ChatGPT

Technical Plan: AI-Powered Loan Document Scanner & Optimizer (Azure + Codex)

Project Structure

Monorepo vs. Polyrepo: We recommend a **monorepo** for this MVP to house both the Flutter client and Azure backend. A unified repository simplifies collaboration with GitHub Copilot/Codex, as all context lives in one codebase. It also ensures consistent naming and easier cross-references between front-end and back-end code. (A polyrepo could be used for strict separation, but a monorepo streamlines development for a small startup team.)

Folder Organization: Structure the repository with clear top-level directories for each component:

- `/frontend` – Flutter app project (e.g. with `lib/`, `assets/`, `test/` for Dart code and tests).
- `/backend` – Azure Functions and services (organized by function app or service type).
- `/backend/functions` – Azure Function App code, possibly further sub-divided by function types (HTTP APIs, blob triggers, orchestrators, etc.).
- `/backend/infra` – Infrastructure-as-code (Bicep templates, ARM JSON, or Terraform if used).
- `/shared` – (Optional) shared libraries or config (e.g. a constants file or API interface definitions that both client and server use).

Using clear naming conventions for projects and resources is key. For example, name the Function App project something like **LoanDocScannerFunctions** and the Flutter project **LoanDocScannerApp**. Within code, use descriptive names for functions (e.g. `ProcessLoanDocumentOrchestrator`, `ExtractLoanFieldsActivity`) and for Flutter components (e.g. `LoanDetailPage`, `EmiChartWidget`). This clarity helps Copilot maintain consistency – Copilot can even follow your naming patterns and architecture style if you keep them regular [1]. In a real-world case study, a team found that Copilot-generated code *"followed our architectural patterns, naming conventions, and logic flow like it had been on the team for months."* [1]

Flutter Module Structure: Inside the Flutter app, follow a standard modular structure. For example, under `lib/`, organize into subfolders like `models/` (data models such as Loan, PaymentSchedule), `services/` (for data syncing or API calls), `providers/` (if using Riverpod for state management), and `ui/` (screens, widgets). This separation keeps UI, state, and logic code clean. Codex can assist by generating boilerplate when it recognizes these folders. For instance, writing a comment like `// Loan model with JSON parsing` can prompt Copilot to create a Dart model class with from/to JSON methods automatically [2] [3]. Likewise, a comment such as `// State notifier for loan calculations` can yield a Riverpod (or ChangeNotifier) class skeleton – e.g. Copilot can produce a full provider class from a simple prompt [4].

Azure Function App Structure: In the Azure Functions project, organize function code by category: - **Orchestrators:** Durable Functions orchestrator workflows (each as its own function, triggered by the

orchestration trigger binding). - **Activities:** The discrete activities called by orchestrators (e.g. an activity to call the Document Intelligence API, an activity to insert into the database, etc.). - **Triggers/Endpoints:** HTTP trigger functions for client-facing APIs (e.g. `UploadLoanDocHttpTrigger` to receive file uploads or URLs), and event triggers like Blob storage triggers (for when a file is uploaded) or Queue triggers (if using a queue to kick off processing).

Keep related functions in the same Function App for simplicity, or logically separate into multiple function apps if needed (e.g. one for front-facing APIs, one for background processing). A single Function App with Durable Functions can host orchestrators and activities together for this scenario.

Use naming conventions in code and infrastructure that reflect their purpose (e.g. Azure storage container named `loandocs`, database named `loandb`, etc.). This not only helps humans but also Codex, which can use these names to infer correct usage in generated code.

Azure-First Architecture

Overview: The solution is designed cloud-native on Azure, leveraging serverless and AI services. The high-level flow is:

1. **Document Ingestion (Flutter App):** A user scans or selects a loan document (PDF or image) in the Flutter app. The app then uploads this document to Azure – either directly to an Azure Blob Storage container via a secured SAS URL or through an HTTPS call to an Azure Function that accepts the file. In our design, we'll use Blob Storage as the central document store.
2. **Trigger Orchestration:** Once the document is uploaded, a processing pipeline kicks off. This can be done via a **blob trigger** or a message in a queue. For example, when a blob is added to the `loandocs` container, a Blob Triggered Function starts a Durable Functions **orchestrator** workflow with the blob's URL. (Alternatively, the Flutter app could call an HTTP-triggered function to start the orchestrator and pass the blob URL or file bytes.)
3. **Durable Functions Orchestrator:** The orchestrator function coordinates the multi-step analysis of the document. Azure Durable Functions enable this serverless workflow to be stateful and resilient. As Microsoft notes, Durable Functions let us chain activities, wait for external events, and even handle human interaction if needed [5] [6]. Each loan document gets its own orchestration instance [6].
4. **Document Analysis Activity:** The first activity uses **Azure AI Document Intelligence** (formerly Form Recognizer) to analyze the loan document. We call the Document Intelligence *Analyze Document* API on the blob URL. This extracts text, tables, and key fields from the loan papers (e.g. borrower name, loan amount, interest rate, tenure, EMI, etc.) [7]. Document Intelligence is a cloud service that uses ML to understand documents [8] – perfect for parsing bank loan forms and PDF statements.
5. **Data Extraction & Validation:** Once the raw content is extracted, another activity function can post-process and validate it. For example, we can have Codex generate a **"LoanFieldValidator"** activity that checks extracted values (e.g. ensure interest rate % is within plausible range, EMI matches formula given principal and rate, etc.). Business rules (like "EMI must cover at least interest each period" or "no negative values") can be coded here. Codex can also help write template-specific extractors: if certain banks have fixed PDF layouts, we might have specialized parsing logic – Codex can generate regex or layout-based parsing code if we supply examples.

6. **AI Reasoning (OpenAI GPT-4):** Another activity can perform advanced AI analysis on the extracted data. For instance, using **Azure OpenAI (GPT-4)** to summarize the loan terms in simple language, or to flag any unusual clauses. This would involve constructing a prompt with the extracted text or key fields. We implement a **Retrieval-Augmented Generation (RAG)** approach: rather than prompt GPT-4 with the entire document (which might be long), we provide a summary of key fields and possibly a chunked version of terms. We also **ground the model's answers** by supplying reference data (like definitions of terms, or user's own loan parameters). Azure's guidance suggests using document content and metadata to ground NLP responses [9].
7. **Database Storage:** The pipeline then stores the processed results. We use **Azure Database for PostgreSQL Flexible Server** as our primary database (a managed Postgres service). One activity will save the loan's details and analysis results to Postgres. This includes structured data (principal, rate, etc.), the extracted text (for search/indexing), and any AI-generated summary or recommendations. We also store **embeddings** for semantic search: using the OpenAI Embeddings API, we convert key passages of the document into vector embeddings.
8. **Vector Indexing:** We have two choices for vector search – use Azure Cognitive Search with vector capabilities, or use Postgres + pgvector extension. In this plan, we favor **Postgres with pgvector** to reduce moving parts (all data stays in Postgres). We enable the `pgvector` extension (called `vector` in Azure) on our Postgres server to store and query embeddings [10]. After Document Intelligence extracts the text, an activity (or the same DB activity) will chunk the text into semantic passages (e.g. each section or paragraph of the loan doc), generate an embedding for each via Azure OpenAI, and upsert those into a `LoanDocumentEmbeddings` table (with columns for vector, reference to loan ID, etc.). Azure's documentation shows how pgvector allows efficient similarity search on embeddings [11] [12]. (If needed, we could also push these embeddings to an **Azure AI Cognitive Search** index instead, which natively supports vector search and could store the content with metadata [13]. Cognitive Search can serve as a RAG knowledge base [14]. Both approaches are valid; using Postgres keeps data in one place, while Cognitive Search might offer advanced text search and scaling.)
9. **Result Notification:** Finally, the orchestrator can signal completion. This might be as simple as updating a status field in the database or sending a push notification via a SignalR message to the app. The Flutter app can periodically poll an API or use a websocket to know when analysis is done and results are ready to present.
10. **Loan Optimization & Querying:** Once processed, the user can view insights about their loan in the app. They can also ask questions (natural language queries) like "How much interest will I save if I prepay ₹100k now?" or "Translate my loan terms to Hindi." When the user poses a question, the app calls an Azure Function (HTTP trigger) for **Q&A**. This function implements the RAG workflow:
11. It takes the question, uses Azure OpenAI Embedding to vectorize the query, and performs a similarity search either in Postgres (via pgvector cosine distance) or in the Cognitive Search index. This retrieves the most relevant chunks of the loan document or related knowledge (e.g. EMI rules) that can help answer the question.
12. It then crafts a prompt to GPT-4, inserting the retrieved context as reference text, and asks the question. The prompt might include a system message like "You are a loan assistant. Answer based on the provided document and data only, and if unsure, say so." This grounding reduces hallucinations.
13. GPT-4 returns an answer which the function relays to the app. We also attach any necessary disclaimers (e.g. "*This answer is AI-generated; please verify key details.*") as required.
14. **Multi-lingual support:** If the user's query or the document is in a language different from what GPT-4 handles best, we use **Azure Translator**. For example, if the user asks in Hindi, the function will

