing pipeline code (as a container or function)** is invoked only when needed. If the job fails, we can configure retries or alerts via these services as well. This method is extremely lightweight — effectively, it's "cron in the cloud." It meets the requirement of a lightweight orchestrator by offloading scheduling to the provider and keeps our stack simple. The trade-off is that logic beyond scheduling (like complex task dependencies or branching workflows) is hard-coded or limited. For our single pipeline, that's acceptable.
- **Prefect (Python Orchestrator)**: As an alternative (or complementary for more complexity), we can use **Prefect**, a modern lightweight orchestration framework in Python. Prefect allows us to define the workflow in code (as a series of tasks in a flow) and can handle scheduling, retries, and logging. It's far less heavy than Airflow – **Prefect is ideal for agile teams/startups due to its simple Pythonic setup and cost-efficient scaling** [16] . We can run Prefect in two ways:
- **Prefect Cloud**: a hosted option with a generous free tier. We could register our flow and use Prefect Cloud's scheduler and UI. The actual work can run on a small agent (which could be our container or a VM) only when tasks execute. Prefect Cloud handles the orchestration logic for free or low cost, eliminating the need to maintain our own scheduler service.

- **Prefect open-source**: run a Prefect engine locally or in a container that wakes up via Cron. This is somewhat similar to using Cron directly but with Prefect's task management. If using this route, we'd schedule the Prefect flow runner itself (so it's a bit meta).
  Prefect's advantage is better workflow management and observability: we get a dashboard, easier failure handling, etc., which Cron/Cloud Scheduler won't provide out-of-the-box. It also integrates with Python nicely, so we can call our processing functions directly in a Prefect task. The downside is some added complexity and potentially an always-on agent. However, Prefect's design is such that it **minimizes infrastructure overhead** – it can elastically scale and doesn't require a heavy webserver/database like Airflow's MetaDB and scheduler processes [17] . In fact, teams report significant cost reductions switching from Airflow to Prefect [18] . For our budget-minded approach, Prefect could give us orchestration features without the full cost of Airflow.

- **Notifications & Logging**: Regardless of method, we will set up notifications for job outcomes. With cloud scheduler, we can integrate an alert (e.g., AWS SNS or email) if the job fails. With Prefect, we can use its notification hooks or simply have the flow send a message on failure. Logging from the job will go to cloud logs (e.g., CloudWatch Logs, which we can view when needed).

Justification: The orchestrator is intentionally kept **lightweight and inexpensive**. Using a managed scheduler (EventBridge/Cloud Scheduler) is effectively free and extremely reliable, fulfilling the "lightweight

orchestrator" requirement by not introducing new infrastructure at all. It's essentially the minimal solution – just a timed trigger. Prefect is introduced as it aligns with our Python tech stack and is built for simplicity and flexibility, making it a good fit if we want more than basic scheduling. Prefect provides a middle ground with much less operational overhead than Airflow (no need to maintain Airflow's many components, and it scales down when not in use) [16] . We avoid Airflow because it would be like using a sledgehammer for a small nail here: Airflow's scheduler and worker processes would run 24/7, incurring cloud costs and requiring maintenance, which contradicts our minimal runtime cost goal. In contrast, our chosen approach either runs *nothing* when the pipeline isn't running (cloud events) or at most runs a lightweight Python service.

In summary, the orchestration layer ensures **scheduled, autonomous execution** without burdening the team with heavy infrastructure. This keeps the operational complexity low and the costs nearly zero when the pipeline is idle, which is ideal for a batch process that might only run once a day or week.

