ChatGPT

Azure-Based Production Architecture for Loan Document Analyzer

Overview and Key Requirements

To build a **production-grade loan document scanner and analyzer on Azure**, we will leverage Azure's fully managed services for **serverless scalability**, data processing, and AI. The solution targets **enterprise-grade scalability with auto-scaling**, a **100% serverless architecture** (no VM or self-managed servers), and **support for all Indian languages** (with a focus on Telugu and Hindi). By using Azure's India regions (e.g. South India datacenter in Chennai) for deployment, we ensure low latency for Indian users and compliance with data residency requirements. Azure's rich AI services (OCR, translation, speech, and OpenAI) are well-suited to handle Indian language content and are cost-effective compared to alternatives [1] [2]. Below is a breakdown of the architecture components and how they meet the requirements:

Serverless and Scalable Components on Azure

- **Azure Functions (Consumption Plan)** – The backend logic runs on Azure Functions, which automatically **scales out** to handle spikes in load without manual intervention. Azure Functions are event-driven and **fully serverless**, meaning you pay only per execution. With the consumption plan, functions can scale to meet high concurrency and then scale back to zero when idle (minimizing cost). This ensures enterprise-grade scalability and resilience by default. We can use HTTP-triggered functions for APIs and blob-triggered functions for processing files as they are uploaded.
- **Azure Blob Storage** – User-uploaded loan documents (images or PDFs) are stored in Blob Storage. This service is serverless, highly durable, and auto-scales for virtually unlimited storage. We can configure an Event Grid trigger so that whenever a new document blob is uploaded, it automatically invokes an Azure Function to process that document. This event-driven pattern allows parallel processing of many documents.
- **Azure Cosmos DB (Serverless mode or Auto-scale)** – Extracted loan data and analysis results are stored in Cosmos DB, a fully managed NoSQL database with global distribution capabilities. Cosmos DB in serverless mode or with auto-scale enabled will **scale throughput** based on usage, so it can handle growth from 100 users to 100k users seamlessly. It also provides <10ms low latency reads/writes, and data is automatically backed up and encrypted. This is used to store structured information like loan terms, EMI schedules, user profiles, etc., in JSON format.
- **Azure Cognitive Services** – Azure's AI services (part of Azure AI Foundry) will handle OCR, translation, and text-to-speech without us managing any servers. These services are inherently scalable as Microsoft manages the underlying infrastructure. Key services include:
- *Document Intelligence (Form Recognizer)* – for OCR and form/table extraction.
- *Translator* – for language translation.
- *Speech Service (Text-to-Speech)* – for generating voice outputs.
- *Azure OpenAI Service* – for running GPT-4/GPT-3.5 models to generate explanations.
- **Azure Logic Apps / Durable Functions** – For orchestrating multi-step workflows (if needed), we can use Durable Functions (an extension of Azure Functions) or Logic Apps. For example, processing a

document might involve multiple steps (OCR → parse → compute → LLM explain → TTS). A Durable Function can manage this sequence with retries and checkpoints, all in a serverless manner. This ensures reliability for longer-running processes without manual threading or state management.

- **Azure Frontend** – The user-facing frontend could be a mobile app (Flutter as mentioned) or a web app. If it's a web app, Azure Static Web Apps or an App Service (with a free Linux plan or container) can host it serverlessly. Static Web Apps also provide built-in CDN and authentication integration. For a mobile app, the app will communicate with the Azure Functions via HTTPS API calls.

Why Azure? – Azure's serverless services fulfill the "no infrastructure" mandate while providing auto-scale and high availability by default. Additionally, Azure offers specific advantages for the India-focused requirements: - Full support for **Indian languages** across services (OCR, translation, TTS) – e.g. Azure Speech supports **12 Indian languages** for text-to-speech (31 voices, including Hindi, Tamil, Telugu, etc.) [3], whereas some competitors support far fewer. - **Cost efficiency** – Azure's pricing for document processing is highly competitive. For instance, Azure's basic OCR and layout extraction costs about **$1.50 per 1,000 pages** [4], and even full structured document analysis with tables is ~$10 per 1,000 pages [2]. This is significantly cheaper than Google's Document AI Form Parser (~$30/1,000 pages) or AWS Textract's table extraction ($15/1,000 pages) [1]. Similarly, Azure Translator costs ~$10 per million characters (half the price of Google's API at $20) [5]. These savings are crucial for a startup on a tight budget.