call Azure Translator to convert the question to English before the embedding and GPT steps, and then translate the answer back to Hindi for the user. Similarly, if the document text extracted is partly in a local language, we can translate it to English for uniform processing. Azure's Translator service can be invoked via its REST API or SDK within our functions. The translation step ensures language is not a barrier in understanding loan details.

15. **Text-to-Speech (TTS) Output:** The user may want to hear the analysis spoken aloud. For accessibility or convenience, we integrate **Azure Cognitive Services Speech** (TTS). The Flutter app can request an audio narration of the summary or answer. This could be done by an Azure Function that calls the Speech service (with the desired voice, e.g. an Indian English voice or Hindi voice) and returns an audio file/stream URL. Alternatively, for quick feedback, the app can use an on-device TTS for offline mode (more on offline strategy below). Using Azure's neural TTS when online will give a higher-quality, natural narration voice. This dual approach (cloud TTS vs local) provides the best experience online and functional fallback offline.

Production-Grade Azure Setup: We'll use Azure CLI and **Bicep** (Infrastructure-as-Code) to provision and configure all resources. The architecture will include: - **Resource Group:** e.g. `rg-loandocscanner-prod` in a chosen region. - **Virtual Network & Subnets:** A VNet (e.g. `vnet-loandoc`) hosting private endpoints for services. At least two subnets: one delegated for the Postgres flexible server, and one for other private endpoints (Storage, perhaps Cognitive services) and Function integration if needed. - The **PostgreSQL subnet** must be *delegated* to `Microsoft.DBforPostgreSQL/flexibleServers` [15]. This means only Postgres flexible servers use that subnet's IP range, ensuring isolation. Azure requires the VNet and the Postgres server to be in the same region [16]. We'll choose a subnet size that accommodates growth (minimum /28 for one server [17], but /26 or /24 if expecting many resources). - The **Function App** (if using a Premium plan with VNet integration) would either integrate into a subnet or use a regional VNet integration. We plan to use a **Premium (Elastic Premium) Plan** for Azure Functions to allow VNet access for outbound traffic (so the functions can call the database and other services via private endpoints). This plan will have the function app join the common services subnet. - **Azure Storage Account:** for blob storage of documents and for function app state (Durable Functions history and checkpoints are typically stored in Azure Storage queues/tables). This storage account will have a **private endpoint** in the VNet so that access from the Function or other Azure resources stays internal. We enable Secure Transfer Required and container-level access policies. The `loandocs` container may have an SAS policy for uploads from the app. - **Azure Database for PostgreSQL (Flexible Server):** configured in *VNet injection* (no public endpoint). The Bicep template will create it with the delegated subnet and set `publicNetworkAccess = Disabled`. We enable the `vector` extension (pgvector) by adding it to the server parameter allow-list and running `CREATE EXTENSION vector;` on the database [10] [18]. The DB will hold loan records, user info (if any), and embeddings. - **Azure AI Services:** - *Document Intelligence (Form Recognizer)* – create a Form Recognizer resource (likely in the same region if available, e.g. **Azure AI Document Intelligence** resource). We'll use the Prebuilt or Custom model APIs for forms. This service can have a **private endpoint** too, mapped to our VNet (so that calls from our functions to it don't go over public internet). - *Azure OpenAI (GPT-4)* – an Azure OpenAI resource with GPT-4 deployment. **Region constraint:** Azure OpenAI is only available in certain regions (as of now, e.g. East US, South Central US, West Europe, Southeast Asia, etc.). We must choose a region that supports GPT-4. If our primary region (say, "Central India") doesn't support it, we may deploy in a nearby region (e.g. Southeast Asia) and accept the data will be processed there. To maintain data compliance, we will enable *"no data logging"* on our OpenAI resource (Azure allows opting out of storing any prompts/results). In fact, Microsoft has stated that if customers opt-out of data collection, the models do not retain customer data, making it feasible to use GPT-4 in another region without violating data residency – *"if you opt out of data collection, the models are stateless. No data is stored overseas."* [19]. We will document this in our compliance notes (so stakeholders know that using GPT-4 via Azure still meets privacy guidelines

as no PII is stored by the service). All other services (Storage, Functions, Postgres, etc.) we keep in our primary region to minimize latency. (Alternatively, we could host everything in East US or another single region that supports all services, but if there are regulatory reasons to keep data in-country, the above approach is a compromise.) - *Translator and Speech (TTS)* – Azure Translator Text and Azure Speech services. These could be provisioned as part of a single **Cognitive Services multi-service account** (which provides a key and endpoint for various cognitive services) or as separate resources (there's a "Translator" resource and a "Speech" resource). We will likely create a Cognitive Services resource that supports both Translator and Speech in one key for simplicity. Both services can also be accessed via public endpoint (they don't yet support private endpoint as widely as storage/DB do), so our function will call them over HTTPS. We will secure the calls by restricting the key usage (no client directly gets the key; only functions use them, or use a managed identity with Cognitive Services if supported). - **Azure Functions (App Service Plan):** We create a Function App (running Node.js, Python, or .NET – any supported language where durable functions and Azure SDKs are available; C# or Python would be common choices). The function app will use a **Consumption plan for dev/test**, but for production with VNet, we use an **Elastic Premium plan**. In Bicep, we define the plan and function app, linking it to the storage account for internal use. We enable **Application Insights** for monitoring function execution and **Azure Key Vault** integration for loading secrets (see below). The function app's settings will include things like the Postgres connection string (or preferably, the credential pulled from Key Vault), the Form Recognizer endpoint/key, OpenAI endpoint/key, etc., but *we will not store secrets in plaintext.* Instead, use Key Vault references or managed identities. - **Private DNS Zones:** When using private endpoints, Azure will create private DNS mappings (like `loandocs.vaultcore.azure.net` to an internal IP for Key Vault, etc.). We will ensure DNS zones for each service (e.g. `privatelink.postgres.database.azure.com` for Postgres, `privatelink.openai.azure.com` for OpenAI, etc.) are set up and linked to the VNet so that our function can resolve the private addresses. - **Azure Key Vault:** All secrets (DB password, Cognitive service keys, etc.) go into a Key Vault. We give the Function App a managed identity and grant it **Key Vault Secrets User** or **Get** access on specific secrets. Our Bicep can create the Key Vault and populate initial secrets (for example, the Postgres admin password can be generated and stored as a secret during deployment). In GitHub Actions, we'll use Azure login with OIDC and then Azure CLI to upload any bootstrap secrets into Key Vault if needed [20] [21]. At runtime, the function app will fetch secrets via Key Vault references in its configuration (Azure allows setting an app setting like `DB_CONNSTR = @Microsoft.KeyVault(SecretUri=https://<vault>.vault.azure.net/secrets/<secretName>/<version>)` which the platform will resolve). This way, no secret values live in app settings or code. Integrating Key Vault centralizes secret management [22] [23] and reduces risk of leaks.

