ChatGPT

Executive Summary

In the quest to extract structured data from Russian товарно-транспортные накладные (ТТН), there are two broad technological approaches: **traditional OCR pipelines** and **modern AI/LLM-driven solutions**. Traditional OCR engines (e.g. ABBYY, PaddleOCR) focus on text detection and layout analysis, requiring subsequent parsing logic to assemble a JSON output. These have matured over decades and excel at printed Cyrillic text, offering high accuracy for TTNs' tables and form fields. Newer **cloud OCR services** (Azure, Google, etc.) combine OCR with pre-trained machine learning models for forms, and can directly output structured data like table line items and key-value pairs. They offer ease of integration and pay-per-use pricing, supporting Russian text reasonably well. Meanwhile, **vision-enabled LLMs** such as GPT-4 with image input represent an emerging approach: they can ingest TTN PDFs/images and *directly generate JSON* by "understanding" the document. This LLM-vision approach is promising for complex, varied layouts, but comes with higher costs and some uncertainty in consistency.

For Russian TTNs, **classic OCR + rules/ML** solutions remain highly competitive. Documents like TTNs have consistent tabular structures and labels (sender, receiver, item lines, etc.), which traditional engines handle with carefully tuned templates or models. Solutions like ABBYY FlexiCapture (and its modern platform ABBYY Vantage) are widely regarded for handling Cyrillic forms with top accuracy and customizable extraction rules [1] [2]. On the other hand, **cloud form recognizers** (e.g. Azure Form Recognizer, Google Document AI) now support Russian and can extract line items and key fields without manual template setup [3] [4]. They shine in quick deployment and scalability. **LLM-vision models** such as GPT-4's vision feature offer a different value: they can flexibly read any TTN (including stamps, signatures, or slight layout variations) and output structured JSON by prompt alone [5]. However, they are not strictly more accurate at raw text reading than specialized OCR; their advantage is in understanding context (e.g. knowing a vehicle number is a license plate even if not explicitly labeled). In practice, LLMs might *not* consistently outperform well-tuned OCR on clear printed text, especially for multi-column numeric data, but they can add value in interpreting noisy or handwritten elements and in formatting the output. A balanced strategy might use OCR for what it does best – precise text extraction – and LLMs for higher-level structuring or validation.

Below, we compare 5 top solutions for extracting TTN data. These were chosen across categories (open-source, enterprise on-premise, cloud API, LLM-based, and hybrid frameworks) for their strong performance on Russian documents. We then provide detailed profiles of each, followed by two recommended system architectures (one emphasizing a budget open-source stack, and one leveraging premium AI services for maximum accuracy). Finally, a step-by-step checklist is provided for conducting a proof-of-concept on real TTN samples.

Comparison of Top 5 Solutions

Solution	Type	Quality for Russian TTNs	Table/Form Support	On-premises / Cloud	Approx. Cost	Pros
ABBYY FlexiCapture (Vantage)	Enterprise OCR platform (on-prem or hybrid)	Excellent Cyrillic OCR accuracy; decades of refinement for Russian forms [1]. Handles TTN text, stamps and poor scans very well.	Strong layout analysis; can define table schemas and extract line items with high precision. Built-in form understanding for invoices/ acts.	On-prem deployment (Windows/ Linux); cloud components available (Vantage).	High license cost (enterprise pricing; $$–$$$).	**+** Best-in-class recognition for printed docs [1]; supports 190+ languages [6] (Russian, etc.); highly customizable rules/ templates; works offline (data privacy) [7]; includes verification workflow.
Azure Form Recognizer (Document Intelligence)	Cloud OCR & form AI service	Very good OCR quality for Russian [11]; prebuilt invoice model supports Russian and extracts fields/ line items in Cyrillic docs [11]. Slightly lower accuracy than ABBYY on some fonts or noisy scans [12].	Excellent table and form support: returns structured JSON of tables, key:value pairs, checkboxes, etc. Pretrained models for invoices, receipts, IDs [13] [14]; or train custom models for TTN fields.	Cloud API (Azure); **on-prem containers available** for OCR & layout (v4) [15] [16].	Pay-as-you-go (approx. $10 per 1k pages for prebuilt models [17]; ~$1.50 per 1k for raw OCR) [18].	**+** No setup – ready to use API; outputs structured data (JSON) out-of-box [15]; supports handwriting and printed text; scalable and integrates with Python SDK easily; container option for private deployment.

Solution	Type	Quality for Russian TTNs	Table/Form Support	On-premises / Cloud	Approx. Cost	Pros
Google Document AI (DocAI)	Cloud document AI service	Top-tier OCR quality (Google's OCR has 200+ language support, including Russian) [19] [20]. Accurately reads printed and some handwritten Cyrillic. However, the specialized *Invoice Parser* model currently does **not** include Russian locale in its trained field extraction (supports many European languages but not RU yet) [21].	Strong layout and table detection: can identify table structures, paragraphs, etc. [22]. Has a general Form Parser for key-values, and an Invoice Parser (great for structured fields *if* language supported). Lacks a pre-built TTN template, but custom training is possible.	Cloud API (GCP). No official on-prem (though can use it within GCP region for data residency).	Pay-as-you-go (roughly $1.75 per 1k text pages [18]; specialized parsers slightly more).	**+** Excellent multilingual OCR and layout analysis [19]; powerful table parsing and form understanding using Google's NLP [23]; provides confidence scores and bounding boxes; easy integration via REST/SDK.

Solution	Type	Quality for Russian TTNs	Table/Form Support	On-premises / Cloud	Approx. Cost	Pros
PaddleOCR (PP-Structure)	Open-source OCR toolkit (Python)	Good overall accuracy on Russian print text (comparable to commercial engines for clear prints, slightly lower on tricky cases). Supports Cyrillic in its multilingual models [6] [24]. Quality can approach ABBYY on standard fonts, but may need tuning for noisy scans.	Advanced layout and table support: **PP-Structure** module reconstructs tables and preserves layout into JSON/ Markdown [25]. Detects table cell structure, and even includes a vision-language model for key info extraction (can find fields like totals) [25] [26]. No built-in TTN template, but fully programmable.	On-prem (self-hosted). Runs on Windows/ Linux (CPU or GPU). No cloud fees (you manage infrastructure).	Free (Apache-2.0); cost is infra and dev time.	**+** No per-page cost [27]; complete control over data (runs locally); active community and continuous improvements (designed to rival commercial OCR) [28] [25]; out-of-the-box support for 100+ languages and mixed language docs [6] [27]; fast with GPU acceleration [27].