Document Ingestion and OCR Pipeline

1. Document Capture (Client Side): The process begins on the client – the user uses a mobile app (or web UI) to scan or select their loan document. The app should integrate an on-device scanner for convenience (e.g., using device camera with auto-crop, edge detection). However, once the image/PDF is ready, it will be uploaded to the cloud for analysis. The upload goes to Azure Blob Storage (e.g., to a specific container for "incoming documents"). We enable an **Event Grid trigger** so that each new file triggers the pipeline automatically. This design is fully serverless and decouples the upload from processing. Optionally, a small **HTTP Function** can be used instead, where the app directly calls an endpoint to submit the document (which could internally put the file in Blob or process it directly).

2. OCR with Azure Document Intelligence: When a new document arrives, an **Azure Function (OCR Processor)** is invoked (via the blob trigger or an HTTP call). This function uses **Azure's Form Recognizer (Document Intelligence)** service to perform OCR and layout analysis on the document. We will use the latest **Read/Layout model** which can extract all text lines and **preserve table structures** from the loan document. Crucially, Azure's OCR engine supports mixed English and regional language content out-of-the-box. For example, if parts of the document are in Hindi or Telugu scripts, the OCR will still extract that text accurately (the underlying model can detect and process **all major Indic scripts** without needing a language code in most cases [6] [7]). The OCR function will call the Azure Cognitive Services Read/Analyze API and get back a JSON with extracted text, lines, words, and table data (cells, rows, columns coordinates).

- We choose Azure's cloud OCR over others because of its combination of **accuracy and cost** for Indian documents. Azure's service covers **Hindi, Telugu, Tamil, Kannada, Malayalam, Gujarati, Bengali, Punjabi, Odia, etc.** and is optimized for multi-language documents. (For reference, Azure's speech services cover 12 Indian languages [3], and the vision OCR supports a similarly broad range.) Moreover, **table extraction is included** in the Document model pricing. In contrast, Google's OCR needs a separate (and pricey) Document AI for tables, and AWS Textract charges 10× more for table extraction [1].

- **Performance:** Azure Form Recognizer can process pages asynchronously; for large documents (hundreds of pages), the function might use the async “Begin analyze” call and periodically check the result or use Durable Functions to wait for completion. For typical loan documents (5-20 pages), synchronous calls may suffice (each page processed in a couple of seconds). The function runtime (Node, Python, or .NET) should use reliable SDKs (e.g. Azure.AI.FormRecognizer SDK) to handle this easily.

- **Fallbacks:** In rare cases of very low-quality scans, Azure’s OCR might struggle. As a backup, we could integrate an alternate OCR engine (for example, an open-source **Tesseract** or **PaddleOCR** running in another Azure Function or container instance). However, maintaining an alternate OCR engine in Azure would add complexity (Tesseract could run in an Azure Container Instance on demand – still essentially serverless). In practice, Azure’s OCR plus some image pre-processing (we can do basic cleanup via the Python Pillow or OpenCV in the function) should handle most cases. The mobile app can also ensure the image is reasonably clear (by guiding user to capture properly).

3. Data Extraction & Parsing: After OCR, we have raw text and possibly structured layout data. The next step is to extract the specific loan information from this text. We implement this in an **Azure Function (Data Parser)** which can be part of the same function that did OCR (for simplicity) or a separate one in a chain. The extraction logic will use a **hybrid approach**: - For known document formats (the top banks like SBI, HDFC, ICICI, etc.), we maintain a set of **template-specific regex patterns** or rules to locate key fields (loan amount, interest rate, tenure, EMI, etc.). For example, if the OCR text contains a line like “Sanctioned Amount: ₹ 5,00,000”, our regex can capture that number. We’ll build these patterns by studying sample loan documents from these banks. Having these templates covers ~80% of cases with very high precision. - If the document doesn’t match any known template (e.g., a smaller bank or an unusual format), we use an **AI-based extraction fallback**. Here we can leverage **Azure OpenAI** to extract fields. For instance, we can prompt a GPT-4 model with: *“Extract the following fields from the loan document text: {text} … output a JSON with principal, interest rate, tenure, EMI, etc.”*. The OpenAI API (through Azure) can parse unstructured text intelligently. To keep this reliable, we will supply hints (like regex findings as partial context) and ask the model *not* to guess if unsure. Because this is a sensitive step (we can’t afford hallucinations here), we might restrict the model to only identify and quote exact numbers from the text (not make up anything). This step will be used sparingly – only when regex templates fail – to minimize cost and avoid errors. - **Validation:** After extraction, the system validates the key numbers. A crucial check is recalculating the EMI from the extracted principal, rate, and tenure using the standard formula and comparing it to the extracted EMI. If the computed EMI doesn’t match the document’s stated EMI (within a tiny tolerance), we flag a discrepancy. This catches OCR or parsing errors (or document inconsistencies). If flagged, we can either alert the user to double-check or fall back to another method (for example, try a different OCR engine or manually highlight the numbers for user confirmation).