Network Security & Private Access: The solution is designed to minimize public exposure: - The database has no public IP – access it only from within the VNet (Functions via VNet integration, and optionally developers via VPN or Azure Data Studio over a private endpoint). - Storage account and Cognitive services use private endpoints – only callable from our Azure resources. (The Flutter app will upload documents via a secure SAS token URL, which is the only public surface if we allow direct blob upload. Alternatively, we could require the app to call an authenticated function to upload, thereby not exposing storage at all. For MVP, a time-limited SAS is acceptable.) - The function app itself **will have a public endpoint** by default (for HTTP triggers). We protect these endpoints by requiring authentication (using Function Keys or Azure AD JWT for user-specific APIs). For more security, we can put the function behind an **API Management** or **Azure Front Door** with a Web Application Firewall, but that might be overkill for MVP. We will at least enable HTTPS-only and, if possible, use Azure AD B2C or similar for user auth in the app, issuing tokens that the function checks (this ensures only genuine app users call the APIs). - We configure **CORS** on the function (or APIM) so that only our app's domain/origin can call it, preventing random internet usage of the endpoints.

Durable Function Patterns: We utilize Durable Functions for long-running and multi-step workflows. The orchestrator will employ **function chaining** (Document analysis -> validation -> storage) and possibly **fan-out/fan-in** if we want to parallelize some tasks (for example, analyzing multiple documents at once, or chunking pages for analysis). Durable Functions automatically checkpoint progress at each `await`/yield, so even if the function app restarts, it will not lose its place [24]. We'll also use the **monitor pattern** for any waiting/polling (though Form Recognizer calls usually provide results via polling internally, we could simply call the async API and then periodically check status via durable timer). Durable Functions allow flexible timeouts and recurring tasks; for instance, we can schedule periodic sync or cleanup using durable timers rather than fixed cron schedules [25] [26]. This will be handy for compliance (data deletion scheduling, discussed later).

By structuring the app around Azure services, we achieve a scalable, serverless architecture: - We can handle spikes in usage (Azure Functions scale out automatically on demand). - Large documents are processed reliably (the orchestrator can wait for the AI analysis which might take several seconds or more, without blocking a single short function execution). - All sensitive data remains within our Azure environment (meeting financial data security needs).

Codex Coding Roles

GitHub Copilot (Codex) will be a virtual team-member throughout development, accelerating both frontend and backend coding:

- **Azure Function Logic Generation:** We will leverage Codex to generate much of the function code. For example, writing a prompt like *"Create a Durable Functions orchestrator that takes a blob URL of a loan PDF, calls Form Recognizer to extract text, then saves results to PostgreSQL"* can yield surprisingly complete code. In one real test, a team provided a prompt for a durable orchestrator and Copilot produced a working orchestrator and activity functions in seconds [27] [28]. We can expect Codex to stub out our orchestrator (with calls like `context.callActivity("AnalyzeDoc", blobUrl)` etc.) and skeletons of the activity functions (`AnalyzeDoc`, `StoreResults`, etc.). It will also handle boilerplate like DurableOrchestrationContext usage, error handling patterns, etc., following Azure's best practices. We'll still review and refine the code, but this gives us a huge head start.
- *Blob Trigger & Binding Code:* We'll ask Copilot to implement a blob-triggered function (or an HTTP upload function). For blob trigger, a simple comment *"// Azure Function BlobTrigger: processes new blob in container 'loandocs'"* can prompt Copilot to write the function signature with the correct binding attributes and a function body that starts an orchestration instance (e.g. using DurableOrchestrationClient to start the orchestrator and passing the blob path). This saves us from digging through docs for binding syntax.
- *OpenAI & Cognitive Calls:* When coding the activity that calls Azure OpenAI or Document Intelligence, Copilot can help with the API usage. By referencing the Azure SDK or REST call format in the prompt, it can produce code to call the service. For example, *"// Use FormRecognizerClient to analyze document"* may lead Copilot to generate the needed code with correct API calls (especially if we have the Azure.AI.FormRecognizer NuGet or SDK imported in a C# function).
- *RAG Implementation:* We will have Codex assist in writing the retrieval-augmented generation logic. This includes:
    - Querying Postgres for nearest embeddings (Copilot can help write the SQL query using the `<->` operator for vector similarity if we hint at it). We might write a comment like "// Query pgvector for top 3 similar embeddings to the query vector" and get a code snippet using

Npgsql to execute `SELECT * FROM Embeddings WHERE embedding <-> @queryVec < 0.1 LIMIT 3;` (for example) [29].
    - Constructing the GPT prompt with system and user messages. Copilot is very good at completing text templates. We can start writing a multi-line string for the prompt with placeholders, and Copilot will fill in a reasonable prompt structure once it sees our intention.
    - Calling the OpenAI completion API via Azure SDK or REST. With an example in mind, Copilot will handle the JSON format for the request (model name, prompts, etc.) and parsing the response.
- *Validators & Calculators:* We have domain-specific logic like EMI validation and amortization calculations. These are well-defined algorithms, which Codex can implement from scratch if asked. For instance, *"// Compute EMI given principal P, annual rate R, months N"* can nudge Copilot to write the formula `emi = P * r * (1+r)^N / [(1+r)^N - 1]` (with `r` as monthly rate) [30]. We will use Copilot to build:
    - **EMI Calculator**: function that computes monthly payment or remaining balance. We'll verify the formula it gives (perhaps comparing against known formula sources). Copilot can also generate an amortization schedule table code easily (looping month by month, calculating interest and principal portions).
    - **Prepayment Optimizer**: more complex logic, but we can describe the desired behavior: "simulate loan payoff if extra ₹X is paid at month M" and let Codex produce code that adjusts the schedule. It might generate a loop that at month M subtracts X from remaining principal and recalculates subsequent EMIs or term.
    - **Extension/Recast Optimizer**: If extending the term to reduce EMI, Codex can implement: increase `N` until EMI falls under target, etc.
    - We will use TDD here: have Codex first generate **unit test scaffolds** for these calculators (e.g. given a small loan with known outcome, test that functions return expected values). Then we let it fill in the function code. This ensures the formulas align with expectations. (Copilot is capable of writing tests too – a prompt like *"// Test that EMI for 100k at 10% for 12 months is X"* will lead to a test case with assertions).

- **Amortization Engine**: The schedule of payments, interest, principal can be generated by Codex, as these formulas are standard [31] [30]. We can simply prompt "// generate amortization schedule list for given loan parameters" and get a function that returns a list of payments and remaining balance after each month. We'll double-check outputs, but this saves time.

- **Flutter UI Development:** Copilot will also dramatically speed up Flutter coding:

- *UI Code:* We can use Copilot to create Flutter widgets by describing them in comments. For example, writing a comment like `// Loan Summary screen with fields for amount, interest, EMI and a chart` will encourage Copilot to output a Column with Text widgets and perhaps a placeholder for a chart. It's known to even generate complex UI code from comments [32] [33]. We will iterate with Copilot to refine UI layouts. It can also suggest improvements if we write a basic structure.
- *State Management with Riverpod:* We plan to use Riverpod for managing app state (e.g. current loan details, list of loans, settings). Codex can produce provider classes and consumer widgets. For instance, a comment `// Riverpod provider for loan list with fetch and add functions` can result in a StateNotifier or a simple Provider class skeleton. Copilot has been shown

to auto-generate state classes from such prompts [34] [35] . We can instruct it to use Riverpod hooks (e.g. `ref.watch` etc.) if needed.