Solution	Type	Quality for Russian TTNs	Table/Form Support	On-premises / Cloud	Approx. Cost	Pros
GPT-4 Vision (OpenAI multimodal)	Vision-enabled LLM (cloud API)	Impressive ability to **"read" and interpret** Russian documents. GPT-4 can handle Cyrillic text and even some handwriting in TTNs (given its training on diverse data). It often accurately transcribes content and understands context like table headers vs values. However, it may occasionally mis-read characters or omit subtle details (not a dedicated OCR engine), so quality is high on average but not guaranteed 100% consistent. Best for complex understanding rather than raw OCR precision.	No explicit table structure output – but via prompting, it can output JSON arrays of line items. Essentially, the model internally parses the layout (it "sees" the table) and you instruct it to structure the output. It can distinguish columns if prompted with examples (e.g., output each line's product, quantity, price, etc.). It will include stamps/ signatures only if asked (it might ignore noisy stamps by default).	Cloud only (OpenAI or Azure OpenAI service). No local self-hosting of GPT-4.	Expensive: billed by image-text tokens. A single page image can consume **thousands of tokens** [30] (OpenAI charges ~$0.03–$0.06 per 1k tokens), so cost might be ~$0.5–1 per page or more, depending on content.	**+** No need to develop parsing rules – the LLM understands the doc and directly yields structured JSON by following a prompt [5]; extremely flexible – works on any layout or noisy document (robust to stamps, handwriting, slight rotations, etc.); can incorporate reasoning (e.g. can total up values or normalize date formats by itself). Rapid to deploy if you have API access.

Sources: Comparison synthesized from vendor documentation and independent evaluations [1] [19] [25] [5].

Detailed Profiles of Top 5 Solutions

1. ABBYY FlexiCapture (FineReader / Vantage)

Overview: ABBYY FlexiCapture is a leading enterprise platform for intelligent document processing. It combines OCR with layout analysis and rule-based extraction. ABBYY's heritage in Cyrillic OCR (originating in Russia) makes it particularly strong for Russian documents. The system includes a designer to define templates ("FlexiLayouts") or train machine-learning classifiers for forms. The newer ABBYY Vantage platform offers pre-built "skills" (like invoice processing) and a cloud-friendly architecture, but the core OCR is based on ABBYY's FineReader Engine, renowned for accuracy.

Architecture & Pipeline for TTN: A typical ABBYY solution for TTNs would involve scanning the document into the system's input, performing text recognition and field extraction using a defined template or ML model, then outputting structured data (e.g. JSON or XML). Specifically for a TTN, one would configure a **table block** to capture the "Goods Information" table: ABBYY can locate the table by lines or by anchor keywords and extract each cell (item name, quantity, weight, etc.). It excels at recognizing cell boundaries even if lines are faint, using internal table recognition algorithms. In parallel, field elements in the header (e.g. sender, receiver, date, vehicle number) can be captured via keywords (like "Отправитель: ___") or trained detectors. The result is a hierarchy of fields and line items which ABBYY exports to a JSON/CSV. If something doesn't parse confidently (e.g. a smudged number), ABBYY's verification station allows a human to correct it before final output.

Best Practices & Tuning: To get the most out of ABBYY for TTNs, implement image pre-processing (ABBYY has built-in preprocessing for skew, noise). One effective practice is to create a **FlexiLayout template** specifically for the standard TTN form (many Russian TTNs follow GOST or standard forms). By defining regions for the table and key blocks, you constrain the OCR to expected areas, improving accuracy. ABBYY's language setting should be tuned to Russian (and any English if alphanumeric codes appear) to reduce errors. If the TTNs have stamps or signatures overlapping text, these can be filtered out by ABBYY's image filters (e.g. "remove colored stamps" filter) or by excluding those regions in the template. For the item names, ABBYY can use dictionaries or pattern matching (if you have a catalog of goods, it can help recognize partial or misspelled names, though this may require custom coding). ABBYY also allows scripting (in .NET or JScript) for post-processing; for example, one can script a check that "quantity * price = amount" for each line and flag discrepancies – useful for validation.

Handling Challenges: ABBYY generally handles noisy prints and low-quality scans better than most, given its long development in OCR. It uses adaptive recognition and can even read dot matrix prints common in older forms. For **handwritten notes**, ABBYY has ICR (intelligent character recognition), but results vary – if someone wrote in pen on the TTN, you might need to enable a handwriting recognition stage for that region. If ABBYY's out-of-the-box Russian model confuses certain characters (say `0` vs `0`), you can adjust its language/alphabet settings to avoid ambiguous characters in certain fields. A known limitation is that ABBYY's full power comes with careful setup – e.g., it may require separate configurations for **different TTN formats** if layout differs significantly. Many companies address this by training multiple document definitions (one for old format TTN, one for new standard form, one for other related docs like УПД) and letting ABBYY classify incoming pages.

Integration: ABBYY provides a robust Python and REST API nowadays (especially via Vantage or FlexiCapture SDK). A Python backend (FastAPI, etc.) can call ABBYY's engine, passing a PDF and receiving JSON. One can also deploy ABBYY in an on-prem server that the backend communicates with. The documentation and community are strong – ABBYY has support forums and integrator partners in

Russia/CIS familiar with TTN use cases. Community activity is moderate (less open than open-source projects, but ABBYY experts do share insights on forums).

Evaluation: ABBYY often achieves near the highest accuracy among all solutions for structured Russian documents. It's typical to see character recognition accuracy >99% on clean prints, and field-level accuracy similarly high when templates are correctly set [1]. The table structures are accurately delineated (it rarely swaps columns, thanks to explicit template definitions). The chief drawbacks are cost and complexity – the license could be expensive if this is a smaller-scale project, and the time to configure might be weeks. However, for a mission-critical, high-volume TTN automation in a large enterprise, ABBYY is a gold standard due to its reliability and on-prem data security.

2. Azure Form Recognizer (Document Intelligence)

Overview: Azure Form Recognizer (recently rebranded under Azure AI Document Intelligence) is Microsoft's cloud service that combines OCR with pre-trained models for certain document types. It is a **serverless API** – you send it a document, and it returns JSON with extracted text and fields. It supports a wide range of languages and was expanded to support Cyrillic and Russian handwriting by 2023 [12] [11]. For TTNs, the most relevant out-of-box model is the **Invoice model**, which is trained to extract common fields (buyer, seller, invoice date, line items, totals, etc.) from invoices. Notably, this model **does include Russian** as a supported language [11], so it can recognize Russian terms like "№ накладной" as invoice numbers, etc., albeit it might label them in English in the JSON (e.g., "InvoiceDate": "..."). Azure also provides a **Layout model** which simply gives text lines, tables, and coordinates, if a fully custom approach is needed.

Pipeline for TTN: Using Form Recognizer for TTNs can be done in two ways: (1) Use the prebuilt invoice model directly. This requires little effort: you upload the TTN PDF/image to the API (or via SDK), specify `model=invoice`, and get a JSON response that includes extracted fields. For example, it will return a list of `lineItems`, each with `description`, `quantity`, `unit price`, `amount`, etc., which often align well with TTN "goods" table lines. It will also return top-level fields like `vendorName` (shipper), `customerName` (receiver), `invoiceDate` (date). In testing, these fields map intuitively: companies have found that the invoice model works for shipping documents since many fields overlap with invoices. (2) Alternatively, for full control, one can use the `layout` model to get all text and bounding boxes, then do custom parsing. But a more efficient approach might be to train a **Custom model**: Azure allows you to label your own documents and train a model to extract specific fields. For example, you could label 5-10 TTNs indicating where "Vehicle Number", "Driver Name", etc. are, and train a custom model that directly outputs those. This custom model training does not require coding – it's done via Azure's Form Recognizer Studio UI. The result is an endpoint that will give you a JSON with exactly the TTN fields you defined (including the table). Given the small project timeline, one might initially use the prebuilt model and later refine with a custom one if needed.