Deployment & Cloud Architecture

The deployment strategy focuses on containerization and serverless/cloud services to achieve a **low-cost, scalable, and portable system**. Below is the proposed deployment architecture and related considerations:

- **Containerization with Docker**: We will containerize the entire application (ingestion + processing + OCR pipeline) using Docker. Containerization ensures that the environment on the MacBook Pro M3 (Apple Silicon) matches the cloud environment. We will use a Python base image (e.g. `python:3.11-slim`) and install required libraries (Pandas/Polars, Tesseract, etc.). Tesseract OCR will be installed in the image via apt (e.g., `apt-get install -y tesseract-ocr`) or using an existing image like `osgeo/tesseract` as a base. We need to ensure the container is built for linux/amd64 (since most cloud runtimes use x86_64). Docker Buildx can produce multi-arch images from the Mac M3, or we can build on a cloud CI. The container will expose a command (entrypoint) to run the batch processing for a given set of inputs (or to fetch inputs from storage). Having everything in one container means we can easily hand it off to any service that can run a container.
- **Serverless Container Runtime**: Deploy the container on a **serverless platform** to minimize costs:
- **Google Cloud Run** (for example) is a great option: it runs containers on demand, scaling down to zero when not in use. Cloud Run only charges for CPU/Memory during actual execution, with automatic scale-up if needed [19] . This means if our invoice job runs for say 5 minutes a day, we pay for 5 minutes of a small CPU's time per day – which is extremely cost-efficient. Cloud Run also provides a free tier which might cover our usage. It supports container images easily and can be triggered via HTTP (which our scheduler can call).
- Alternatively, **AWS Fargate with ECS** can achieve a similar result. We can define a scheduled task in Amazon ECS to run our container on a schedule (EventBridge triggers an ECS Task). Fargate will spin up the container on a lightweight VM for the duration, then shut down. Billing is per-second of runtime with no idle cost. This is analogous to Cloud Run's model (though Cloud Run is a bit more automated). AWS also has **Lambda** with container image support: since our code might be somewhat heavy (OCR libraries), packaging as a Lambda container (or a smaller Lambda package with layers) is possible. Lambda has a hard 15-minute limit per invocation, but our batch might well finish in that time. If not, Fargate/Cloud Run have no such limit.
- If using Azure, **Azure Container Instances** or **Azure Functions** with a timer trigger could be used similarly. The principle remains: use a managed service to run the container only when needed.
- **Architecture Diagram (Textual)**: *Data Flow*: The overall architecture can be visualized in steps.

- **Schedule Trigger** – e.g., a Cloud Scheduler event kicks off daily at midnight.
- **Container Launch** – The serverless platform (Cloud Run/ECS) spins up the Docker container for the job.
- **Ingestion** – The running container code connects to the input source (reads files from the cloud storage or receives a list of files via the trigger).
- **Processing & OCR Pipeline** – Inside the container, the Python pipeline processes each file: CSV/JSON via Pandas, PDFs via PDFPlumber or OCR (Tesseract) as needed, extracting text and parsing fields.
- **Transformation** – All extracted data is aggregated into a structured in-memory table (DataFrame), where further cleansing or calculations occur.
- **Storage Output** – The container then writes the results (CSV/Parquet) back to cloud storage (e.g., an output bucket or database). If needed, it could also send a completion notification (e.g., post a message or email).
- **Container Shuts Down** – The job finishes and the container is torn down by the serverless platform (scales to zero). No compute resources remain running.

In this flow, each layer we described is encapsulated in the container's runtime. The orchestrator (scheduler) ensures step 1 triggers reliably. The cloud storage acts as the source and sink for data, decoupling the stateless compute from the data. This architecture is inherently scalable and cost-efficient: for instance, if one day we need to run two batches, the scheduler could trigger two containers (or Cloud Run could auto-scale if we set it up for parallelism), and we still pay purely per execution time used.

