The American Journal of Engineering and Technology
ISSN 2689-0984 | Open Access

TYPE Original Research
PAGE NO. 215-221
DOI 10.37547/tajet/Volume07Issue09-17

Overcoming the Real-World Pitfalls of Google Document AI

Oleksandr Tserkovnyi
TrialBase Inc., Principal Engineer Dominican Republic, Punta Cana

OPEN ACCESS

SUBMITTED 08 August 2025
ACCEPTED 14 August 2025
PUBLISHED 30 September 2025
VOLUME Vol.07 Issue 09 2025

CITATION
Oleksandr Tserkovnyi. (2025). Overcoming the Real-World Pitfalls of Google Document AI. The American Journal of Engineering and Technology, 7(09), 215–221. https://doi.org/10.37547/tajet/Volume07Issue09-17

Abstract: This paper discusses the practical and feature gaps that were encountered with Google Document AI in building the AI product at TrialBase platform (ai.trialbase.com), which automates legal document analysis. Results matter because there is an explosion of electronic legal documents that require fast and reliable parsing, which is essential for systems based on LLMs and retrieval-augmented generation. Standard Document AIs seldom work well in practice, even if there are no damaged PDFs, and if a large dataset is being used, wherein the API quota is not hit, and processing costs do not matter. The architecture proposed in this paper is robust, efficient at transforming various documents into structured data. Event-driven microservice architecture with message queues and a PDF sanitization pipeline solves real-world problems by enabling ProcessorPool (multiple processors using synchronous Document AI API to go beyond quota limitation concurrently drastically reducing processing times). Pre-sanitization, coupled with asynchronous batch processing and a custom load balancer, got a tenfold speed increase with enhanced reliability over real-world legal documents. The article is meant to help LegalTech researchers and practitioners, workflow developers, and engineers working on high-performance, reliable Google Cloud-based projects.

Keywords: Google Document AI, LegalTech, document analysis automation, PDF processing, RAG, LLM, microservice architecture, asynchronous processing, ProcessorPool.

Introduction

In current LegalTech practice, document analysis automation is a core determinant of efficiency and cost

reduction. Practically, platforms that are meant to support the legal process need to deal with large volumes of multi-type documents: police reports, medical bills, and deposition transcripts, all other materials related to the case. The primary task is to convert scanned pictures packed into PDFs into neat text that can be viewed later using large language models and a technique called retrieval-augmented generation (RAG), which enables attorneys to reliably apply large language models to practical tasks, such as the automated preparation of demand letters.

It is founded on the creation of TrialBase, a product designed to perform automatic deposition management across the entire lifecycle. A key element of this system includes an AI-related component whose functionalities shall review all case files and detect cross-references among them, as well as help prepare outputs of a legal nature, such as demand letters. It should be noted that the commercial product TrialBase (trialbase.com) is primarily focused on automating and managing the deposition process, including document package preparation, participant coordination, and the generation of legal artifacts during the event. By contrast, the AI module (ai.trialbase.com) addresses ancillary pre- and post-deposition stages, such as intake forms, demand-letter drafting, and broader pretrial preparation, providing structured inputs to improve downstream LLM-driven workflows. It is exactly at the 'analyze all case documents' stage that most technical challenges regarding data extraction come into play. Google Document AI was selected as the major tool towards this objective because of the motivation of project infrastructure being fully integrated into the Google Cloud Platform (GCP) ecosystem, so as to reduce using third-party providers and for easy maintenance. The article sequentially lists down the evolution steps of the document processing approach shares where standard implementations go short comes and at last shares optimized architectural solution.

Materials and Methodology

The study draws its learning from the practical experience gained in designing and running TrialBase. The research materials included a corpus of legal documents: police reports, medical bills, deposition transcripts, and related procedural files. Most came as PDFs, which mostly serve as containers for scanned images, thus requiring optical character recognition (OCR) technologies.