Performance and Scaling: Azure Form Recognizer is optimized for scale – it can process thousands of pages per day easily, as it's a cloud-managed service. Throughput can be increased by parallel API calls (with some rate limiting to watch). In terms of speed, each page is typically processed in ~1-2 seconds (slightly longer if using the invoice model due to its complexity). The service returns confidence scores for each field, which is useful to identify uncertainties (e.g., low confidence might trigger manual review or secondary processing). Azure's OCR quality on Russian print is high – comparable to Google's. It reliably reads Cyrillic characters, and the layout engine identifies table structures correctly in most cases (it will list table cells by row and column).

Integration & Example Code: In a Python backend, one would use the `azure-ai-formrecognizer` SDK. For instance, using the prebuilt invoice model could be as simple as:

`client.begin_recognize_invoices(from_url=document_url)`, then polling the result. The result object will have fields like `fields["VendorName"].value`, etc. For on-prem scenarios, Azure offers containers – e.g., one can run a Docker container locally that provides the OCR/layout model, but note that the invoice model container was not available at last update (only layout and custom models can run on-prem). So, in a sensitive environment, one strategy is to run the layout container on-prem to get raw data, then use a local code to parse or even call an on-prem LLM.

Accuracy & Limitations: In practice, the Azure invoice model does a good job with multi-column tables and rarely mixes up columns – it uses positional understanding to keep quantities and prices aligned properly [31] [32]. It can sometimes misclassify a field; for example, we might see it output something in the `ShippingAddress` that was actually the destination if the layout is unusual, but for standard Russian forms which have clearly labeled sections, this is infrequent. One limitation is that any field it doesn't know (like "Vehicle number" or "Driver's license ID" on a TTN) will not appear in the output – that data isn't lost (it will be in the full text or table) but not in a named field. You'd have to retrieve it manually from the text. This is where either a custom model or post-processing with regex/keyword on the raw text can help. Another limitation is handling of **stamps or signatures**: these might appear as irrelevant text blobs in the output (e.g., a circular stamp might get partially recognized as random letters). The Azure model doesn't have a concept of "ignore stamp," so one should either preprocess images to drop colored stamps or post-filter unlikely text (e.g., certain gibberish strings).

Overall, Azure Form Recognizer provides a **quick deployment** path – within a day you can be extracting TTN data with decent accuracy. It's particularly attractive if you already use Azure or want easy scaling. The costs are reasonable (for example, ~\$10 for 1,000 pages for the invoice model [17], which for thousands of TTNs per day is far cheaper than an LLM approach). And critically, it now supports on-prem container execution for core OCR, addressing data residency concerns.

3. Google Document AI

Overview: Google's Document AI is a close counterpart to Azure's service. It offers specialized parsers (invoices, receipts, etc.), a general form parser, and a powerful layout/OCR engine. Google is known for its research in computer vision and has deployed an OCR that can read over 200 languages, including Russian, with high fidelity [20] [33]. For TTNs, Google's **Form Parser** or **Document OCR** will likely be used. The dedicated **Invoice Parser** is highly capable for many languages, but as of the latest docs, Russian isn't in its pretrained set [34] – so while it will OCR all the Russian text, it might not classify field labels in Russian into its schema. Nonetheless, Google's engine can still extract the table structure and text content, leaving the mapping of fields to the developer or a potential fine-tune (Google allows something called "Uptraining" on their parsers with your data [35]).

Using Google Doc AI for TTN: A straightforward approach is to call the *Document OCR (aka Enterprise OCR)* to get a JSON with all text, including structural data (paragraphs, lines, coordinates). This output is similar to Azure's layout: it includes a list of pages, each with blocks, lines, words, and detected tables. The table data comes as cells with coordinates and text, which you can assemble into rows/columns. This means you get the TTN's line items effectively as a table structure but without semantic labels on columns. Then you could interpret the first row as headers (if present in the document) and know that, e.g., column1 = item name, col2 = quantity, etc., by position. The header fields like shipper/consignee would come as text blocks which you'd need to map (e.g., find text after "Грузоотправитель:" label). Google's newer **LayoutLM-based parsers** can assist – for instance, the general Form Parser model is trained to detect key-value pairs in forms, so it might automatically pair "Отправитель" with a value (company name) in the output JSON. This can work even if it doesn't fully understand the language, simply by layout, but it's not guaranteed for every field in Russian.

Another option is to try the Invoice Parser on a TTN. It will yield a JSON with invoice fields in English (like "supplier_name", "total_amount"). If the TTN closely resembles an invoice in structure, some fields will populate (e.g., total sum, dates, names) because the model can often infer them by position even if the labels are Russian. But others might be blank or wrong due to language differences. Uptraining (fine-tuning) the model on a Russian TTN dataset could then teach it new field labels, but that requires a decent set of labeled docs and is a more involved project.

Quality and Edge Cases: Google's OCR quality is among the best – it's particularly good at difficult cases like skewed photos or mixed-language text. So for example if a TTN item description contains both Russian and an English part number, Google will handle that seamlessly (its multi-language OCR can process Latin and Cyrillic together [20] [33]). For **tables**, Google's algorithms (dating back to DeepMind's research) do a great job connecting cell text that is spatially aligned. If a TTN has merged cells or some irregular table formatting, there is a chance of cell segmentation issues, but generally standard ruled tables are correctly recognized. One known strong point: Google OCR can detect **handwriting** in documents (it supports 50+ languages for handwriting) – if someone has written quantities or signed in Cyrillic, it can often transcribe that as well [36] [37]. You might get a JSON output where a signature is just represented as illegible text (if the signature is cursive, these systems usually output `########` or similar for unreadable parts, or very low confidence letters). That can serve as a flag that a signature exists.

Integration: Google Document AI has a REST API and client libraries (Python, etc.). The usage in Python might involve their `google.cloud.documentai` library. Example: you create a processor for "OCR" or "Form Parser" and then call it with the file. The response includes text blocks and entities. For development, Google provides an online Document AI Workbench where you can visually see what it extracts – this is useful for a quick evaluation on a few TTNs to decide which model to use. Keep in mind that Google's service is cloud-only; however, it offers regional processing (you can choose EU servers, etc., which might be important for compliance).

Costs: The cost is similar to Azure's – on the order of a few cents per page for form parsing. The snippet above suggests ~$1.75 per 1000 pages for base OCR [18] which is extremely low. However, specialized models like the invoice parser might cost more (Google's pricing for invoices might be around \$0.03 per page – not publicly stated here, but typically a bit higher than plain OCR). They also have a free quota for a small number of pages per month which could cover a small-scale pilot.

Limitations: Without native Russian field support, Google might require more custom work than Azure for TTNs. But given the high OCR fidelity, that custom work could simply be applying regex to the full text. For example, to get the "vehicle number", one could regex for something like `[АВЕКМНОРСТУХ]{2}\d{3,6}` (since Russian plates have certain patterns) in the text output. Similarly, totals can be found by looking for a currency symbol or the word "Итого". This hybrid approach (Google OCR + custom code) is akin to using Tesseract, but with far better initial OCR quality and layout info. One more consideration: **Data privacy** – sending docs to GCP might be a concern; if that's the case, Azure's on-prem container or ABBYY on-prem would be preferred. Google currently does not offer an on-prem container for Document AI.