- **Cost-Saving Measures**: We specifically design deployment to minimize costs:
- **Scale-to-Zero**: Using services that scale to zero ensures we **pay nothing when the pipeline is not running** [19] . This is a huge cost saver for batch jobs.
- **Right-sizing**: We will allocate minimal necessary resources to the container. For example, if our job can run in 1 vCPU and 2GB RAM, we won't allocate more. We can profile memory usage during testing and pick a small size. Smaller instances mean lower cost per minute. We also benefit from short runtimes (thanks to efficient processing libraries, the job might only take a few minutes). Shorter runtime = lower bill.
- **Spot Instances (optional)**: If using a VM or ECS, and if the schedule is flexible, we could use spot instances for even cheaper compute. But with serverless (Cloud Run/Lambda), this is not applicable (they already operate at low cost for small jobs).
- **Reuse Free Tiers**: Many cloud providers have free tier offerings (e.g., Cloud Run gives some free CPU seconds, Lambda has free requests per month, S3 has free tier storage up to some GB). By staying within these limits, the running cost might be virtually $0. We will attempt to design within free allowances if possible (for moderate use, this is realistic).
- **Monitoring and Auto-shutdown**: Ensure logs are monitored so if something goes wrong and a container runs longer than expected, we catch it. Also set timeouts – e.g., Cloud Run can have a max timeout to auto-stop a runaway job. This prevents unexpected cost.
- **Container Build Cache & CI**: Use CI/CD to build and deploy containers efficiently. This doesn't directly affect runtime cost but ensures quick iteration. We note that building for ARM (Mac M3) vs AMD (cloud) might need some care – using buildx or a cloud build service will resolve any architecture differences.

Justification: The deployment approach leverages modern cloud practices to keep operations lean. By containerizing, we achieve **portability** – the same Docker image runs on any cloud. By using a serverless container service, we align with a **pay-per-use model**, which is explicitly beneficial: *"Cloud Run automatically scales down to zero when no requests, ensuring cost-efficiency; you only pay for CPU/memory while code is*

running" [19] . This directly addresses the tight budget requirement. We avoid having an EC2 VM running 24/7 waiting for cron – instead, the platform itself handles the cold start and termination. There is a trade-off that each job will have a cold start delay (a few seconds to start the container), but given our batch nature, this is negligible.

Using Docker also means local development on the Mac can be done in the same environment (via Docker Desktop) to iron out issues. This eliminates "it works on my machine" problems when deploying. Additionally, containerization encapsulates dependencies like the Tesseract binary, which simplifies installing that on cloud (no need to manually configure the environment each time; the container has it).

Finally, this architecture is inherently ready to **scale out** if needed: we could run multiple containers in parallel for multiple batches or increase resources for a bigger job without redesigning the system. The cloud handles the scaling – we remain focused on our code. All these points make the deployment and ops **low-maintenance and cost-effective**, which is exactly what we want.

Future Scalability and Enhancements

While the proposed stack is optimized for the current scope (hundreds of invoices per batch, moderate data), it's important to consider how we can scale or extend the system if volumes grow or requirements change. Here are options and pathways for scaling in the future:

- **Scaling Volume (Horizontal)**: If the number of invoices increases significantly (say to thousands or more per batch), we can scale horizontally by processing in parallel:
- The stateless design allows deploying multiple container instances concurrently. For example, Cloud Run or ECS could spin up N containers to process chunks of the invoice list in parallel. We could partition the input (e.g., split the list of files into N parts) and have each instance run the same code on its portion. This would linearly reduce total processing time. The orchestrator can coordinate this (e.g., Prefect can spawn parallel tasks, or we trigger multiple events in EventBridge). This way, even if volume increases 10x, we can keep wall-clock time low by adding some parallelism, while cost scales roughly with usage (which is expected).
- Within a single container, we can also employ **multithreading or multiprocessing** for certain steps. For instance, OCR on images can be parallelized using Python's `concurrent.futures` to OCR multiple files at once (Tesseract itself runs on one core per image, but we can run separate images on separate cores). On a multi-core VM, this can utilize available CPU better. Libraries like **Dask** or **Ray** could be introduced to distribute the DataFrame operations or file processing across cores or machines in a more structured way if needed. For example, Dask can create a cluster of workers (even on one machine or many) to parallelize Pandas operations while keeping a similar API. This might be considered if Pandas/Polars on one core become a bottleneck. The trade-off is added complexity, so we'd only jump to this when necessary.
- **Scaling Data Size (Vertical)**: If each invoice file becomes larger (e.g., very long PDF invoices or more data per invoice), we can allocate more resources to the container (more memory, CPU). Polars in streaming mode can handle very large datasets by processing data in chunks rather than loading all into RAM [8] . Thus, even if data per batch grows from megabytes to many gigabytes, we could still manage on a single powerful machine by using streaming and chunk processing (or using cloud instances with high memory for the duration of the job). We may also break the batch into smaller sub-batches to avoid any memory pressure, processing sequentially if needed (still within the same triggered run, just dividing work).