- *Chart Integration (fl_chart):* The app will visualize loan data (e.g. remaining balance over time). The **fl_chart** library is ideal for this, supporting line charts, bar charts, pie charts, etc. [36] . We will let Copilot write the boilerplate for a LineChart. By including a sample from fl_chart's docs or even just the class names (LineChartData, FlSpot, etc.) in a comment, Copilot can fill out a basic line chart configuration (like setting up axes, passing a list of `FlSpot` points for each month vs balance). We'll specify details (like "X-axis is month 0..N, Y-axis is balance in ₹"), and Copilot will do a lot of the heavy lifting – it often knows common usage patterns from open source. After generation, we'll adjust colors or labels, but it saves writing from scratch.
- *Voice Integration:* On the Flutter side, integrating an offline TTS (using say `flutter_tts` plugin) and possibly recording voice input could be done. Copilot can help write the integration code: e.g. for flutter_tts, writing `FlutterTts _tts = FlutterTts();` and some method names might prompt it to complete setup (like setting language, calling `_tts.speak(text)`). It won't know our exact UI flow, but given a hint, it will provide a decent starting point.
- *Form & Input Validation:* There will be forms for user input (maybe to input a custom loan or adjust prepayment). We can rely on Copilot to write form validation logic in Dart (for example, ensuring a number field is positive, or interest is between 0-100). These are straightforward if we describe them.

- *Testing:* Copilot will also assist in writing widget tests. A prompt in a test file such as `testWidgets('displays login button', ...` led Copilot to suggest a full test routine in an example [37] . We will use it to generate tests for critical UI flows (like adding a loan, seeing the analysis). It ensures we don't forget key assertions, and we can refine the tests from its output.

- **Shared Code & Config:** We can maintain certain constants or helper functions in a shared location (like interest rate conversion, date formats). Copilot can help produce those too (e.g. conversion from annual rate to monthly factor). It often picks up repetitive calculations and suggests making them a helper – effectively refactoring assistance.

Throughout all these tasks, we'll apply **Codex's output as a base and then refine**. It's important to review AI-generated code for correctness, especially financial calculations and security (we will add type checks, error handling where needed). But Codex significantly reduces grunt work and even offers creative solutions. As one team observed, Copilot "**parsed and reshaped data**", handled mapping and formatting logic, and even injected recommended best practices (like using HttpClient properly) on its own [28] [38] . We can expect similar when it works on our project – e.g. it may suggest using transactions when writing to the database or retries when calling external APIs.

By clearly **describing our intent in comments and docstrings**, we essentially "pair program" with Codex. We become the architect (outlining what needs to happen), and Codex translates a lot of that into code. This speeds up development of features like: - Durable function orchestrators and activities, - Data models and providers in Flutter, - Networking code (API clients, etc.), - UI layout and even styling (it might suggest some Material design conventions by default).

We'll maintain an iterative workflow: prompt Copilot to generate code, run/test it, adjust prompt or code, repeat. This approach will be used to implement everything from the **consent dialogs** to the **EMI calculators** – nothing is off-limits for a first draft from Copilot, which we then polish.

AI Workflow Design

The AI components (document understanding and Q&A) need careful design for prompt engineering, model selection, and user safety. We will design the workflows as follows, with Codex aiding in prompt and logic creation:

- **Document Understanding Prompting:** After extracting text via Document Intelligence, we might want GPT-4 to summarize or interpret the loan document. We will craft a prompt template for this, along lines of: *System message:* "You are a financial assistant analyzing a loan document." *User message:* "Summarize the key loan terms and identify any potential issues in the contract. Loan details: [extracted key fields]. Document text: [important sections of text]."

Codex can help us write this template. We'll have it generate a few variations and then we'll test which yields the best, most factual summaries. Because GPT-4 can sometimes **hallucinate** (especially if the prompt is not grounded), we ensure we always include actual data from the doc. We also instruct the model not to make up numbers. Additionally, we plan to include a disclaimer in the model's answer like: "*This summary is generated from the provided content; please verify against your original documents.*" We can enforce this by appending a note in the prompt or by post-processing the answer to add a disclaimer (the latter might be safer to guarantee it's there). Codex can assist by identifying all outputs from the model and wrapping them with the disclaimer text as needed.

- **Retrieval Augmented Generation (RAG):** For interactive Q&A, as described earlier, we use RAG. The design considerations:
- **Embeddings store:** We will pre-index each user's documents with embeddings in PG (and possibly have a general knowledge base too, see next point). Each embedding vector will be stored alongside the text chunk and metadata (e.g. page number or section). The RAG function will use *cosine similarity* search to fetch relevant chunks for a query.
- **Knowledge Base & Pre-caching:** We intend to **pre-load common knowledge** about loans into the system. For instance, definitions of terms like *"interest rate"*, *"prepayment penalty"*, *"amortization"*, or standard formulas could be stored. We can maintain a small static FAQ or glossary either in a file or in the database. Before answering user queries, the system can search not only the user's document but also this static knowledge base for relevant info. This improves answers to general questions (e.g. "What does 'floating rate' mean?" can be answered from the glossary rather than GPT guessing). We'll create embeddings for this glossary as well at startup. Codex can help write the code to initialize these common embeddings and merge them with user document search results.
- **On-Device Model ("Gemma"):** We plan a **tiered inference** approach:
    - First tier: Use **on-device logic or models** for simple questions or offline mode. "Gemma", as referenced in our blueprint, would be this component. It might not literally be an LLM on device due to size, but we can implement a rule-based responder for certain queries when offline. For example, if user asks "What is EMI?", the app could have a local answer stored ("EMI stands for Equated Monthly Installment…"). Or if the user asks to calculate something simple, the app's local code can handle it using formulas. If we had a smaller ML model (some teams fine-tune TinyML or use something like GPT-2 small for offline), that could be integrated, but initially a ruleset + cached answers can serve as Gemma.
    - Second tier: Use **Azure OpenAI GPT-4** when online for complex or natural language questions that go beyond the static knowledge. This gives the best quality answers leveraging the document content.

    - The app will decide which tier to use: if offline or if the query matches a known offline capability, use Gemma; otherwise, call the cloud.
    - We'll include a *"smart fallback"* in code: e.g., try to answer via local logic and if confidence is low or it's a complex query, then either queue it for cloud when internet is back or prompt the user that internet is needed for full analysis.

Codex can assist implementing this branching. For example, we can write a function `answerQuery(queryString)` and in comments outline: "if device offline -> call localAnswer(query), else -> call cloudAnswer(query)". Copilot will likely stub that out accordingly.

- **Prompt Structuring & Context:** Codex will help ensure our prompts to GPT-4 are well-structured:
- It can help format retrieved chunks as quoted context in the prompt (maybe bullet them with citations).
- It can also help implement a **"max token"** safeguard: chunk the context to fit within model limits (e.g. if doc is long, include top 3 relevant chunks only).

- We should also instruct GPT-4 to indicate when it's unsure rather than hallucinate. A system message like "Do not fabricate data not in the context" will be included. We can have Codex include such instructions automatically each time it builds the prompt string.