In summary, Google's solution is extremely capable on the technical side (likely no issues handling TTN prints), but integrating it for structured output will either use a not-perfectly-aligned prebuilt model or require some coding. It is a strong candidate if you value the underlying OCR accuracy and are willing to bridge the last mile for TTN-specific fields yourself.

4. PaddleOCR (Open-Source Engine with PP-Structure)

Overview: PaddleOCR is an open-source OCR library from Baidu that has grown into an "OCR + Document AI" toolkit. It's written in Python (with some C++ for speed) and can run on CPUs or GPUs. For our use case, the key feature is **PP-StructureV3**, the component of PaddleOCR that performs layout analysis and table recognition [25]. Essentially, PaddleOCR can take a TTN image and not just give you the text, but also a structured representation (like JSON or even Markdown) that mirrors the document's layout [25]. This includes detecting tables, which is crucial for TTNs. The library supports Russian out-of-the-box; its multilingual text recognition model (PP-OCRv3 multilingual) is trained on 80+ languages including Russian, so it can accurately transcribe Cyrillic text.

Pipeline & Tools: Using PaddleOCR for TTN might involve these steps: First, use the detection model (`PaddleOCR` has a text detector for finding text regions). For a form, the detector will find blocks of text – it might, for instance, outline each cell of the table or each line of text in the header. Next, the recognition model transcribes each outlined region. PP-Structure then tries to make sense of all those text boxes: it can group them into a table if it detects lines/columns, or into a list of key-value pairs if it sees a label and value alignment. PaddleOCR can directly output an HTML-like or JSON representation of tables (the documentation shows converting documents into HTML with preserved structure). For TTNs, one strategy is to detect only the table region first (maybe by looking for the "Goods information" table by keywords or large table lines using an image segmentation approach) and run table structure recognition on that region for higher accuracy. PaddleOCR also introduced **PaddleOCR-VL (Vision-Language)** in version 3.3 [38], which effectively is a smaller inbuilt model (0.9B parameters) that can understand document elements in a more holistic way. This can be used to extract key information (similar to an LLM but lightweight). For example, they have **PP-ChatOCR** which can take questions or tasks – one could potentially prompt it: "Extract the seller, buyer, and total amount" and it would use the OCR results plus an understanding model to output those answers [39]. This is an advanced feature that might require some setup, but shows that PaddleOCR is moving toward a hybrid OCR-LLM approach internally.

Best Practices: Since PaddleOCR is free and open, one can customize a lot. For TTNs, you might do some image preprocessing manually: e.g., apply an OpenCV filter to remove color stamps (if they are red or blue, drop that channel) before OCR, so the stamp text doesn't confuse the OCR. You can also configure the OCR to restrict its character set – if you know a field is numeric, you can post-filter non-numeric characters. While PaddleOCR's default models are general, you can fine-tune on specific data if needed. For instance, if the default recognizer has trouble with some specific font used in your TTNs, you could gather a small dataset and fine-tune the recognition model (this is non-trivial but possible with their tools). Similarly, you could train the detection model to better segment the TTN table if needed. However, in many cases the pre-trained models are sufficient.

Output & Parsing: Once PaddleOCR gives you a JSON of the document, you'll need to map it to your desired output schema. It might give coordinates and text of each cell – you would then produce your list of items by reading that JSON. If PP-Structure's table output is used, it might literally give the table as an array of rows, which is very convenient. For the header fields, you might get something like a list of key-value pairs where key is "Отправитель" and value is the name. If not, you can manually map by finding the text near those keywords. Writing a parser for Paddle's raw output likely involves iterating through recognized text boxes and using heuristics (e.g., anything above the table region is a header field; anything in a certain area of the page corresponds to specific metadata).

Performance: PaddleOCR is optimized in C++ and supports batch processing on GPU. It can process a page in a fraction of a second on a modern GPU, and a couple seconds on CPU. This means even thousands of TTNs per day is feasible on modest hardware (one might deploy a service with, say, 2 CPU

cores and handle a doc every 2-3 seconds; or with a GPU to do dozens per second). The only caveat is memory usage when using large models (the vision-language model is 0.9B, which might use a few GB of RAM). But you can choose not to use that and just stick to detection+OCR+structure which is lighter.

Limitations: Being an open solution, PaddleOCR might mis-structure something occasionally. For example, if the table lines are faint, it might not perfectly split a row, causing two item lines to merge into one in the output – you'd then have to catch that by validation (like if one row has two product names, maybe split it). Or the opposite: if an item description is long and wraps to a second line within one cell, the OCR might think it's two separate rows. Commercial tools handle many of these cases with sophisticated rules; with Paddle, you may need to implement some logic to merge or split lines based on context (e.g., if a "quantity" column is empty on a row but the next row has data, probably the text was a wraparound from above). Additionally, while Paddle supports handwriting recognition, it's not as strong as Google's or ABBYY's for cursive script – it's mainly tuned for printed text. So handwritten notes on TTNs might be missed or require a separate approach (like using a smaller dedicated model for digits if someone writes a number by hand).

Conclusion: PaddleOCR provides a **zero-cost, fully on-prem** solution that can achieve a lot of what expensive platforms do [27] [29]. The trade-off is that *your development effort* replaces what those platforms provide turn-key. For a test project or a budget-sensitive deployment, PaddleOCR is an excellent starting point: you can quickly set up a pipeline and see results. The flexibility also means you can integrate it into a Python stack easily (there are no license restrictions, and you can modify the code). Many developers in the community have contributed, ensuring that for common tasks like table extraction, there are examples available [40]. For instance, some have published sample code on how to use PaddleOCR to parse tables into CSV. Utilizing those resources, one could get a TTN-to-JSON prototype running relatively fast, and then refine specific corner cases as they appear.

5. GPT-4 Vision (Multimodal LLM)

Overview: GPT-4 with vision capabilities represents a new paradigm: instead of explicitly programming how to parse the document, you *ask* the model to do it. The GPT-4 model (as of late 2023/2024) can accept images or PDFs as input and process their content in the context of a chat prompt. Several other models (e.g. Claude 3 potentially, Google Gemini in the future) are expected to have similar image understanding. Here we focus on OpenAI's GPT-4. It was trained on a vast amount of web images and documents, which likely included forms and invoices, so it has some inherent knowledge of layout and semantics. For example, it "knows" what an invoice or shipping document typically looks like and what information is usually present. This is not a deterministic parsing – it's generative and inferential.

How it works for TTN: The usage scenario is: you provide the TTN document (either as an image or PDF file) to the GPT-4 API, along with a prompt that instructs it to output a structured result. Typically, one would use a **system message** like: *"You are an assistant that extracts structured data from documents. The user will send an image of a TTN. Extract the following fields in JSON: sender, receiver, date, each line item with product, quantity, weight, price, etc., and totals."* Then in the user message, attach the image/PDF. GPT-4 will then produce a JSON response with those fields filled. The power of this approach is that you don't need to explicitly tell it how to find something – GPT-4 will read the Russian text (it's fluent in Russian as well) and figure out, for instance, that "Грузоотправитель: ООО Ромашка" means the sender's name is "ООО Ромашка". It will understand table columns by context (e.g., a column with all numbers like "20, 15, 10" next to product names might be quantities or weights; if one column has "kg" in some entries, that's likely weight; if another has currency symbols or looks like an amount, that's price or total). GPT-4 is capable of reasoning to some extent: if the table doesn't explicitly label "Net weight" vs "Gross weight" but has two numeric columns, it might deduce which is which by typical order or values.