All these parsing operations happen within Azure Functions, which can scale out if many documents are processed concurrently. The stateless nature of functions (each invocation independent) fits well here. **State** (the extracted data) is stored in a database (next step) rather than in memory, so multiple function instances can work in parallel.

4. Storage of Extracted Data: Once we have structured data (loan details), we save it in **Azure Cosmos DB**. Each loan record can be a JSON document containing fields like `loanId`, `userId`, `bankName`, `principal`, `interestRate`, `tenure`, `emi`, etc., along with perhaps the full amortization schedule or we might store that separately. Cosmos DB offers **serverless** or auto-scaling throughput – initially, costs will be minimal (there's even a free tier for 1000 RU/s). As user count grows, Cosmos can transparently scale to handle it. We enable features like **Autoscale** (which adjusts RU/s between a floor and ceiling based on usage) to ensure performance under load without over-provisioning. All data is **encrypted at rest** automatically, and can be partitioned by `userId` for efficient querying (so each user's data is grouped).

- We also store any user-specific preferences (like language choice, notification preferences) in Cosmos or Azure Table Storage. But Cosmos can serve as a one-stop database for both user profiles and loan records.
- **Deletion and security:** If a user requests data deletion, a function can delete their items from Cosmos (and any blobs). Cosmos DB's **point-in-time restore** and backup features ensure we can recover from accidental deletions if needed, but we will implement a proper *"Delete My Data"* flow to comply with privacy laws (discussed later).

5. Loan Calculations Engine: A core part of the analyzer is computing things like the amortization schedule, interest/principal breakdown over time, impact of prepayments, balance after certain years, etc. This **does not necessarily require any cloud service** – it's pure business logic that can run either on the client (app) or on the server side. In our Azure architecture, we have two options: - **Client-side calculation:** We could offload this to the mobile app (since once the loan terms are known, calculating EMI or interest is straightforward). This would save server resources. The app (written in Flutter/Dart) could use a library or custom code to calculate the EMI and schedules using decimal math. The earlier plan suggested using high precision decimals to avoid floating-point errors – Dart's `decimal` package or JavaScript's bignumber libraries can be used if on client. - **Server-side calculation:** Alternatively, we implement this in an Azure Function (perhaps the same one that handles parsing can also compute additional results). For example, after extracting data, the function computes the EMI (to double-check and to present to user if not in doc), total interest over life, year-by-year breakdown, etc. It can also simulate **prepayment scenarios** – e.g., "if user pays extra ₹X per month or a lump sum of ₹Y after 2 years, how many months does it cut from the tenure and how much interest is saved?" – using the formulas provided. Because this calculation might be requested on-demand by users (for example, user slides a slider for prepayment amount and sees updated graphs), having it on client provides instant feedback. But to adhere to "serverless cloud architecture" and for less capable devices, implementing it as a cloud function call is also fine. We should ensure to use a high-precision math library on the server (C# decimal type if .NET, or Python's `decimal.Decimal`) so that results are accurate. These computations are very fast (a few milliseconds), so cost impact on the function is negligible.

6. Summary of OCR Pipeline:
In summary, the ingestion pipeline on Azure looks like:

- User uploads document → **Blob Storage** (South India region)
- **Event Grid** fires → **Azure Function "OCR & Extract"** runs:
- Calls Azure Form Recognizer to get text/tables.
- Parses key fields via regex/AI.
- Validates and stores results in Cosmos DB.
- (Optionally) triggers next steps like analysis and report generation.
- The function returns a success response to the app, and the user can then view the extracted data and analysis.

All of this is serverless. If 1000 users upload documents at once, Azure will spin up multiple function instances in parallel – processing many files concurrently. Blob Storage and Cosmos DB can handle the parallel load by design (Cosmos will auto-scale RU/s as needed, Blob can handle massive parallel uploads). There are no fixed servers that could become a bottleneck.

AI Explanation and Insights Engine

One of the standout features is the **AI-driven explanation** of loan terms and personalized insights (like how to pay off faster, refinance options, etc.). On Azure, we implement this with a combination of Azure OpenAI Service and Azure Cognitive Search for grounding:

- **Azure OpenAI Service (LLM)**: We will use Azure's hosted GPT models to generate explanations in natural language. Azure OpenAI provides access to models like GPT-4 and GPT-3.5-Turbo via an API, with the benefit of **Azure's enterprise security** (the data stays in Microsoft's cloud, and we can apply role-based access, private network if needed, etc.). For most user queries (like "Explain my loan's terms" or "How can I pay off faster?"), a GPT-3.5 Turbo model may suffice at a much lower cost, whereas more complex multi-step reasoning might use GPT-4. We can configure both and choose dynamically: e.g., default to GPT-3.5, but if the user asks something complex or if we detect the need for higher accuracy, route to GPT-4. The cost is pay-per-1000 tokens (Azure bills similarly to OpenAI's rates). We will **cache** common explanations to reduce calls – e.g., the explanation of "What is EMI?" or "difference between reducing balance and flat interest" can be generated once and stored. We could store these in Cosmos DB or Azure Cache for Redis to serve subsequent users instantly.