- **Voice and Translation Workflow:** For narration:

- If online, when user requests "Read out the summary", the app calls our function which uses Azure Speech. The function will likely call `SpeechSynthesizer.SpeakTextAsync` or a REST call to the TTS endpoint with the text. We have to pass the desired voice (for an Indian accent English voice, or a Hindi voice, depending on user preference). We'll use Codex to write this integration – provide the REST endpoint format in a comment and let it fill the code to POST to `https://<region>.tts.speech.microsoft.com/cognitiveservices/v1` with the proper headers and SSML.
- Offline, if no connectivity, the app uses `flutter_tts`. We integrate this plugin, and use a pre-downloaded voice if possible or the native TTS engine. The app will detect connectivity and either call the cloud function or use local. We might have Codex generate an abstraction like `speakText(text)` that internally chooses cloud vs local.

- For translation in UI: if the user wants the entire analysis in another language, we could pre-translate the key outputs using Azure Translator. E.g., after generating the English summary, we call Translator to get a Hindi version if user's language is Hindi. This can be done server-side or in app using Translator's API (if small text, it might be okay to call directly from app using a function-provided token). Codex can help by writing a quick translator call using the Translator Text API (which is a simple HTTP POST).

- **Hallucination Safeguards & Disclaimers:** We take multiple steps to keep the AI outputs accurate and safe:

- **Grounding:** Using RAG as described, GPT's answer will be based on real data from documents [9] [13]. We will log what context we provided each time, to later audit any odd answers.
- **Verification:** For critical calculations, we won't rely solely on GPT. For example, if GPT-4 answers "You will save ₹50,000 interest by prepaying", we will cross-verify that with our own amortization logic.

The function can calculate the same scenario and compare. If the numbers diverge significantly, we know GPT may have hallucinated or erred. We can then choose to trust our calculation and adjust the answer, or at least flag uncertainty. This could be implemented as a simple check: after getting GPT's answer, parse any numeric result and compare with deterministic computation. Codex can help parse out numbers from text for this check.

- **User-facing Disclaimers:** Every AI-provided result shown to the user will include a note like "(AI-generated)" or a tooltip saying the content is generated and may not be 100% accurate. We'll incorporate this into the UI by design. For voice output, we might have the voice prepend "This is an automated analysis. Please consult a financial advisor for confirmation." We will ensure this is done consistently – the app's design and legal requirement likely demand it. This can be stored as a constant string that is always appended.
- **Consent and Confirmation:** If the AI suggests an action (like "You should refinance your loan"), we will treat it as an informational suggestion, not a definite instruction. The UI will likely frame it as "Suggestion: Consider refinancing, because … (AI explanation)". This way the user understands it's not a guaranteed outcome. All these text phrasings we will craft with care, and Codex can help by providing rewording or tone adjustments as needed.

In summary, the AI workflow combines **Azure's robust AI services** with **prompt engineering** and **code logic to ground and verify** AI outputs. By designing with layers (offline, cloud, verification), we ensure the feature is both powerful and responsible. Codex's role is not just code – it's also an assistant in writing out these complex prompt strings and glue logic, saving us time and catching things we might overlook (like handling API errors or null responses).

Offline-First Data Strategy

For an app dealing with personal financial data, an **offline-first approach** ensures users can access their loan information and calculations even without internet. Our plan:

- **Local Database:** The Flutter app will include a local database to cache and persist loan records on device. We will use **Drift** (formerly Moor) as our Dart database toolkit, paired with **SQLite**. To secure sensitive data, we enable **SQLCipher** encryption for SQLite. Specifically, we'll use the `sqlcipher_flutter_libs` package with Drift, which transparently provides 256-bit AES encryption for the database [39] [40]. This means all loan details stored on the phone are encrypted at rest. We'll generate an encryption key (likely derived from the device or user credentials) – possibly using Flutter Secure Storage to hold a key – and supply it when opening the Drift database. (Drift's documentation shows how to call `NativeDatabase.open()` with a setup to execute `PRAGMA key = 'passphrase';` for SQLCipher [41] [42], which we'll implement).
- The local DB will have tables like `Loans`, `Payments`, `Analyses`. When the user scans a document and results come back from the cloud, we insert a record in `Loans` (with basic fields) and maybe an `Analysis` record that contains the summary, recommendations, etc. We also store any user-added data (if user can manually input a loan, that goes into the same tables).
- **Drift setup:** We define the schema using Drift's annotations or .drift files. Codex can expedite writing these model classes and queries. For instance, we list fields (loan amount, rate, etc.) and Copilot can generate the Dart data class and DAO methods. If we want to use Daos or direct queries, it can also fill those in.

- With encryption in place, even if someone extracts the app data, they can't read it without the key. (We'll ensure to handle key management properly – possibly tie it to device biometrics or user login if applicable, so that on first app install we set up the key).

- **Synchronization with Cloud:** Whenever the app goes online, it should **sync** data with the Azure backend (Postgres). This ensures a user can use multiple devices or recover data if they lose the phone, and also that heavy analysis done in cloud gets down to the device.

- We will implement a sync mechanism in the Flutter app, likely in a background isolate or triggered on app launch/when connectivity is regained.
- **Pull Sync:** The app will fetch any new or updated records from the server. For example, if user scanned a doc on web or another device (future expansion), or if our cloud analysis added recommendations to a loan entry, we need to update the local copy. We can have an API like `GET /loans?modifiedSince=TIME` to retrieve changes. The Function backend would query Postgres for any loan data changed after that timestamp and return a list (with deletes indicated perhaps). The app then merges these into the local DB.
- **Push Sync:** Conversely, if the user adds or edits data while offline (maybe adds a manual note or adds a new loan entry manually), the app will mark those records as "pending upload". Once online, it calls an API (like `POST /loans` or `PUT /loan/{id}`) to send those changes to the server. Our durable function (or a simple Function, since these are small requests) will write them to Postgres. We use some form of simple conflict resolution – e.g., last write wins, or the server could version records and reject if there is a conflict. Since this is mainly single-user data, conflicts are rare (it's not collaborative data). A reasonable strategy is to include a `lastModified` timestamp in each record both locally and on server. When syncing, compare and take the latest. If the server and local both changed a record differently, we might prompt the user or merge if possible. Codex can implement a merge function that compares fields and decides which to keep – but for simplicity, we might just decide one source of truth (server wins except when offline changes exist, then if conflict, prefer device's if recent).
- We will have Codex generate much of the (de)serialization code: transforming Dart objects to JSON for API and vice versa. Drift gives us objects, and we'll use something like `json_serializable` or manual parsing. Copilot is great at writing these boilerplates (mapping fields to JSON keys).
- **Network Handling:** We will integrate connectivity checks (perhaps using `connectivity_plus` package) to know when to trigger sync. If continuous sync is complex, we might also add a "Sync" button for user to manually trigger.
- **Background sync:** If using Firebase Cloud Messaging or similar, we could push a notification to device on server changes. But to keep it simpler and fully Azure, we might not do that for MVP. Instead, periodic sync (e.g. every launch or every X hours) could be done. We can use Flutter's background fetch or just on app resume events.

- Codex can create a service class `SyncService` with methods `syncDown()` and `syncUp()`. We'll outline in comments what each should do and let it draft the logic, including handling pagination if many records, handling errors, etc.

- **Data Merge Strategies:**

- For **inserting new loans**: that's straightforward – generate a new GUID or ID on the client, or let the server assign an ID and return it. We might use GUIDs so that even offline-created records have a unique ID that won't conflict with server's IDs (server can use the provided ID).
- For **updates**: tag each record with `lastModified` and maybe a `source` (cloud vs local). If an update happens offline, we mark it and during sync, if the server's lastModified is older, we apply the update. If server's is newer, we have a conflict (meaning the user's local data was stale). Because it's mostly user-driven data, conflicts might not occur often unless the user uses two devices. But we handle it gracefully: possibly prefer server data but alert user, or just take the latest timestamp always.
- For **deletions**: We need to sync deletes too. If user deletes a loan on one device offline, mark it as deleted (soft-delete in local DB) and sync up so server deletes it (or marks as deleted), then propagate to other device. We'll likely include a `isDeleted` flag and a deletion timestamp.
- **Audit Trail**: We might keep minimal logs of changes for debugging sync issues (maybe just in debug mode).