The method used a multi-level approach that included actual testing, making the structure, and comparing Document AI processors. First, it tested three main types of processors: Document OCR, which identifies and extracts text in different types of documents, Layout Parser, which identifies and extracts document layouts and chunks (Xu et al., 2021), and Form Parser, which extracts form elements such as text and checkboxes (Powalski et al., 2021). The purpose was to discover one answer for many legal issues. It also tested an Invoice Parser, revealing its flaws with both tables and words (Appalaraju et al., 2021), as well as a Utility Parser, which extracts more than 30 fields from Utility statements: amount, line items, etc., and an Expense Parser, which extracts from Receipts, including supplier, total amount, tip, etc.. A step occurred before testing, which involved copying the actual environment. The first idea was straightforward: placing documents in the GCP cloud store, utilizing Document AI with a simple API call, and then enhancing the knowledge base for the RAG setup. Tests on files of different sizes brought out a limitation of 15 pages for Form Parser as well as quotas of 6 requests per minute per processor being used. To verify these constraints, a performance analysis was done whose results are captured in tables and measurement logs.

The architecture for asynchronous processing was based on batch processing via LRO. Documents got chopped up to the max limits of DocAI (100 pages, 1GB per chunk). For even more reliability, add yet another pre-sanitization pipeline for files with qpdf, Ghostscript, and pdf-cpu. This will standardize and repair corrupted PDFs, hence minimizing errors on the Document AI side.

The last stage constituted the design and testing of a personalized pool of processors (ProcessorPool) that would allow for parallel calling of the same synchronous Document AI API thus bypassing the low performance realized from batchProcessDocuments natively. ProcessorPool distributed page processing across numerous pre-created Form Processor instances, while respecting quotas and gRPC-channel constraints. This enabled comparison of native and custom approaches and a quantitative estimate of the acceleration achieved (up to a tenfold reduction in time).

Results and Discussion

The system's central concept is that users upload a collection of documents that the system must analyze to build a RAG knowledge base. The majority of files are

PDFs, which are essentially containers of scanned images of pages. This disables any simple text extraction and hence necessitates optical character recognition (He & Schomaker, 2017). Google Document AI is making a full implementation available at the low end, abstracting from the developer all low-level, fiddly details. This includes content-type detection (text versus image content), segment dispatching to OCR, and then processing and merging results.

The service provides various specialized "processors." For this study, three core types were considered: OCR Processor, an enhanced version of standard OCR augmented with Google models and supporting image formats and PDF; Layout Parser Processor, intended for documents with clear structure, such as DOCX or XLSX, extracting layout elements (text, tables, lists) (Xu et al., 2021); and Form Parser, the primary processor chosen for implementation, optimized for documents containing forms (medical bills, questionnaires, reports), yet also demonstrating high effectiveness on ordinary scanned letters and textual documents (Powalski et al., 2021).

Empirical tests have proven that more specialized processors, namely the Invoice Parser, Utility Parser, and Expense Parser, do not supersede the Form Parser in terms of universality and accuracy even on domain-specific documents. For example, it may completely miss table columns or line breaks (Appalaraju et al., 2021). The Form Parser yielded more completeness and accuracy of extraction because textual blocks, for example, physicians' summaries, are never omitted, and also recognizes table structure correctly. Hence, the Form Parser was selected as a universal instrument for most incoming documents.

The initial working hypothesis envisioned a linear, straightforward workflow. The client application would upload documents to GCP cloud storage via a pre-signed URL and then dispatch a request to the server to process them. The server, in turn, would invoke Document AI, await the result, enrich the RAG system with the output (create vectors and persist them in the database), and notify the client upon completion.

The initial implementation followed this plan and consisted of two API endpoints: one to generate the upload link and the other to initiate processing. It worked well with small files (1-10 pages), but when a 22-page document was tried, it hit against very rigid limitations of the synchronous Document AI API: File size must not exceed 40 MB, and page count, as shown in Table 1, is capped at 15.

Table 1. Performance and Constraints of the Form Parser (compiled by author)

Pages per document	15 pages	Form Parser processor limits	You can raise this to 30 pages by setting imageless_mode = true in the ProcessRequest payload.
Image resolution	40 megapixels per page	Content limits	Applies only to image files; PDFs are exempt.
Requests per minute	6 req/min	"Online process requests per minute (single region)" quota	This bucket is per project × processor type. All Form Parsers you create in the same region share the same 6-RPM allowance.

Global API ceiling	1,800 req/min *per user*	General Document AI quota	Rarely is the bottleneck, but it applies across all Document AI methods you call from that user in the project.

As a temporary measure, a utility was implemented to split documents into chunks, process them sequentially, and then merge the results

Nevertheless, this approach proved untenable for large files typical of medical documentation (e.g., 700 MB and 1,600 pages). First, the long wait times (over 30 minutes) triggered timeouts in the browser or network infrastructure. Second, the user experience (UX) was severely degraded, as users had to remain on the page, awaiting completion, which increased the risk of churn. Third, processing errors sporadically arose due to PDF integrity issues.