Output and Accuracy: When prompted to strictly output JSON, GPT-4 will generally follow the format given. You typically provide an example JSON structure in the prompt [41] [42] (with empty strings or nulls as placeholders) to guide it. The result can be impressively structured – it might include an array of items with each field correctly placed. Users have reported success in using GPT-4 vision to extract data from invoices and receipts without any explicit OCR step [41] [42]. In those cases, GPT-4 not only read the text correctly but also *normalized it*. For example, if a date is written as "12.01.2025" on the document, GPT-4 might output it as "2025-01-12" if asked to normalize date format. It can also do things like currency conversion or tax calculations if you ask (though for TTN we likely just want raw data).

However, GPT-4 is not infallible. It might miss subtle details: e.g., if there is a very faint stamp saying "COPY", it could ignore it (which is fine), or it might misinterpret a smudged character in a product name where a pure OCR might still get it by pattern. Also, GPT-4 has a context length limit (and image inputs get chunked into patches of 512x512 pixels [30]); extremely long documents or multi-page TTNs would eat into those limits. A multi-page TTN might need to be processed page by page. There's also a risk of minor hallucinations – usually, for a straightforward extraction, GPT-4 sticks to the text, but if the prompt is not constrained, it might "fill in" a value that was actually missing or unreadable (e.g., if an item line is partially cut off, GPT might guess the rest of the word, whereas a normal OCR would just give partial text or an error). Therefore, for high-stakes data, one would implement a validation layer: for instance, cross-check that numeric totals add up, or even run a secondary OCR to verify critical fields.

Integration: Currently, GPT-4 Vision is accessed through OpenAI's API (or Azure OpenAI). One would send the document as base64 or as a file to the API endpoint with the prompt. The integration is straightforward in Python using the `openai` package: you construct a message with an attachment (the image) and get a response. The challenge is handling errors or timeouts (the model might take several seconds, especially if the document text is long, because it has to "read" potentially thousands of characters and then formulate the JSON). There are also rate limits and concurrency considerations – it's not trivial to process thousands of pages quickly without hitting API limits, though one could parallelize with multiple API keys or processes to some extent.

Cost Considerations: As noted, GPT-4 Vision can be quite expensive per page [30]. The image is converted to tokens internally – OpenAI has indicated that they split images into a grid of 256x256 or 512x512 patches which then are encoded. A full A4 page might become tens of thousands of tokens. The response JSON is also some tokens (though that's usually smaller). The charges at e.g. $0.03 per 1k input tokens mean if a page yields, say, 15k tokens, that's $0.45 just to process one page. If TTNs are one page each, that's 45¢ each; 1000 such pages would be \$450 – far above using an OCR service (which would be ~$10-$20 for 1000 pages as we saw for Azure/Google). So GPT-4 might only be cost-justifiable for relatively low volume or for a **"last resort" processing of documents that other methods failed on**. It's also possible that the cost will come down over time or that smaller multimodal models (like GPT-4 "mini" they mentioned, or open-source ones) could be used at lower cost, albeit with some quality trade-off.

Augmenting vs Replacing OCR: It's important to note that GPT-4 Vision could be used in a hybrid way. For example, one could first run a cheaper OCR (like PaddleOCR or Azure layout) to get raw text, and then feed that text into GPT-4 (text input, not image) asking it to structure it. That reduces cost dramatically because text tokens are far fewer than image tokens. GPT-4 on text can still apply reasoning (like understanding which piece of text is which field). This approach might miss some context that the image provides (like table column alignment), but one can encode positions in the prompt. Alternatively, one could use GPT-4 vision for just specific tricky parts of the document rather than the whole thing – e.g., isolate a region with handwritten notes and ask GPT-4 to read just that. This "spot use" can be effective.

Reliability: When GPT-4 works, it feels like magic – one can throw a messy scan at it and get nicely formatted JSON back. But for production, consistency is key. Testing on a batch of sample TTNs would be needed to see if it ever produces malformed JSON or swaps field names. Usually, if you provide a strict schema example, it will adhere to it. There have been community experiments where GPT-4 was given an invoice and it returned JSON with all line items correctly [43]. Users note that sometimes it might drop a line or mis-order them if the prompt wasn't explicit. So prompts might include instructions like "Make sure every line item from the table is included in the output." The model's ability to distinguish columns is typically good if the table has clear headers or separators; if not, it uses content (like it knows quantity will be a number, etc.). On very complex tables, I have seen GPT-4 sometimes output a combined field instead of separate (like merging two columns into one if it wasn't sure). Those are edge cases to be mindful of.

Privacy & On-Prem: As a cloud service, GPT-4 is not usable for highly sensitive docs unless the organization is comfortable with OpenAI's data policies (which currently say they don't use API data for training and will delete after some time). There is no on-prem version of GPT-4. Some companies are exploring local open-source multimodal models (like LLaVA, Donut, etc.), but as per instruction, those are not yet at GPT-4's level and not considered here.

In conclusion, GPT-4 Vision is a cutting-edge solution that can drastically simplify development (no need to code parsing) and handle a wide variety of document layouts. For a small-scale scenario or a one-off project where quality is paramount and cost is secondary, it could be the best choice. However, for large-scale automation of TTNs, it's likely to be an expensive component, so one might use it selectively – for example, use a cheaper OCR pipeline for 95% of cases and fall back to GPT-4 for the hardest 5% (like documents with unusual structure or poor quality that the OCR couldn't confidently parse). This combination can yield an optimal balance, leveraging GPT-4's intelligence without incurring its full cost on every document.

Recommended System Architectures

Based on the above, here are two viable architecture scenarios for a TTN recognition system, tailored to different priorities:

A. Budget-Friendly Open-Source Pipeline

Tech Stack: *PaddleOCR* (for detection, recognition, table parsing), *Python* (FastAPI or Flask backend), optional *rule-based post-processing*.

Pipeline Steps:

1. **Preprocessing:** Convert incoming TTN PDFs to images (if they are digital PDFs, you might extract the text layer instead – but assuming scans for generality). Apply image cleanup: e.g., binarization to remove light artifacts, and color drop to remove stamps (since we only need printed text). This can be done with OpenCV or Pillow in Python.

2. **OCR & Layout Analysis:** Run PaddleOCR's detection and recognition on the preprocessed image. Use the PP-Structure module to get structured data [25]. This will yield (a) a JSON or Python dict of text blocks, and (b) identified tables with rows and cells content. If PaddleOCR doesn't automatically separate the header fields, you'll have all text lines with coordinates.

3. **Parsing to JSON:** Write a parsing function that takes Paddle's output and constructs the final JSON. For example:

4. Identify the "Goods table": Paddle might provide it directly. If not, you can detect the largest table by looking at coordinates or number of contiguous text columns. Assemble each row into a line item JSON (with fields like name, quantity, weight, etc.). Since Paddle preserves reading order, cells in the same row should come sequentially. Map them to the correct key by position (e.g., if you know column 3 is weight, assign the text of the 3rd cell to `"weight"`).
5. For top/bottom sections (sender, receiver, etc.), search the recognized text for known keywords ("Отправитель", "Получатель", etc.) and take the text following them as the value. Because Paddle gives coordinates, you can alternatively locate the field value which often appears consistently to the right of the label.