- We will rely on Codex to implement these merge rules in code. By writing pseudo-code in comments like "if local.lastModified > remote.lastModified, send local changes, else pull remote changes", Copilot can translate that to actual Dart logic inside our syncing function.

- **Working Offline UX:** The app will be fully functional offline for any already stored data:

- The user can open loan details, see all the analysis that was last available (from the last sync or processing).
- They can use calculators (which we implement fully on-device as well, for "what-if" scenarios). E.g., the app can locally compute a new amortization if user tweaks something, without needing GPT. This covers a lot of utility even offline.
- If they try to add a new document offline, we have two options: (a) allow them to scan and queue it for processing when online (store the image/text locally and mark for upload), or (b) require internet to analyze. We can implement a limited offline analysis: e.g. if the doc is a standard format, we could attempt device OCR using something like Firebase ML Kit or tesseract, but that's complex and likely out of scope for MVP. So we might notify user that internet is required to analyze new documents (but we won't block them from scanning – we'll just store it and process later). We can show the doc in a list as "Pending analysis".
- Codex can help by writing conditional UI: e.g. show a greyed-out analysis with note "Pending upload" if `loan.isAnalyzed==false` due to offline.

In essence, our strategy uses **Drift+SQLCipher for secure local storage** [40], and a custom **sync engine** to propagate data. We considered using an existing sync solution (like Hasura, or Firebase as a backend) but since we already have Azure and custom logic, a little custom sync code is fine. There's also packages like `offline_sync_kit` [43] which we could draw inspiration from, but writing our own with Copilot is quite feasible.

By designing offline-first, we ensure the app is robust in India's varied connectivity environments. All heavy AI tasks (document parsing, GPT answers) do require cloud, but the user can always view what's already downloaded and run local "what-if" calculations without a round trip.

Codex's assistance in this offline-online bridging is critical: from generating the **local database schema**, to the **serialization**, to the **diff/merge logic**, we'll use it to quickly produce working code and then test edge cases.

DevOps with Codex

To achieve a production-ready deployment, we will set up automated DevOps pipelines using GitHub Actions and infrastructure-as-code, with Copilot accelerating the process:

- **Infrastructure as Code (Bicep Templates):** We will write Bicep templates to define our Azure infrastructure (as outlined in the architecture section). This includes VNet, subnets, storage account, function app, Postgres, cognitive services, Key Vault, etc. Bicep is a concise language, but writing it from scratch is time-consuming. We'll leverage Copilot to generate snippets of Bicep:
- For example, in our Bicep file, we start typing a resource like `resource storage 'Microsoft.Storage/storageAccounts@2022-09-01' = { ... }` and Copilot often can auto-complete the required properties (name, sku, kind, etc.) because it has context of common patterns. We can use it to get the correct syntax for subnet delegation (`subnet.delegations` to Postgres), private endpoints (which involve resource `Microsoft.Network/privateEndpoints` with sub-resources for each service), and others. We'll verify against Azure docs, but Copilot's suggestions can be quite accurate for known patterns.
- We'll likely parameterize certain things (like environment name, region). Copilot can help set up those param declarations.
- The Bicep will also include outputs (like perhaps the Key Vault name, so we can use it in pipeline).
- After writing Bicep, we'll use Azure CLI or Azure PowerShell to deploy it. This too can be added to our pipeline (e.g. an Action using `azure/cli@v1` to run `az deployment group create -f main.bicep -g rg-loandocscanner-prod`). Copilot can basically write that YAML step when we describe it.
- **GitHub Actions CI/CD:** We will establish at least two workflows:
- **Infrastructure Deployment Pipeline:** This is triggered manually or on changes to the infra files. It logs into Azure (using OIDC so no secrets needed for auth) and deploys the Bicep. We'll set up Azure credentials as OIDC: store `AZURE_CLIENT_ID`, etc., in GitHub secrets, and configure the workflow for OIDC authentication [44] [45]. Copilot can reference Microsoft's examples to set this up. After login, we'll run an Azure CLI action to deploy. We will also include a *"what-if"* (dry-run) step for safety in non-prod.
- **Application Build & Deployment Pipeline:** On pushing new code to main (or via pull request merges), this pipeline runs. It will:
    - Use actions to set up the environment (for .NET or Node or Python for Functions, and Dart for Flutter).
    - **Backend:** Compile and test the Azure Functions code. If using .NET, run `dotnet build` and `dotnet test`. If all good, deploy the function app. We can use the `azure/functions-action` or `azure/webapps-deploy` to zip deploy the package to Azure. If using Python, we'd package and deploy similarly. We might also containerize the function (especially if using Python to avoid venv issues) and deploy a container – but that adds complexity. Likely, we use the run-from-package deployment for simplicity.
    - **Frontend:** We might set up a separate job to build the Flutter app APK/IPA (if targeting mobile). However, deploying mobile apps isn't as straightforward automated (it goes to Play Store/App Store). For MVP, we might skip automating app store deployment and just ensure

the app builds. We can use codemagic or other CI for Flutter if needed, but that might be outside scope. At minimum, we ensure the Flutter code passes analysis and tests (run `flutter analyze` and `flutter test`). We can integrate that into GitHub Actions using the Flutter action.
    - Codex will help writing these steps. For example, we type `- name: Build and deploy function` and Copilot might auto-fill a script with Azure CLI to do `func azure functionapp publish`. Or it might suggest using specific GitHub actions.
    - We will ensure the pipeline includes environment-specific variables (like resource names). We'll store non-secret config in the repo or use Action's `env` for small things, and for secrets, use GitHub Secrets.
- **Key Vault Secret Integration:** As mentioned, we avoid storing secrets in GH. For local dev, we might have a .env, but in CI, we use Azure Key Vault. We'll give our GitHub Actions workflow permission (via OIDC) to get secrets from Key Vault [23] [20]. For example, the workflow can retrieve the Postgres connection string or Function app publish profile if needed (though we can also use AZ CLI to publish by resource name and avoid needing a password). A sample snippet we'll use: an Azure CLI step to `az keyvault secret show` and export the value as env var, as shown in MS docs [46]. Copilot can basically copy that approach for us. This way, the pipeline can inject necessary secrets at build or deploy time securely.

- We will store our Azure service connection info (client ID, tenant, subscription) as GitHub Secrets and use OIDC login as recommended, rather than storing a service principal password.

- **Testing & QA Automation:** We plan to run automated tests in CI. Copilot will help us write integration tests for critical functions:

- For instance, an integration test that calls the orchestrator with a sample document and verifies it inserts a DB record. We could use Azure Functions Core Tools in a test mode or simply factor logic so it can be called in a test without actual Azure triggers. Alternatively, after deployment to a test environment, run a pipeline step that triggers a function with a known input and checks the output (maybe by reading from the DB or checking a status in storage). We can script that with Azure CLI or a small test script (Python or JS) in the pipeline. Codex can help write that script.

- UI tests for Flutter (widget tests) will run in CI via `flutter test`. We ensure any golden image tests are handled (or skip for now). If needed, we could set up Firebase Test Lab or emulators for integration tests on real devices, but that might be later.

- **Monitoring and Logging:** Part of "DevOps" is ensuring we have observability. We will use Application Insights (already integrated with Functions) to monitor logs and performance. We'll also possibly add additional logging in code (like log when an analysis starts and ends, etc.). These logs can be viewed in Azure portal. If we wanted to be fancy, we could set up alerts (like if a function fails often, or if CPU is high). But for MVP, basic monitoring is enough. We might output custom events (e.g. "DocumentParsed" events) to App Insights for telemetry on usage.