To address long waits, the system migrated to the asynchronous batch processing provided by Google Document AI. This mechanism operates similarly to the synchronous path but enforces different limits and uses a distinct API grounded in long-running operations (LRO). The architecture was modified: after file upload, the client initiated processing and subsequently polled the server for status at intervals, freeing users from having to wait on an active tab. The server started a batch job in Document AI. When checking on its progress, it looked at the LRO state.

Batch-processing limits go much higher: up to 1 GB per chunk and up to 100 pages. The system was enhanced to split large documents into manageable chunks, adhering to these limits. This solution not only successfully handled large files but also improved user experience. However, real-world data stress tests revealed new issues, both in terms of instability with specific files and general performance, which motivated an architectural redesign at a more fundamental level.

It has been observed that a large percentage of PDFs uploaded by users are damaged or not standard-compliant, most likely due to multiple conversions and the use of low-quality tools for scanning/merging. In such cases, Document AI throws back an error of file corruption or invalid format. It has also been empirically noticed that though the 100-page limit is not crossed more than 50 pages in a chunk mostly fails, the same document split into 20-page chunks succeeds.

The solution, therefore, was to set up a pre-processing and PDF “sanitization” pipeline before dispatching data to Document AI. The open-source command-line toolchain used included qpdf, Ghostscript (gs), pdf-cpu, mutool, and pdfseparate. Among the features that those tools offer are validation and repair, check file structure and fix errors in it; optimization and compression, make the file smaller without losing any quality; PDF version updates, take legacy documents up to what has become a modern standard, so there’s improved compatibility; decryption, removal of protection from encrypted documents.

A multi-tier fallback mechanism was developed, whereby the system attempts to process the file using different utilities at various tiers, as shown in Figure 1.

```
export async function repairPdf(inputPath, tmpDir) {
  const repairTempDir = join(tmpDir, `repair-${Date.now()}`);
  // Create the repairTempDir if it doesn't exist
  await fs.mkdir(repairTempDir, { recursive: true });
  // 0. Try to upgrade PDF version first using qpdf
  const upgraded = join(repairTempDir, `upgraded-${Date.now()}.pdf`);
  try {
    // First pass: try to fix and upgrade the PDF
    await run("qpdf", [
      "--linearize",
      "--object-streams=generate",
      "--compress-streams=y",
      "--qdf",
      "--force-version=1.7",  // Force a newer version
      "--decrypt",            // Remove encryption if present
      inputPath,
      upgraded
    ], 30000, 'upgrade pdf');
    console.debug(`[REPAIR] PDF ${inputPath} upgraded to newer version using qpdf`);
    inputPath = upgraded;
  } catch (e) {
    console.warn(`[REPAIR] Failed to upgrade PDF using qpdf ${inputPath}`, e);
  }

  // 1. pdfcpu validation
  try {
    await run("pdfcpu", ["validate", "-quiet", inputPath], 30000, 'repair pdf');
    console.debug(`[REPAIR] PDF ${inputPath} is valid using pdfcpu`);
    return inputPath;
  } catch { /* fall through */ }

  // 2. mutool clean
  const cleaned = join(repairTempDir, `clean-${Date.now()}.pdf`);
  try {
    await run("mutool", ["clean", inputPath, cleaned]);
    console.debug(`[REPAIR] PDF ${inputPath} is valid using mutool`);
    return cleaned;
  } catch (e) {
    console.warn(`Unrepairable PDF ${inputPath}`, e);
  }

  // 3. give up — caller will rasterise page-by-page
  console.debug(`[REPAIR] PDF ${inputPath} is unrepairable`);
  throw new Error("Unrepairable PDF");
}
```

Fig. 1. Repair PDF code sample (compiled by author)

This stage ensured that only valid, decrypted, and standardized files were sent to Document AI, which significantly increased the overall reliability of the system. Previously, it was highlighted that there was a basic flaw. The client application still had to be responsible for some part of the logic (status polling). Without queues, load management could not be properly handled when many users were uploading files at once. This necessitated a fully event-driven architecture.