6. Compile all extracted fields and the line items array into the JSON schema you defined.

7. **Post-processing & Validation:** Implement checks such as:

8. Data type conversions: numbers like quantities/masses should be output as numbers (not strings). You might need to parse "1,500" as 1500 (watch out for comma vs dot in Russian locale).
9. Totals verification: if there's a total sum field, compare it to the sum of line item amounts to flag any discrepancy.
10. Completeness checks: ensure that the number of line items in JSON matches the number of lines detected; if not, possibly run a secondary check or mark as needs review.

11. Normalize text: e.g., trim whitespace, unify date format (convert "12.01.2025" to "2025-01-12" if needed, or to a `YYYY-MM-DD` string).

12. **Output:** Return the structured JSON via an API endpoint. This JSON contains everything required by downstream systems.

Metrics to Track: During development and testing, measure: - **Field accuracy:** What percentage of fields (e.g., sender name, each line's quantity) are correct? This requires a ground-truth set of TTNs to compare against. - **Line item recall:** Did the system extract all line items present? If some were missed or merged, that's a recall issue. - **OCR character error rate:** Helpful to gauge raw OCR quality; e.g., if item names have typos after OCR. - **Processing speed:** Time per document. On CPU, maybe it's 2-3 seconds; on GPU, could be <1 second. Ensure it meets the throughput requirement (thousands per day means a sustained rate of e.g. one every few seconds is fine). - **Cost:** Here mainly the infrastructure – if on-prem server, the "cost" is hardware + maintenance. This might be negligible if using existing servers.

Scalability: This pipeline can be scaled by running multiple instances of the OCR service. PaddleOCR on GPU can handle many requests sequentially; if volume grows, you can spin up another container/pod with PaddleOCR. Because it's stateless, load balancing is straightforward. One caveat: if you need to scale **horizontally** and use GPU, ensure each instance has a GPU or use CPU with more instances (since CPU is slower).

Quality Improvement Strategy: Start with the pre-trained PaddleOCR models. Evaluate on, say, 50 sample TTNs. If you notice consistent mistakes (e.g., misreading of the letter "З" as "3" in certain font), consider customizing the language model or using PaddleOCR's ability to blacklist characters (maybe restrict recognition to Cyrillic + common symbols to avoid confusing with similar Latin characters). If table detection fails on some layouts, you could incorporate a heuristic: e.g., if Paddle doesn't output a table but you know one should exist, use the raw text lines to reconstruct it (maybe by detecting

consistent spacing or using lines as separators – OpenCV can detect drawn table lines as well). Over time, as you gather more TTN variations, you might decide to label a small set and fine-tune Paddle's detection model to better handle them (this is advanced and only if absolutely needed). Simpler, you might add some rule-based adjustments: for instance, if "quantity" and "weight" got swapped in output (maybe due to no headers in the image), you can identify that the weight column usually has "kg" or typically larger numbers than quantity, and swap them back.

Pros & Cons: This architecture is very **cost-effective** and keeps data in-house. It gives good accuracy (maybe not as perfect as a finely-tuned ABBYY, but with effort can come close). The downside is the engineering effort – you are essentially replicating what commercial systems have done. But given it's a test project (a POC), this effort is manageable and actually beneficial for learning the edge cases. If, for example, after POC, results show that accuracy is 90% and not improving easily, one could then justify moving to a paid solution. But if you reach, say, 97% accuracy with open source, you might stick to it and save cost. This architecture is also modular: you can swap PaddleOCR with another OCR (Tesseract, MMOCR, etc.) if needed, or integrate a small ML model for field classification later.

B. High-Accuracy Hybrid with Commercial OCR + LLM

This scenario aims for the **best quality** and still reasonable cost by combining a reliable OCR service with a large-language model for refinement.

Tech Stack: *Azure Form Recognizer* (cloud OCR+extraction) as primary, *OpenAI GPT-4 (text-only)* as a secondary step for validation/augmentation, *Python* backend orchestrating calls. (Alternatively, ABBYY on-prem could replace Azure for primary OCR if available; and GPT-4 or another LLM for secondary.)

Pipeline Steps:

1. **Primary Extraction (OCR+Structure):** Send each TTN to Azure Form Recognizer's **prebuilt Invoice** model (or a custom trained TTN model if that has been developed). The response will include structured data: seller, buyer, date, line items (with description, quantity, amount, etc.) [44] [32]. Assuming the TTN is similar to an invoice, most of the table will appear under `lineItems`. Some fields specific to TTN (like vehicle info) won't be there, but the raw text can be retrieved via the `ReadResults` (which gives full text and coordinates).

2. **Secondary LLM processing:** Now take the output from Azure and check for any gaps:

3. If Azure missed fields (e.g., vehicle number, or maybe it put the entire "destination address" in one field but you want to split city/ZIP – any nuance not handled), formulate a prompt for GPT-4 to extract or refine that. A simple way is to feed GPT-4 the *raw text of the document* (which we got from OCR) along with instructions. For example: "The following is text extracted from a TTN document. Extract the vehicle number and driver's name." Provide the text. GPT-4 will output those details. This is using GPT-4 in **text mode** rather than vision, which saves cost (we rely on Azure's OCR for the heavy lifting).
4. Additionally, use GPT-4 to **verify** critical fields. For instance, ask GPT-4 (with the extracted JSON passed in) to double-check if the sum of line item totals equals the reported total, and if not, identify the discrepancy. GPT-4 can perform such checks or even re-calc totals if needed.
5. If Azure's structure needs transformation (say Azure returns field names in English but you want Russian or specific JSON schema), GPT-4 can easily map/rename keys in JSON or fill slight differences. However, this could also be done with straightforward code – GPT-4's unique value is more in understanding freeform text.

Essentially, GPT-4 acts as a *smart post-processor*: it uses its understanding to catch things a rigid system might miss. For example, if a note on the TTN says “Deliver by 5 PM” scribbled on it, GPT-4 might catch that if prompted, whereas Azure would ignore it as un-modeled data.

1. **Merging Results:** Combine Azure’s structured data with any additional pieces from GPT-4. For instance, Azure gave you line items and totals, GPT gave you vehicle and driver, so you merge those into one final JSON. If GPT-4 flagged any issues (maybe it says “The totals do not match, possibly an OCR error on line 3.”), you could include a field like “needs_human_review”: true, or attach the comment for a human checker.

2. **Output:** The final JSON is returned. It contains all relevant fields, presumably with very high accuracy due to the double extraction and validation. Data stays consistent because any potential OCR mis-read might be caught by GPT-4’s language reasoning. (E.g., if Azure read an item name slightly wrong but GPT-4 using context corrects the spelling to a known product name, you might choose to trust GPT’s correction if it looks reasonable).