- **Deployment strategy:** For production release, we might use *staging slots* or *blue-green deployment* for the function app. But initially, it might be fine to deploy directly to production since it's small-scale. We can set the pipeline to deploy to a *test environment* first (maybe a separate Azure resource group for test), run some smoke tests, then promote to prod. GitHub Actions has environments where we could require a manual approval before prod deploy.

- **Codex in DevOps:** Copilot helps beyond just code – it can write YAML and scripts. We will use it to:

- Create our GitHub Actions YAML files. A lot of Actions syntax is boilerplate (like the OIDC login steps, checking out code, setting up languages). Copilot is quite adept at producing these if we comment our intent. For instance, writing `# Setup .NET and run tests` might get us a ready snippet with `actions/setup-dotnet` and `dotnet test`.
- Write Azure CLI commands for one-off tasks (like a script to initialize the DB schema on first deploy, or to upload a dummy document for testing).
- Generate documentation for the team: we can have Copilot draft a README for developers on how to deploy, just based on our pipeline config.

In summary, the DevOps pipeline will ensure each commit is tested and the infrastructure and app stay in sync. Everything is code-reviewed and reproducible. By leveraging Codex for the pipeline, we treat the **CI/CD config as code** as well, meaning we can develop it faster and reduce manual errors.

This robust CI/CD setup with Infrastructure as Code and secrets management readies us for compliance and smooth iteration – any new developer can get environment running by running the same Bicep or pipeline, and deployments are consistent and auditable.

Compliance & Consent Logic

Handling personal financial documents requires strict compliance with data privacy laws (e.g. GDPR) and user consent, especially since we use AI on the data. We design features to obtain user consent and manage data lifecycle accordingly:

- **User Consent Flow:** When the user first uses the app (or first uses the AI features), we present a clear consent screen. This will explain what data is collected (e.g. their documents, extracted info), how AI will be used (e.g. "we use AI to analyze your documents"), and any data sharing that occurs (with cloud services). The user must agree ("Accept") or they can decline (in which case we either limit functionality or exit). We'll have Codex assist in creating this UI screen in Flutter – likely a simple `AlertDialog` or a full screen with a scrollable policy text and two buttons. We can have Copilot draft the legalese if we provide some templates (though legal text we might get from legal advisors, but formatting it in app is straightforward). We ensure it's stored that the user gave consent (e.g. a boolean in local storage and in our backend linked to the user account if accounts exist).
- If the user is not authenticated (say we don't require login), the consent might be stored locally. We might also generate a unique user ID (random GUID) on first run, so that if later the user requests data deletion we can identify their data on the server.

- The consent UI would likely not allow proceeding until they scroll through (to ensure they saw it). We can use Flutter widgets for that (Copilot can help make e.g. a Checkbox "I agree" that only enables the continue button when checked).

- **Privacy Policy & Terms:** We will also include a section in-app where users can review the privacy policy and terms anytime, and withdraw consent if they choose. Codex will help create these screens quickly (static text screens). It might also help by converting markdown or HTML policy text into a Flutter `RichText` or set of Text widgets.

- **Data Retention and Deletion (Auto-Erasure):** We commit to deleting or anonymizing user data after a certain period or on request:

- **User-initiated Deletion:** Within the app, provide a "Delete my data" option. If a user taps this (and confirms), the app will call an API (e.g. `DELETE /user/data`), which triggers deletion of their data in our backend. Implementation: We can have a Durable Function orchestrator handle this too – one that upon receiving a delete request, will wipe or anonymize records in Postgres (delete rows or replace personal fields), delete blobs from storage, and remove any search index entries. This orchestrator should also log an audit entry (so we have a record that data was deleted upon request).
- Codex will assist writing the deletion routines – e.g. SQL to delete a user's rows across multiple tables (loans, analyses, etc.), and Azure SDK calls to delete blobs. We need to be careful with Durable Functions here: deletion should be idempotent and handle partial failures (e.g. if it deletes DB but fails to delete blob on first try, it can retry blob deletion).
- **Scheduled Auto-Deletion:** To comply with regulations like GDPR data minimization and also not to hold data indefinitely, we plan to auto-delete personal data after a retention period (the blueprint mentions **7 years** which might align with some financial record-keeping requirement or could be a business choice). We will implement this via a scheduled Durable Function or a combination of Timer triggers:
    - One approach: Each piece of data (loan record) has a timestamp when it was added. We also maintain an "expiry date" = timestamp + 7 years.
    - We can use a **Durable Timer** in an orchestrator set to fire at the expiry date [26] [47]. For example, when a loan is processed, we start a deletion orchestration that waits until expiry and then deletes that data. Durable Functions can schedule timers far into the future (they survive restarts by checkpointing). This is elegant because it's per-document scheduling.
    - Alternatively, simpler: a periodic **Timer Trigger** function that runs daily, finds any data older than 7 years and deletes it. This might be easier to implement and more adjustable. We could write a Timer trigger in Bicep (CRON schedule every 24h) and have it query the DB for old records and delete them.
    - Either way, after deletion we should possibly keep a minimal audit trail that data was deleted at X date to prove compliance.
- **Audit Logs with Retention:** We will maintain an **Audit** log of key events (document uploaded, analysis done, data deleted) in an append-only manner. To make these tamper-proof and meet financial audit needs, we will use Azure Blob Storage with **immutable storage policies**. Specifically, an Azure Blob Container (say `auditlogs`) can be created with a **time-based retention policy of 7 years** (which Azure supports up to 400 years) [48]. We will log events as blobs or append blobs. Once written, the immutability policy means they cannot be altered or deleted until the retention period passes [49] [50]. This satisfies regulations that certain records must be kept unmodified (WORM storage) [51]. We'll set the policy to lock after perhaps a short test period. After 7 years, these logs can either auto-expire or we can archive them.
- We'll use Codex to implement writing to these logs. For example, after a successful document processing, an entry like "USER XYZ – Doc ABC analyzed – on 2026-02-06" is written. The retention is handled by storage policy (no code needed except initial setup). We do need to be cautious to not log sensitive content in these audit logs (just IDs and actions, not full document text, to respect privacy).

- If regulators need proof of deletion, these logs can show that it happened (plus we can log a checksum of deleted data or something if needed).

- **Secure Handling & Least Privilege:**

- Our Function will use Managed Identities to access resources (instead of embedding keys). For instance, it will connect to Postgres using Azure AD auth if possible (we can create a DB user mapped to the function's identity). If not, at least the DB creds are in Key Vault and not in code.
- The storage container for documents could be configured to only allow access via the Function's identity, not public. If app uploads directly, it uses SAS tokens we generate with limited scope.

- We'll enforce that no data leaves our system unauthorized: e.g., the OpenAI API call should not log content (we opt out as mentioned).

- **Consent Withdrawal:** If user withdraws consent (maybe a toggle in settings), we treat it as a request to delete data. Alternatively, if they withdraw consent but still want to use app without AI, we could stop processing new docs but keep local data. However, likely if they don't consent to data processing, the functionality is severely limited. We'll define that clearly. Possibly we say "If you don't consent, you can use the manual calculator features but not document scanning".

- If they withdraw after already having data in cloud, we could either immediately delete it or anonymize it. We'll lean toward deletion as courtesy (unless there's a legal reason we must keep it for 7 years – but if so, we would then anonymize it and keep only minimal info).

- The app will allow toggling some settings like whether to send usage telemetry (we should respect if they turn off analytics, though for now, we mainly have functional data).