The new architecture cleanly separates responsibilities among system components. The client (front end) merely requests a pre-signed URL and uploads the file to cloud storage; its role then concludes. The messaging layer (Pub/Sub) automatically triggers an event after a successful upload, such that it gets published to a topic. The Upload Handler (which is a microservice listening to this topic) will create a document record inside the database (storing the file path, user id, and case id), set some initial status, and then dispatch a task into the next service's queue. The Document Worker prepares and processes files inside Document AI. PDF sanitization, chunking, and recognition are performed, then update the status inside the database and send the extracted text to the next queue. The LLM Worker gets already processed text from a queue split in accordance with token limits. It gets classified by any fast model for example flash-2.0 synopsis generation with a primary model vector embeddings built using the text-embedding-004 model Genkit AI then stored inside PostgreSQL with pgvector extension for further usage inside the RAG system.

This setup makes sure that elements are loosely joined, making it easy to work on and keep up with each part. The plan with lines (Cloud Tasks) and Pub/Sub provides a robust system that can handle the most load, as it brings additional service units in Cloud Run when needed. Even after setting the building right, the speed problem stayed. The native async batch API from Google turned out to be shockingly slow. Processing a large document (1,600 pages, 700 MB) took about 50 minutes, and this was sometimes only to obtain an error response about the source file. Meanwhile, processing a 100-page document via a custom implementation with sequential synchronous API calls took 1–2 minutes. Extrapolating this result (16 × 2 minutes + 20% margin) yielded an estimate of 39 minutes, already faster than the native solution.

This led to the hypothesis that a faster batch-processing system could be built by parallelizing the fast synchronous API calls (Li et al., 2022). To that end, a key component, ProcessorPool, was developed. Essentially a custom load balancer, it manages a pool of several (e.g., 20) pre-created Form Processor instances in Document AI.

The ProcessorPool's operating principle, shown in Figure 2, is that the pool maintains a list of available processor IDs and tracks the current load on each, honoring the synchronous API constraint of no more than six requests per minute per processor. When handling a large document, it is first split into individual pages. The pool then distributes page-processing tasks to the least-loaded processors, ensuring that limits are not exceeded. Execution proceeds in parallel using pLimit: the main thread does not block waiting for each page but awaits completion of all promises at the top level. A retry policy is applied to pages whose processing failed. The system operates a shared pool for all documents uploaded concurrently and processes them on a FIFO basis.

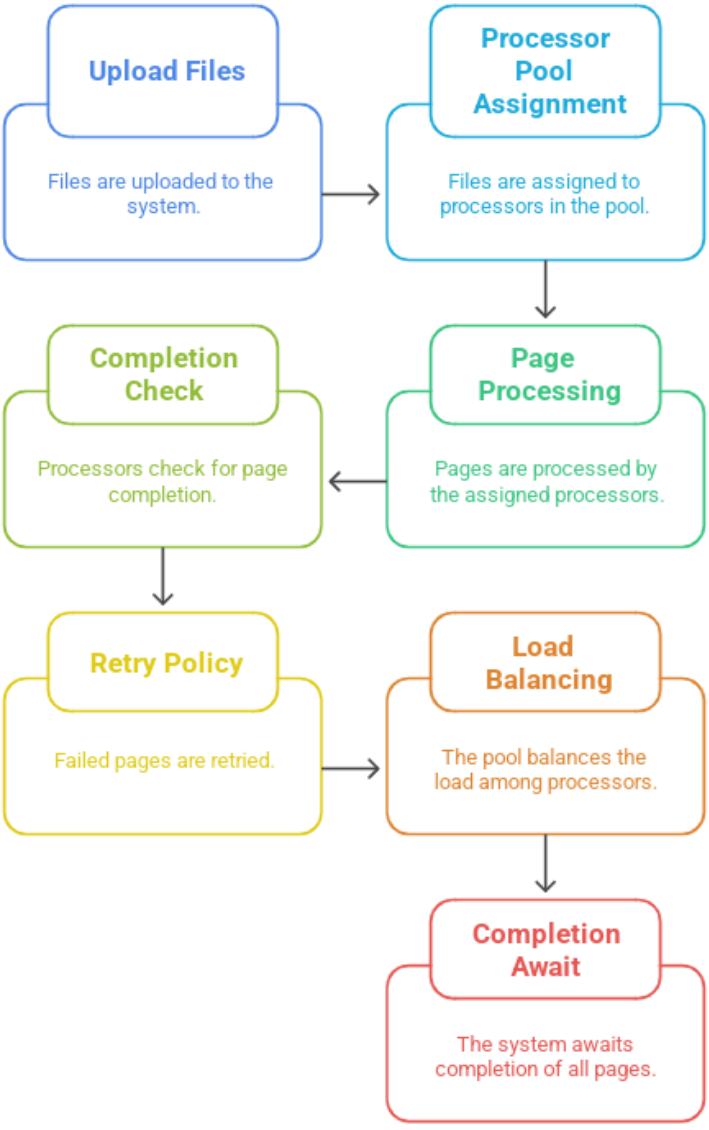

Fig. 2. ProcessorPool Sequence (compiled by author)