- **Retrieval-Augmented Generation (RAG) with Cognitive Search**: To **prevent hallucinations and ensure factual accuracy**, we integrate a knowledge base of trusted reference documents:

- We will collect RBI circulars (such as the RBI's guidelines on prepayment – e.g., "RBI Pre-payment Charges Directions, 2025"), bank terms and conditions, and relevant financial FAQs. These documents can be **uploaded to Azure Cognitive Search** with the **Vector Search** capability enabled. We'll use Azure Cognitive Search to index the content and generate embeddings (Azure can use OpenAI Embeddings for this, or we can provide our own).
- When a user asks something like "Can I be charged for prepaying my loan?", our system will first query Cognitive Search with the question (and perhaps some key terms like "prepayment penalty India") to retrieve relevant passages (for example, the RBI directive that says no charges on floating loans [2]). We then feed those passages into the prompt for the Azure OpenAI model, so it has the **ground truth** to base its answer on. This ensures the answer cites the actual rule (and we can even have the AI include a snippet or reference).

- Azure Cognitive Search is a fully managed service that can scale to handle many documents and queries. It also integrates well with Azure OpenAI via Azure's **Cognitive Search + OpenAI** solution (they often call it "Azure OpenAI on your data"). This essentially abstracts the RAG pattern – we could use the built-in approach where you ask Azure OpenAI a question and it will use the cognitive search index as context if configured. That could simplify development.

- **Combining with Business Logic**: We also ensure that any numerical advice or scenario (e.g., "How much will I save if I prepay ₹5 lakh in year 2?") is actually computed by our deterministic engine rather than guessed by the LLM. For example, the app can call a function to compute that scenario, and then supply the result to the GPT model to phrase it nicely. We never want the GPT to fabricate numbers. We can enforce this by design: the app or function does the math, and the LLM just explains the meaning in layman terms. Azure OpenAI supports features like function calling – we could use that to let the model request a calculation from our API if needed, but that might be overkill. A simpler approach is to pre-calculate likely scenarios and just fill them in.

- **Multi-language Responses**: The user might prefer to read/listen in Telugu or Hindi. Rather than prompting GPT to answer in Telugu directly (which it can do, but its financial knowledge might be stronger in English), we will **generate answers in English first** (leveraging the full power of the model's understanding), and then translate the final answer. This approach was recommended in the research and works well to ensure accuracy. Azure provides the **Translator Text API** which can reliably translate English to Telugu, Hindi, Tamil, or any of the 20+ official Indian languages. We will call Azure Translator for the model's output text to get the localized version. This translation step is fast and costs only $10 per million chars [5] (and we get 2M chars free per month on the free tier), so it barely impacts cost. The output can then be sent to the user (or to TTS for audio). By doing this, we ensure the explanation content is correct (coming from GPT's English knowledge and our grounding docs) and then accurately conveyed in the user's language via a specialized translation system.

- **Text-to-Speech (TTS) for Voice**: To truly make this app accessible, we use Azure Speech's **Neural Text-to-Speech** to convert the explanation text into natural speech. Azure's TTS supports languages including **Telugu and Hindi with realistic voices** [3]. For example, there are specific neural voices for Hindi (like "Aarti" or "Arjun" as mentioned in research) and for Telugu (e.g., a female and male Telugu voice). These voices can even handle some English mixed in (common for financial terms) thanks to code-mixing support. The Azure Speech service is serverless – we send it text and it returns an audio stream. We can do this on the fly whenever the user requests to listen to an explanation. The first 500k characters per month are free, and after that it's ~$16 per million characters for neural voices. This is quite affordable (one million chars is roughly 11-12 hours of spoken audio). We can cache the audio for common explanations: for instance, if many users will hear the same explanation of "what is an EMI", we can store that MP3 in Blob Storage and directly stream it next time, instead of synthesizing repeatedly.

- **Caching and Optimization**: To keep latency low, we will implement caching at multiple levels:

- **Result caching**: If the same user asks the same question twice, obviously return the stored answer. Even across users, many will ask similar things (the app can have a pre-defined list of explainers like "Loan Summary", "Prepayment Impact", etc., which we generate once per loan). These can be stored in Cosmos DB or an Azure Cache for Redis for quick retrieval.
- **Pre-compute common terms**: We know that certain financial terms or concepts will be asked about frequently. We can pre-generate multi-language explanations for, say, the top 50-100 concepts (EMI, interest rate, tenure, credit score, prepayment, balance transfer, tax rebate, etc.) using our pipeline and store those in a dictionary. The app can even ship with some of these answers offline for instant response.