Metrics: - **Field accuracy and recall** should be extremely high here. We measure how often any field is wrong after the two-step process. Ideally, GPT-4 fixes what Azure missed, so we might get, say, 98-99% field correctness. - **Comparison of steps:** We can log Azure’s output vs final output differences to see how much GPT-4 is modifying. This helps ensure GPT isn’t making unnecessary changes. - **Processing time:** Azure might take ~1-2 seconds per page, GPT-4 (text prompt with maybe a few thousand chars of text) might take another 1-2 seconds. So total ~3-5 seconds per doc, plus overhead. Well within limits for batch processing thousands per day (concurrent calls can be made if needed). - **Cost:** Azure cost for 1000 pages ~ $10. GPT-4 text cost for, say, 1000 pages: if each prompt+response is ~2000 tokens, that’s 2M tokens, which at $0.06/1k = $120. So combined maybe ~$130 per 1000 docs. This is an estimate, and can be lower if prompts are smaller or using the OpenAI 16k context model which might be cheaper per token. It’s still an order of magnitude more than pure OCR, but significantly *less* than using GPT-4 on images directly. If volume is 10k pages, that’s $1300 – maybe acceptable for a business if the accuracy gains reduce manual work.

Scalability: Azure Form Recognizer can scale via the cloud (just respect their service limits or purchase higher throughput if needed; generally fine unless tens of thousands per hour). GPT-4 API has rate limits per minute, but one can apply for higher limits. If needed, the process could be multi-threaded: e.g., process 5 documents in parallel (calls to Azure are async, calls to GPT can also be async). With careful pipeline management, a throughput of dozens per minute is reachable, which suffices for thousands per day.

Advantages: This architecture leverages the strengths of each component: - Azure (or ABBYY) provides a **solid backbone** of OCR and structured extraction for most fields with high confidence. - GPT-4 injects **flexibility and intelligence**, handling any text that is not covered by the predefined model, and doing holistic checks. - On-premise possibility: if using ABBYY instead of Azure, you keep data entirely internal and still could use an LLM (though if it’s OpenAI’s, data goes out; one could use an Azure-hosted GPT-4 in a region ensuring compliance, or a smaller local LLM for validation if really needed).

Potential Pitfalls: One must be careful in prompt engineering for GPT-4 to not mislead it. For example, if you show it Azure’s JSON and ask “Is everything correct?” it might overzealously change things that were fine. It’s better to ask specific tasks (extract X, verify Y). Also, error handling: if GPT-4 fails or gives an error (rare but possible), the system should have a fallback – perhaps just return Azure’s output or try again. Since we have two external services, there are two points of failure, so robust retry logic and validations are needed.

Continuous Improvement: Over time, this system can learn from itself. For instance, if manual reviewers still find errors, you can analyze whether they were due to Azure misreading or GPT mis-structuring. If Azure frequently misreads a particular field (say it confuses "O" vs "0" in one field), one could add a custom post-process rule or even train Azure's custom model with those examples to improve it. If GPT-4 made an incorrect inference (maybe it merged two lines incorrectly), you adjust the prompt to clarify that scenario. Because GPT-4's outputs are not easily tweakable (short of changing prompt or waiting for model improvement), some organizations might opt to fine-tune an open-source LLM on the task – but given GPT-4's superior ability currently, prompt iteration is usually enough.

Alternate Commercial Variant: We could consider an **ABBYY-only** high-accuracy scenario too: that would be ABBYY FlexiCapture with a human verification step. That is, accept ABBYY's results into a verification UI for a human to quickly glance and correct if needed. This ensures near 100% accuracy. However, it introduces a manual step and doesn't leverage LLMs. Since the question explicitly asks for usage of "strong LLM services" in one variant, the Azure+GPT approach was described as it aligns with that. If one did have ABBYY already, it might actually reduce the need for GPT – ABBYY might catch everything. But one could still use an LLM to auto-check ABBYY's output. In any case, the described hybrid approach shows how to get top quality with relatively lower dev effort by using AI services collaboratively.

Checklist for Practical POC

To implement a proof-of-concept for TTN data extraction in 3–5 days, follow these steps:

1. **Gather a Representative Document Set:** Collect examples of TTNs covering the variability you expect in production. Aim for at least 20–30 samples that include:
2. **Digital PDFs** exported from systems (e.g. 1C) – these will test how the pipeline handles perfect input (and ensure we don't unnecessarily OCR text that could be extracted).
3. **Scanned paper TTNs** – including a mix of good quality scans and some poor quality (slight blur, skew, low contrast, etc.).
4. **Photos** if applicable (TTNs photographed by phone).
5. Documents with **stamps, signatures, or handwritten marks** (to see how they impact results).
6. If possible, different form layouts: e.g., older vs newer TTN forms, or similar docs like УПД (Universal Transfer Document) or TORG-12 forms, to test generality.

7. Ground truth: For each sample, manually prepare the expected JSON (or at least note down key fields and line items) so you can later measure accuracy.

8. **Set Up OCR Tools for Testing:**

9. Install PaddleOCR and its dependencies (or an alternative open-source OCR like Tesseract for a baseline). Make sure Russian language is enabled (for Tesseract, load `rus` language pack; for Paddle, use the multilingual model).
10. Sign up for cloud OCR trials: Azure Form Recognizer (it offers free pages in the free tier) and Google Document AI (Google may have a trial or you can use a small number of pages free). Also, if possible, get a trial of ABBYY – ABBYY Vantage has a trial where you can use their invoice skill, or ABBYY FineReader Engine trial (requires contacting sales, which might be slow for a 5-day POC, so focus on easily accessible ones).

11. Access to GPT-4: Ensure you have API access (OpenAI API with GPT-4 availability, or Azure OpenAI if you go that route). If not, plan the POC without it or use GPT-3.5 as a placeholder to gauge how an LLM might perform, knowing GPT-4 will do better when you have access.

12. **Initial Tests with Each Solution:**

13. Run a **quick OCR scan** on 2–3 documents with each tool:

    - PaddleOCR CLI or script: get text and see if it identifies table structure.
    - Azure Form Recognizer: use their sample tool or Postman to send a file to the invoice API. Inspect the JSON result – see which fields are filled, how line items come through [32].
    - Google: use the online demo in Document AI Workbench with the form parser or OCR. Check if it got the table and key fields.
    - If you have an ABBYY trial (for example, Vantage's invoice skill), try a TTN through it (maybe via their UI) to see results, or use ABBYY FineReader PDF (the desktop software) just to OCR and see text results as an indication of quality.
    - Note the **ease of integration** as you do this – e.g., Azure's output is already nicely JSON with line items, while Paddle gives you raw results requiring coding. This will factor into effort.

14. **Compare Quality Outputs:** For a few documents, compare the extracted data from each:

15. Do all line items appear correctly? (e.g., count the number of line items each tool found and compare to actual).
16. Are numeric fields (quantities, prices) correct and not confused? Check if any tool mixed up, say, quantity and weight columns.
17. How do they handle Russian text – any obvious misreads (wrong characters or gibberish)?
18. Do they capture the shipper/consignee names accurately?

19. This qualitative check will reveal if, for instance, Tesseract is way behind (likely it will have more errors in layout) or if Paddle is nearly as good as Azure on these samples, etc.