A critical technical nuance involved the payload-size limit of the gRPC channel used by the Document AI SDK. Despite the API's declared 40 MB limit, the gRPC channel cannot transmit more than 4 MB. Therefore, pages exceeding ~3.5 MB (base64) are uploaded to temporary storage before processing, and a link is passed to Document AI.

Introducing the ProcessorPool enabled a tenfold acceleration on some documents. For example, the end-to-end processing time for a 1,000-page document (including sanitization, recognition, and RAG creation) fell to 12 minutes. Moreover, this approach provided granular control, enabling the precise identification and debugging of issues at the page level.

Conclusion

The normal ways to use Google Document AI that the documentation suggests are not good for making a system that works well and is reliable when dealing with actual legal papers. Problems in practice- faulty PDFs and poor results from the native batch API caused the need for a complete multi-stage fix.

The design produced a microservices architecture, message queues, and a presanitization pipeline that could deliver the level of fault tolerance and scalability aimed for. ProcessorPool can do synchronous API calls in parallel, which beats the so-named state-of-the-art asynchronous method today by up to 10x in processing time. What this means is essentially instant, reliable conversion-at-scale from arbitrary volumes of heterogeneous documentation into structured data ready for analysis by LLMs. Future work will involve training and tuning Document AI processors to achieve better recognition accuracy and efficiency for specific document types.

Quotas and limits related to Google Document AI work. In synchronous (online) processing, major quotas include a 15-page document limit and a 6 requests per minute per processor type in a region. For batch processing, the limits are higher: up to 5 concurrent requests per processor, up to 10,000 pages in active processing per project, and up to 100 pages per document for the Form Parser.

Cost is a significant factor. The Form Parser rate is $30 per 1,000 processed pages. Cost analysis during testing revealed that processing a standard multi-page medical document can cost up to $45. This aspect requires careful consideration when shaping the product's economic model.

Beyond API limitations, other technical challenges were uncovered. The constrained resources of Cloud Run (maximum 8 vCPU and 32 GB RAM) demand careful balancing of parallel processes to avoid out-of-memory errors; an optimal configuration for a single instance proved to be 25 parallel operations. Service timeouts also matter: Cloud Tasks is limited to 1,800 seconds (30 minutes), so jobs that may run longer are best moved to Cloud Run Jobs, where the timeout extends to 7 days. Additionally, sporadic connectivity issues were observed between one service and the Cloud PostgreSQL database, which were resolved by fine-tuning the connection parameters.

References

1. Appalaraju, S., Jasani, B., Kota, B. U., Xie, Y., & Manmatha, R. (2021). DocFormer: End-to-End Transformer for Document Understanding. *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. https://doi.org/10.1109/iccv48922.2021.00103

2. He, S., & Schomaker, L. (2017). Beyond OCR: Multi-faceted understanding of handwritten document characteristics. *Pattern Recognition*, *63*, 321–333. https://doi.org/10.1016/j.patcog.2016.09.017

3. Li, Z., Guo, L., Cheng, J., Chen, Q., He, B., & Guo, M. (2022). The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Computing Surveys, 54(10s),* 1-34. https://doi.org/10.1145/3508360

4. Powalski, R., Borchmann, Ł., Jurkiewicz, D., Dwojak, T., Pietruszka, M., & Pałka, G. (2021). Going Full-TILT Boogie on Document Understanding with Text-Image-Layout Transformer. *Arxiv*. https://doi.org/10.48550/arxiv.2102.09550

5. Xu, Y., Xu, Y., Lv, T., Cui, L., Wei, F., Wang, G., Lu, Y., Florencio, D., Zhang, C., Che, W., Zhang, M., & Zhou, L. (2021). LayoutLMv2: Multi-modal Pre-training for Visually-rich Document Understanding. *In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 2579–2591. https://doi.org/10.18653/v1/2021.acl-long.201