- **Batch requests**: If the user has multiple loans, or multiple questions, we can parallelize calls to the LLM or combine them in one prompt to reduce overhead. Azure OpenAI allows up to certain token limits per request; we'll craft prompts efficiently to include only necessary info.

- **Safety and Accuracy Controls**: We will incorporate the safeguards mentioned: system prompts that instruct the model not to make up facts, an output check against known info (we could even use a smaller model or heuristic to verify the LLM's output doesn't contradict our knowledge base). Each answer will include a disclaimer to the user (we can have a one-line "AI-generated explanation, please verify with your bank for final confirmation" in the UI, as a best practice).

In Azure, all these AI components are fully managed. Azure OpenAI can scale to many requests (we might need to request a quota increase as our user base grows, but it's built to handle enterprise loads). Cognitive Search can handle large document sets and queries per second, and can be scaled by changing the service tier (or adding replicas/partitions) without downtime. The nice part is we avoid maintaining our own vector database or search server – less ops work for a solo developer.

Multi-Language Support (Telugu, Hindi, and more)

Supporting **all major Indian languages** is a priority, especially for users in South India (Telugu-speaking, etc.). The architecture ensures multilingual support at multiple levels:

- **User Interface Localization:** The app's UI should be available in Telugu, Hindi, and other languages as needed. For a Flutter app, we can use packages like `easy_localization` to maintain translation strings for all text in the app (buttons, labels, instructions). This is a one-time effort per language. We will start with at least English, Hindi, and Telugu. The user can choose their preferred language on first launch (or the app can detect device language and ask to confirm). This covers static UI elements.

- **Document Language Handling:** Loan documents themselves might contain regional language text (e.g., some banks include a Hindi section). Our OCR via Azure Form Recognizer automatically detects language and extracts text regardless of language [7]. We do not need separate OCR for each language – it's unified. So whether the interest rate is written in English or Hindi words, we'll get it. If any field is in a regional script (say an address in Telugu), it will be extracted. We might then translate those to English internally if needed for parsing (Azure has OCR output with language codes for lines, which we can feed to Translator if we need to interpret the content). However, most critical loan fields (numbers, etc.) are numeric or in English, so this might not be a big issue.

- **Output Translation:** As discussed, we generate explanations in English and then use **Azure Translator** to convert to the target language. Azure Translator supports **Telugu, Hindi, Tamil, Kannada, Malayalam, Marathi, Bengali, Gujarati, Odia, Punjabi, Urdu, etc.** basically all 22 official languages (Telugu and Hindi being explicitly supported). The translation quality for these languages is very high – on par with Google's, as noted by many benchmarks. We also have the option to use open-source IndicTrans models if needed, but since the user specifically asked for "anything in Azure better," we stick to Azure's service for simplicity and reliability. The translated text is then displayed to the user in the app UI.

- **Text-to-Speech Voice Output:** For Telugu and Hindi users who prefer listening, **Azure Neural TTS** is utilized. We will call the TTS API with the translated Telugu text (for example) to get an audio stream. We must specify the voice/language code, e.g. `te-IN` for Telugu (India) and choose a specific voice like "SrijaNeural" (if available) or for Hindi `hi-IN` "AditiNeural" etc. The voices are natural and can handle typical loan vocabulary. Since Azure TTS supports 12 Indian languages with many voices [3], users can even choose a male/female voice or a different style if we expose that option. The audio is then played in the app. If connectivity is an issue, the app might download certain common audio files for offline use, but that's an enhancement.

- **Regional Focus Configuration:** Deploying in the Azure South India region ensures lower latency for South India users (like Telugu speakers in AP/Telangana) – faster responses for translation and TTS since the services have regional endpoints. We can also enable **Azure Front Door or CDN** if we had static content to distribute (like if we host a web front, CDN could cache content globally and in India). But for dynamic content, simply being in-region is great. It's also possible to replicate services in Central India region (Pune) if we need multi-region redundancy within India.

By using Azure's multilingual capabilities, we avoid relying on third-party APIs that might not guarantee data residency or might have limitations. Azure's AI services will keep our data within the region if we configure them to (for example, we ensure our Cognitive Services resource is created in South India region, so processing happens there). This is important for privacy as well.