20. **Select the Final Stack:** Based on the quick tests:

21. If an open-source solution like PaddleOCR was nearly as accurate as Azure and you have the dev capability, you might choose the open-source route for POC (to avoid dependency on cloud). Otherwise, leveraging Azure's ready-made output might let you focus on other parts.
22. Decide if GPT-4 is needed in the POC: If, say, Azure or Paddle alone already got 95% of fields, you might skip LLM for now in POC due to time. But if you notice certain things missing (like vehicle info), you might plan a step to use GPT-4 or a regex to grab that from full text.

23. Also consider time: implementing PaddleOCR parsing might take longer than integrating an API. In a 3-day effort, using Azure's output might get you to a demo faster. You can note the trade-off in the report.

24. **Implement the Pipeline (Coding):**

25. Write a small Python script or application that does the entire process on one document (then you'll extend to batch):
    - Accept a document (PDF/image).
    - If PDF, decide: if it's text-based (you can quickly check if PyMuPDF can extract text), then parse text directly instead of OCR – that would be a huge win for digital PDFs (100% accuracy for text extraction). If image or scanned PDF, proceed to OCR.
    - Call the chosen OCR service or library to get structured results.

- If using open OCR, implement the table parsing and field extraction as described in the architecture. If using Azure, translate its JSON to your JSON schema (rename fields, etc.). If using GPT-4, call it with either the image or text depending on plan.
- Return or print the final JSON.

26. Run this on a few files and manually verify the JSON against expected. Tweak as necessary.

27. **Measure and Record Performance:** Take your set of test TTNs:

28. Run them through the pipeline.
29. Calculate accuracy metrics: e.g., create a simple scoring where you compare each field in the JSON to the ground truth (you might do this manually for POC if ground truth is not in a file). Note if any line items are missing or extra.
30. Compute an approximate field accuracy % and line item recall %. If you have time, do a brief error analysis (e.g., "2 out of 30 documents had errors: one missed a line with a handwritten quantity, another misread a company name due to stamp overlap").

31. Also note processing speed per doc and any significant differences between methods (maybe Paddle took 4s vs Azure 2s, etc., at small scale it's fine).

32. **Evaluate Cost:** For each approach, extrapolate the cost:

33. For open-source, cost = essentially zero (just server cost).
34. For Azure/Google, use their pricing to say "if 1000 TTNs, this approach costs ~$X". For GPT, estimate tokens and cost as done above. This POC phase likely used free tier or trial credits, but for real use you want these numbers.

35. This will be important for recommending which to use. Perhaps you find that Azure's accuracy was only slightly better than Paddle's, but costs a lot – you'd mention that in final choice.

36. **Decide on Final Approach:** Based on POC results, decide which solution (or combination) is optimal for the real deployment. Maybe you find: "Azure Form Recognizer alone gave 95% accuracy. With a bit of post-processing or a few-shot GPT prompt, we can bump it to 98%. That might be sufficient, so we'll use Azure + small GPT usage." Or you might find Paddle got you to 90% but to reach 98% would require a lot more dev or an LLM anyway – then maybe a commercial engine is justified. The POC's purpose is to reveal these practical trade-offs.

37. **Document Next Steps:** Conclude the POC with what additional work is needed to productionize:

    - Do you need to label more data and train a custom model (Azure or ABBYY) to improve something? (e.g., custom model for TTN fields if invoice model wasn't ideal).
    - Can you handle edge cases like very bad scans by perhaps routing them to a human or a second pass? (Maybe part of a plan: if confidence < 80% on some fields, mark for human verification).
    - Note integration steps with actual systems (like inserting JSON into a database or ERP) but that's beyond extraction itself.

By following this checklist, in ~3-5 days you should have: - A working prototype that takes TTNs and outputs JSON. - Measured accuracy and a sense of whether it meets requirements or if further tuning is needed. - A decision point on using open-source vs cloud vs hybrid in the final solution, backed by empirical data. - A clearer idea of budget (if any external APIs) and how to improve over time (which could involve gradually incorporating LLM or training custom OCR models, etc., as discussed).

Through this deep exploration, we've identified leading solutions and mapped out how to leverage them for automating TTN data extraction. The best choice often depends on the specific context – volume of documents, sensitivity of data, budget, and internal AI expertise. Classic OCR systems like ABBYY remain hard to beat for Cyrillic-heavy, structured forms when maximum accuracy is needed. Cloud OCR/IDP services offer fast setup and good quality, now supporting Russian layouts quite well. Vision+LLM approaches represent the future, already delivering good results, and can be combined with traditional OCR to cover gaps. A hybrid strategy – employing the right tool for each aspect – is thus a prudent approach for building a robust TTN processing system. With a successful POC in hand, the organization can move forward confident in the chosen technology stack, knowing the strengths and limitations and how to address them.

---

1 6 10 12 14 15 16 22 24 27 29 36 37 Comparing the Top 6 OCR (Optical Character Recognition) Models/Systems in 2025 - MarkTechPost
https://www.marktechpost.com/2025/11/02/comparing-the-top-6-ocr-optical-character-recognition-models-systems-in-2025/

2 4 13 19 23 31 32 44 Who's the King of Invoice Recognition? A Full Comparison of 5 Leading Cloud-Based AI Platforms | Yeeflow Blog
https://blog.yeeflow.com/post/whos-the-king-of-invoice-recognition-a-full-comparison-of-5-leading-cloud-based-ai-platforms

3 11 Language and locale support for prebuilt models - Document Intelligence - Foundry Tools | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/language-support/prebuilt?view=doc-intel-4.0.0

5 30 Using Azure OpenAI GPT-4o to extract structured JSON data from PDF documents - Code Samples | Microsoft Learn
https://learn.microsoft.com/en-us/samples/azure-samples/azure-openai-gpt-4-vision-pdf-extraction-sample/using-azure-openai-gpt-4o-to-extract-structured-json-data-from-pdf-documents/

7 8 9 ABBYY Flexicapture vs AZURE Form Recognizer
https://www.linkedin.com/pulse/abbyy-flexicapture-vs-azure-form-recognizer-parul-yadav

17 Azure AI Document Intelligence and Power Platform AI Builder: When To Use What
https://lanternstudios.com/insights/blog/azure-ai-document-intelligence-and-power-apps-ai-builder-when-to-use-what/

18 Azure Document Intelligence Pricing for AI
https://www.byteplus.com/en/topic/414752

20 21 33 34 35 Processor list | Document AI | Google Cloud Documentation
https://docs.cloud.google.com/document-ai/docs/processors-list

25 26 28 38 39 GitHub - PaddlePaddle/PaddleOCR: Turn any PDF or image document into structured data for your AI. A powerful, lightweight OCR toolkit that bridges the gap between images/PDFs and LLMs. Supports 100+ languages.
https://github.com/PaddlePaddle/PaddleOCR

40 Extract Table Data from Documents with OCR and Computer Vision
https://medium.com/@sudhanshu.dpandey/revolutionising-table-extraction-simplifying-document-processing-39b6eb2db05c

41 42 Extracting Invoice Data Smarter with GPT Vision | by Zein Ismalingga | Medium
https://medium.com/@ismalinggazein/extracting-invoice-data-smarter-with-gpt-vision-b51880325395

43 I wanted to extract information from invoice using GPT-4o, which can ...
https://community.openai.com/t/i-wanted-to-extract-information-from-invoice-using-gpt-4o-which-can-be-image-or-pdf/932265