- **Advanced OCR/ML**: If accuracy or complexity needs increase (for example, invoices in many layouts or low-quality scans), we might integrate more advanced document processing:
- Use of **Machine Learning models for key field extraction**: e.g., transformer-based models (like LayoutLM or Google's Document AI) that understand invoice layout could be employed. There are open-source projects and libraries for document layout analysis (like **LayoutParser** or **detectron2** for text region detection). These would improve extraction robustness but require adding ML inference (which might need a GPU for speed, or at least more CPU). This is a future path if our regex-based approach struggles.
- **Cloud OCR services**: If we reach a point where accuracy trumps cost, we could selectively use services like **Amazon Textract** or **Google Cloud Vision API**. These services can directly extract structured fields from invoices (Textract has an AnalyzeExpense API for invoices that returns line items and fields). They scale automatically for large volumes. The downside is cost per page, so we might only opt in if volume is high and internal OCR is failing too often. It could also be a hybrid approach: maybe use Tesseract for most and fall back to Textract for those that didn't parse correctly, to minimize usage.
- **Storage/Data Pipeline Evolution**: With growth, storing results in a proper database could become beneficial. For instance, if we accumulate millions of invoice records over time and need to query them frequently, loading from flat files is not ideal. We could set up a **cloud data warehouse** (BigQuery, Snowflake, or AWS Redshift) or a relational DB to load the processed results incrementally. Since our output is already structured, this would be straightforward. We could even automate the load (e.g., an AWS Lambda that inserts CSV output into a RDS database). Using a query-optimized system would allow building dashboards or integrations on the invoice data. Until that scale, the file-based output suffices.
- **Workflow Orchestration at Scale**: If more pipelines or complexity arises (for example, after processing invoices we have additional steps like reconciliation or posting to an accounting system), we might introduce a more sophisticated orchestrator like **Apache Airflow or Dagster**. This would make sense if we have multiple dependent tasks, complex schedules, or a need for an UI to manage dozens of pipelines. Airflow can handle enterprise-scale workflows and has a large ecosystem, but it introduces higher overhead (maintenance and cost) as discussed. Dagster focuses on data asset lineage, which might be useful if our pipeline grows into a larger data platform. However, unless scaling necessitates it, we'd likely continue with Prefect or cloud schedules for simplicity. Prefect itself can scale with our needs (it can orchestrate hundreds of tasks and has an agent-based scaling model). Notably, Prefect's elasticity would help us avoid idle infrastructure even as we scale (it can dynamically start flows on infrastructure only when needed) [16].
- **Cost Management**: As usage grows, we will continuously monitor costs. Scaling up might prompt exploring committed use discounts or more efficient resource use:
- If the job runs longer or more often, evaluate if a small always-on instance is cheaper than serverless (though with hundreds of files, likely serverless remains cheaper until extremely frequent runs).
- Use of spot instances or reserved instances for any long-running pieces.
- Optimize the code to cut down runtime (e.g., if OCR is the slowest part, possibly use a faster OCR engine or distribute it as mentioned).
- Cloud cost monitoring tools or alerts to catch any unexpected surge (for example, if someone accidentally triggers many runs).