Enterprise-Grade Architecture Considerations

Beyond the core functionality, an enterprise-grade solution needs to address deployment, monitoring, security, and compliance:

- **Deployment & CI/CD:** The entire infrastructure can be defined as code (using Azure Bicep or ARM templates, or Terraform) to ensure reproducibility. We can structure the Azure resources as follows: a Resource Group containing Storage account, Function App, Cosmos DB account, Cognitive Services accounts (Form Recognizer, Translator, Speech, OpenAI), Cognitive Search service, Application Insights for monitoring, and maybe an API Management if needed. Using GitHub Actions or Azure DevOps Pipelines, we set up continuous integration to build and deploy our Functions and any other code. For example, every push to main triggers a workflow that deploys the latest function code (using Azure Functions deploy or containers if using custom container for function runtime). Similarly, we can automate running tests and doing security scans.

- **Application Insights & Monitoring:** We integrate **Azure Application Insights** with our Function App for real-time monitoring and logging. This will capture logs from functions (like each step of OCR, parsing, etc.), exceptions, performance metrics, and even custom events (e.g., we can log an event "DocumentProcessed" with properties like duration, pages count, etc.). We set up alerts – for instance, if any function error rate goes above 1% or if CPU/memory usage hits limits, we get notified. App Insights also provides an analytics query interface to investigate issues and usage patterns. This is crucial for enterprise reliability, as we need to detect problems early (like if OCR fails on a certain format, we'll see exceptions and can fix our parsing logic).

- **Azure API Management (APIM):** If we plan to expose some APIs to external partners or a frontend, wrapping the functions behind APIM can provide throttling, authentication, and monitoring. For our own mobile app, APIM is not strictly required (the app could call function URLs directly, possibly using function keys or Azure AD auth). But if we scale up and perhaps offer services to third parties (as mentioned, maybe B2B licensing), APIM would allow exposing certain endpoints with usage plans and API keys in a controlled way. APIM itself is not fully serverless (it's a managed service with a fixed cost), so early on we might skip it to keep costs low, but it's an enterprise best practice for API endpoints.

- **Authentication & Security:** For user authentication, since we want phone number OTP (common in India), we might integrate **Azure AD B2C** with custom policies for phone sign-in, or use a third-party OTP service. Azure AD B2C can allow social logins and email/password, but phone OTP requires either using Azure Communication Services (to send SMS) or some workaround. We could also use Firebase Auth just for the OTP if that's easier (though that's a Google service, it's standalone and many Indian apps use it). However, keeping everything in Azure, we might consider **Azure Communication Services SMS** for OTPs and then maintain our own simple verification backend. In any case, once users are authenticated, we secure their data access by scoping all database queries to their user ID (and using auth tokens to identify user calls). We store minimal personal data – maybe just a phone number or email as identifier and mapping to their records.

- **Secret Management:** All secrets (API keys for Cognitive Services, Cosmos DB keys, etc.) are stored in **Azure Key Vault**. Our functions will be set up with a managed identity that has access to the necessary secrets in Key Vault. This way, we avoid putting any keys in config files or code. This is important for enterprise security – keys can be rotated easily and they're not exposed. Azure Functions can directly reference Key Vault secrets in application settings using a special syntax, making it convenient.

- **Compliance with DPDPA 2023:** As noted, India's Digital Personal Data Protection Act (effective Nov 2025) imposes several requirements:

- **Consent:** We will include a consent screen when the user first uses the scanning feature. This will explain what data is collected (loan details), how it's used (to analyze and give recommendations), and we must get explicit confirmation. The privacy policy and consent text will be provided in all major languages (Telugu, Hindi, etc., as needed) to ensure understanding. We log this consent (e.g., a timestamp in Cosmos DB under the user's profile). Azure's logging and App Insights can also keep an audit of when data was processed.
- **Data Minimization:** Our architecture already leans toward minimal data retention – for example, we do **not** need to store the actual image of the document long-term. We can process it and discard it (or store it short-term for say 24 hours in case re-processing is needed, then auto-delete via a Lifecycle Management rule on Blob Storage). We only keep the extracted structured data which is the minimum needed to fulfill the service.
- **Right to Erasure:** Implement a feature for the user to delete their account/data. This would trigger an Azure Function that deletes their records from Cosmos DB, removes any blobs (if stored), and also deletes any cached AI outputs related to them. We can use Cosmos DB's change feed or an Event Grid event to confirm deletion completion. We should also remove their consent record. Essentially, purge everything for that user. Azure's compliance is our friend here: after deletion, if backups exist, we might not have direct control (Cosmos keeps automatic backups for some time), but DPDPA is more about not *using* the data after deletion. We'll document that procedure in our policy.
- **Breach Notification:** Azure has security center and monitoring – if any unusual activity or breach is detected, we have to notify the Data Protection Board of India within 72 hours. We would set up alerting on any unauthorized access patterns. Also, using Azure's built-in security features (like Defender for Cloud) will help identify vulnerabilities or suspicious access.

- **Data Localization:** By deploying all services in India regions, personal data stays in India. Azure Forms, Cosmos, etc., in South India region means the data isn't leaving the country, aligning with any localization expectations in law or simply user trust. We should double-check that any call we make to external services (like if we had used a third-party) doesn't transfer data abroad. With this Azure-only architecture, we're safe on that front.

- **Performance and Auto-Scaling Configuration:** Enterprise scale means planning for high throughput. Azure Functions consumption plan can scale out to as many instances as needed (by default up to 200 instances, and each instance can handle a certain number of concurrent executions depending on memory/CPU). For heavy workloads (like if suddenly thousands of people scan at once), we might consider Azure Functions **premium plan** with auto-scale rules for more control, or Azure Container Apps for long-running OCR tasks. But initially, consumption plan should suffice. We will also use **Azure Service Bus or Storage Queues** if we want to decouple the front-end request from the heavy work. For example, the upload function could put a message in a queue for "process this document" and return immediately. Another function reads from the queue and does the OCR and processing. This is a common pattern to avoid timeouts and to level out spikes (queue will buffer requests if too many). Service Bus and Storage Queues are both serverless message brokers (Service Bus is more enterprise with retry, DLQ, etc.). This ensures even if load spikes, we don't drop tasks – they just queue up and functions scale to drain the queue.

- **Testing and Quality Assurance:** We will use Azure's features to support a robust dev/test pipeline. We can create separate **environments** (Resource Groups or even separate Azure subscriptions) for dev, test, and production. Each environment can have its own instance of the functions and DB (maybe smaller sizing). We'll run integration tests using sample documents (including Telugu/Hindi content) to verify the OCR and parsing yields correct results. Azure's Form Recognizer can mis-read some characters (like 5 vs S, etc.), so we'll test and refine our regex/LLM parsing accordingly. Using App Insights, we can log the confidence of OCR and maybe flag low-confidence cases for manual review during testing.

Diagram of the Architecture (Description)

(While a visual diagram is not shown here, the architecture can be visualized as follows:)

User App (Mobile/Web) → (uploads document) → **Azure Blob Storage** → (Event Grid trigger) → **Azure Function: OCR & Extract** → (calls Cognitive Service Form Recognizer) → **Loan Text & Data** → (Function parses data) → **Azure Cosmos DB** (store loan info) → **Azure Function: Analysis** (compute schedules, etc.) → **Azure OpenAI** (generate explanation) → **Azure Cognitive Search** (provide reference data to OpenAI) → **Azure Translator** (English to Telugu/Hindi) → **Azure Speech** (synthesize audio) → **User App** (receives structured data, text explanation, and audio).

All these components reside in Azure's cloud, with no persistent servers to manage. Azure handles scaling each component as needed: - Blob Storage and Event Grid can ingest huge numbers of files/events. - Azure Functions scale out compute instances under the hood. - Cosmos DB scales RUs and storage automatically. - Cognitive services scale to handle concurrent requests (with Microsoft's global infrastructure behind them). - Each Azure service has SLAs (usually 99.9% or higher uptime), ensuring reliability.

Cost and Scalability Estimates

This Azure architecture is designed to start small (low cost for a few users) and seamlessly scale to millions of users: - In initial development or pilot (say <100 users), the costs are minimal. Many Azure services have **free tiers**: Form Recognizer offers 500 pages/month free, Translator 2M chars, Speech 500k chars, Functions a million executions, etc. Our usage might mostly fall in free limits. Even paid, e.g., 1000 pages OCR is ~$1.50 [4] , 100k chars TTS is <$2. So within a budget of ₹1–2 lakh/year (~$1500-$2500), we can run this easily, especially if we leverage the free credits (Azure for Startups offers credits that could cover a lot of this early usage). - As we grow to thousands of users and beyond, costs scale linearly with usage, but efficiently. For example, **per document scan** cost might be around $0.005–0.01 (including OCR, a bit of AI, etc.). If we have 100k scans, that's maybe $500 – which is nothing compared to the value delivered. We will continuously monitor costs via Azure Cost Management; and we can set budgets/alerts to avoid surprises. - Auto-scaling means we don't have to pre-provision large servers for fear of spikes – Azure handles it. We just have to be mindful of any service limits (for instance, Cognitive Services have rate limits per resource; if needed, we can request quota increases or set up multiple resources and distribute load). - If the user base hits tens of thousands and usage patterns change, we can consider optimizations like **Azure CDN** to cache static parts of responses, or even deploying certain functions as **Azure Container Apps** on a KEDA-powered scale (if we need more control for specific heavy workloads like an OCR on GPU – Azure offers GPU VM or container options, but that's not serverless unless we use Azure Batch maybe).