In essence, the system is designed to **gracefully grow**: thanks to containerization and stateless processing, we can scale out by adding parallel containers or scale up by using bigger machines without architectural changes. The single-machine approach with efficient libraries is surprisingly scalable – as evidenced by Polars handling tens of gigabytes with ease on one node and even terabyte-scale with streaming [8]. This

means we likely can defer a major architectural overhaul for a long time. If we ever cross that threshold, introducing a distributed compute engine (Spark or Flink) would be a consideration, but by then the volume would likely justify the overhead.

For now and the foreseeable future, this stack provides a robust foundation that balances performance and cost. Each component can be swapped or scaled as needed: e.g., swap Pandas with Polars for more speed, swap Tesseract with an ML model for more accuracy, upgrade the orchestrator when pipelines multiply, etc. This modularity ensures we are not locked in and can adapt the tech stack in a targeted way to meet new challenges.

Conclusion

This technical stack proposal offers a **clear, actionable plan** for building the batch invoice processing system. By using Python-centric and single-machine optimized tools at each layer, we satisfy the project requirements with minimal bloat: ingesting multiple formats easily, using OCR to extract text, parsing key fields, and performing transformations all within a lightweight pipeline. We have justified each tool choice (and noted alternatives) with an emphasis on keeping the solution lean and cost-effective. The architecture described (and diagrammed in text) shows how data flows from raw invoices to structured outputs under the governance of a simple scheduler. We also provided practical deployment advice (containerization and serverless execution) to minimize cloud runtime costs, aligning with the tight budget. Importantly, this proposal is future-proofed with options to scale and evolve the system as needs grow – without paying the price of premature over-engineering today.

By implementing this stack, the user can develop on a MacBook Pro M3 with confidence that the same code will run efficiently in the cloud, processing invoice batches on schedule, and only incurring cloud charges when it's actually doing useful work. This fulfills the core objective: a **cloud-ready, budget-conscious invoice processing pipeline** that is both **high-performance** (thanks to optimized single-machine tools) and **maintainable** (thanks to straightforward Python code and simple infrastructure). The result will be an automated system turning diverse invoice documents into consistent, structured data with minimal manual effort or expense.

---

1 5 8 Polars — Polars at Decathlon: Ready to Play?
https://pola.rs/posts/case-decathlon/

2 4 How to Extract Data from Invoices Using Python: A Breakdown
https://nanonets.com/blog/how-to-extract-data-from-invoices-using-python/

3 Polars vs. pandas: What's the Difference? | The PyCharm Blog
https://blog.jetbrains.com/pycharm/2024/07/polars-vs-pandas/

6 7 Spark vs Polars. Real-life Test Case. - by Daniel Beach
https://dataengineeringcentral.substack.com/p/spark-vs-polars-real-life-test-case

9 10 11 Open-Source OCR With Tesseract: A Practical 2026 Guide
https://unstract.com/blog/guide-to-optical-character-recognition-with-tesseract-ocr/

12 13 14 15 invoice2data · PyPI
https://pypi.org/project/invoice2data/

16 17 Airflow vs Prefect vs Dagster: Choosing Your Data Orchestrator | Talentblocks Blog
https://talentblocks.com/blog/airflow-vs-prefect-vs-dagster-choosing-the-right-data-orchestrator-for-your-project

18 Modern Orchestration: Endpoint's evolution from Airflow to Prefect
https://www.prefect.io/blog/modern-orchestration-endpoint-s-evolution-from-airflow-to-prefect

19 How to Minimize Costs When Deploying a Full-Stack Application to Google Cloud Run: 9 Cost-Effective Tips | by Fahima Mokhtari | Medium
https://df-mokhtari.medium.com/how-to-minimize-costs-when-deplyoing-a-full-stack-application-to-google-cloud-run-9-effective-tips-46f06bb433fb