Improving with Azure-Exclusive Features ("Anything better in Azure?")

Since the question asks if there's anything in Azure that does it better, here are a few Azure-specific enhancements we can leverage: - **Form Recognizer Custom Models:** Over time, we can train a **custom model** in Azure Form Recognizer for Indian loan documents. Azure allows custom training (either template-based or neural) using our labeled examples. For example, we could feed 50 sample loan sanction letters from different banks with labels for "LoanAmount", "InterestRate", etc., and train a model. This custom model could then extract those fields in one shot, potentially eliminating the need for our regex/LLM parsing for those formats. The cost for custom model inference is higher (about $30 per 1k pages as per Azure pricing [8]), so we might use it selectively. But it could boost accuracy and speed. Azure's AI Studio (Document Intelligence Studio) provides an interface to do this easily. This is something unique to Azure's platform that could "do it better" once we have data to train. - **Azure Cognitive Services Speech (Speech-to-Text)**: We primarily focus on TTS for output, but Azure also has Speech-to-Text covering 13 Indian languages [3] . We could in the future allow users to ask questions by voice (in say Telugu or Hindi) – the speech-to-text service will transcribe it to text, which we then process similar to typed input. This could be a differentiator for illiterate users or those who prefer speaking to an assistant. Because Azure STT is also serverless and supports those languages, it fits well in our stack. - **Azure OpenAI Fine-tuning or System Prompts**: Azure allows setting default system prompts at the service level for OpenAI. We can craft a system prompt that is tailored to financial advice context and set it for our OpenAI deployment. This can help consistently format answers (e.g., always include a line "This is not financial advice" or always use simple language). Additionally, if we find GPT-3.5's knowledge lacking in some Indian-specific finance topics, we could fine-tune a model or provide a few-shot examples in the prompt. Azure supports fine-tuning on some models (not GPT-4 yet, but 3.5 yes). This fine-tuning could incorporate Indian loan terminology and even some bilingual examples, so the model better handles code-mixed inputs (e.g., a user question that is partly Hindi, partly English). - **Integration with Microsoft Ecosystem**: We could leverage other Azure or Microsoft

365 services if needed. For example, if we ever integrate with Outlook or Excel (some advisors might want to export data), Azure's Graph API and Logic Apps connectors could help. Also, Power BI Embedded could be an option for richer visualizations if we wanted to embed interactive charts (though our Flutter charts likely suffice). Azure's focus on enterprise means down the line we can plug this solution into larger systems (like maybe an NBFC's loan management system) using Azure AD and APIs securely.

Conclusion

By using an Azure-centric architecture, we achieve an **auto-scaling, serverless solution** where all major components are managed services. This dramatically reduces the DevOps burden (especially valuable for a solo developer) while meeting performance and localization needs. Azure's strong support for **Indian languages** (OCR, translation, TTS) [3] and the cost advantages for key features (document processing, etc. at a fraction of competitors' pricing [1] [2]) make it an ideal choice for this product. The app will be able to **scan any Indian loan document, understand it, and explain it in Telugu, Hindi, or any preferred language** to the user with voice support – all powered by Azure behind the scenes.

Most importantly, this architecture is robust and **ready for growth**: it can start small (even free) and scale to millions of users without fundamental changes. Each layer is isolated and can be improved or expanded (e.g., adding more languages, more AI capabilities) without affecting the others thanks to the modular, event-driven design. It's a future-proof foundation for building India's first comprehensive loan document analyzer.

Sources: Azure Cognitive Services documentation and pricing (Form Recognizer, Translator, Speech) [4] [2] [5]; Azure Speech services coverage [3]; AWS Textract pricing for comparison [1].

---

[1] How to Use Amazon Textract for Easy Document Data Extraction and Automation
https://www.cloudoptimo.com/blog/how-to-use-amazon-textract-for-easy-document-data-extraction-and-automation/

[2] [4] [8] What is the cost of document intellegence service with form recognizer - Microsoft Q&A
https://learn.microsoft.com/en-us/answers/questions/5665475/what-is-the-cost-of-document-intellegence-service

[3] Unlocking Indian regional voices with Azure speech services.
https://www.linkedin.com/pulse/unlocking-indian-regional-voices-azure-speech-services-arvind-raman-5kdwc

[5] LLMs are 800x Cheaper for Translation than DeepL : r/LocalLLaMA
https://www.reddit.com/r/LocalLLaMA/comments/1jfh1d7/llms_are_800x_cheaper_for_translation_than_deepl/

[6] [7] Language and locale support for Read and Layout document analysis - Document Intelligence - Foundry Tools | Microsoft Learn
https://learn.microsoft.com/en-us/azure/ai-services/document-intelligence/language-support/ocr?view=doc-intel-4.0.0