- **Compliance Standards:** We map our approach to standards:

- GDPR: Provide ability to delete data (Right to be Forgotten), data export if needed (we could add an option to export all their loan info as JSON or CSV), and transparent consent.
- Local regulations in India: Ensure data residency if required (our approach tries to keep data regionally, except the model inference in worst case).
- Fintech rules: Possibly need to keep records 7 years (hence our retention). Also we keep audit logs as WORM which is often required by regulations (e.g. SEC 17a-4 for financial communications – our immutable logs meet similar criteria [52]).

Codex will help implement many of these compliance features quickly: - It can generate the *consent form UI code* and even suggest phrasing. For example, if we put in a comment some key points, it might write a nicely formatted paragraph in a Text widget. (We will validate the text with legal advisors, but it helps as a draft.) - It will generate deletion code, which is tricky to get right (e.g. it might for instance produce pseudocode to iterate user's loans and delete each, etc.). We'll test those carefully. - It can help with the timer or orchestrator setup for scheduled deletion. By describing the scenario ("schedule deletion in 7 years"), it might set up a `context.createTimer(Context.CurrentUtcDateTime + TimeSpan.FromDays(2555))` (7 years ~ 2555 days) and then continue with deletion calls – which is exactly what we need (though we have to ensure the orchestrator doesn't overload memory with too many timers; might be fine given our scale). - For the audit log, given an example, Copilot can probably write the code to append text to a blob and set the immutability policy if not already set. (Actually, we'll set the policy via Azure CLI once on container creation, not in code.) - **Testing compliance logic:** We'll also have Copilot generate unit tests for these flows (like a test that creating a loan schedules a deletion, or that deletion actually removes all data). This gives confidence and documentation of intended behavior.

Overall, compliance is built into design, not tacked on. Users will know their data usage, can control it, and data will self-clean after a period. These features, often neglected in MVPs, are made feasible thanks to Codex accelerating the development of "boilerplate" but crucial code, like policy screens and data management tasks.

Automation Priorities

In building this platform, we will prioritize automating the most complex or time-consuming components with Codex, allowing the small development team to focus on fine-tuning and business logic. Key areas where we "let Codex build it" include:

- **Prepayment Optimization Module:** We need a component that can evaluate various **prepayment scenarios**. For example, "If the user pays an extra ₹X at month Y, how much sooner will the loan finish or how much interest will be saved?" Doing this manually involves recalculating the amortization schedule with the lump sum reduction. We'll offload this to Codex:
- We'll outline the algorithm: iterate through payments, when you reach month Y, deduct X from remaining principal, then continue calculation either with same EMI (loan ends sooner) or recalc EMI (if contract says EMI reduces). There might be different modes (keeping EMI same shortens term vs reducing EMI keeps term same).
- We can have Codex generate functions like `List<Payment> simulatePrepayment(Loan loan, int month, double extraAmount)` returning a new schedule, and another to compute summary of interest saved.
- We'll also have it generate multiple test cases: e.g., if prepayment is made at start vs at end, interest saved should differ sensibly (paying earlier yields more savings). We can supply known small examples (maybe even compare to an online loan calculator for validation).

- By letting Copilot write this, we quickly get a working solution which we then verify for edge cases.

- **EMI Extension Optimizer:** In some cases, users might want to reduce their EMI by extending the tenure (common in loan restructures). We can implement a feature: "Reduce my EMI by X%, what new term would that require?" or "I want to pay ₹N per month, how long to finish the loan?" This involves solving the amortization formula for N (which might not have a closed form easily, but can be done via numeric iteration or using log formula).

- Codex can help either derive the formula or just brute-force by incrementing months until the EMI fits.
- We describe: "given desired EMI, find term months such that EMI formula output <= desired (with maybe a binary search)". Copilot can implement binary search or iterative approach for us. We'll test on known values (if EMI is equal to interest-only payment, term tends to infinite).

- Another angle: optimizing *when* to refinance or what rate drop makes sense – those could be future expansions, but not top priority now.

- **Sample Logic Implementation:** For features like these optimizers, we'll also rely on sample scenarios to validate. We can write a few user stories (like "User has loan of 5 Lakhs at 10% for 5 years. If they prepay 50k at year 2, what's the new duration?") and then ensure our system gives a

reasonable answer (maybe compare with manual calc). Codex can generate these scenarios as documentation or tests.

- **Test Scaffolding & Automation:** We place high priority on test automation to catch any financial miscalculations. Copilot will help create **unit test scaffolds**:

- For the formula calculations (EMI, interest, etc.) – provide known reference outcomes (possibly from an external calculator or formula). Codex can even embed the formula derivation as comments to make the test self-explanatory.
- For the orchestrator workflow – we can simulate an orchestration by calling activity functions with dummy data in tests (not fully running durable runtime but by modularizing logic). Copilot can help craft mocks or fakes for the AI calls (so tests don't call actual Azure services).
- For sync logic – simulate conflicting updates and see that our merge picks the right winner.
- For compliance – test that data older than 7 years is filtered by our deletion function.

Having these tests ensures that as we refine code (with Copilot's further assistance), we don't break earlier functionality.

- **Continuous Improvement via AI:** Once the MVP is up, we will continue to use Codex to optimize code. Perhaps identify any slow spots (using profiler or App Insights) and then prompt Copilot to refactor for better performance (like using bulk inserts instead of loops for DB, or caching results). Codex might suggest more optimal data structures if prompted.

- **User Feedback Loop:** If early users find issues or request features, we incorporate quickly and use Copilot to implement. For instance, if users want a feature to compare two loans (maybe to help decide between options), we can rapidly prototype that with Copilot writing the comparison logic.

By prioritizing high-impact features (prepayment and EMI adjustments) and trusting Codex to handle the heavy lifting, we maximize development velocity. The team can focus on verifying accuracy and ensuring a good UX, rather than hand-coding every formula and loop.

Finally, automation isn't just in code: our processes (like deployment, testing, data management) are automated to a high degree, meaning we can scale and iterate fast without getting bogged down in manual steps.

Sources:

- Azure Durable Functions used for orchestrating document processing pipelines [24] [7]
- Azure AI Document Intelligence for extracting structured data from loan documents [7] [8]
- Azure AI Search or vector DB stores embeddings for RAG-based Q&A [13] [14]
- pgvector extension on Azure Postgres enables vector similarity search for embeddings [10] [12]
- GitHub Copilot generating durable function orchestration and activities from a single prompt [27] [28]
- Copilot accelerates Flutter development (UI, state management, API calls, tests) [32] [53]
- Drift local database with SQLCipher provides on-device encrypted storage [40] [41]
- Using Azure Key Vault and OIDC in GitHub Actions to manage secrets securely [22] [45]

- Immutable blob storage for compliance – supports retention policies up to 400 years [48] and WORM storage for regulatory compliance [49] [51]
- Region considerations for Azure OpenAI (GPT-4) and opting out of data retention for compliance [19]
- EMI calculation formula for loans (amortization formula) [30]
- FL Chart library for Flutter supports visualizing data in line/bar charts [36]
- Example of Copilot understanding business logic and implementing transformation + error handling in orchestrated workflows [28] [54]
- Copilot generating state management boilerplate from simple prompts [34] [35]
- Durable Functions allowing flexible scheduling and recurring processes (monitor pattern) [25] [26]

---

[1] [27] [28] [38] [54] Khoj Information Technology | From Code Assistant to Cloud Architect: How GitHub Copilot Built Our Azure Function Apps
https://www.khoj-inc.com/from-code-assistant-to-cloud-architect-how-github-copilot-built-our-azure-function-apps/

[2] [3] [4] [32] [33] [34] [35] [37] [53] Flutter + GitHub Copilot = Your New Superpower / Blogs / Perficient
https://blogs.perficient.com/2025/07/04/flutter-github-copilot-your-new-superpower/

[5] [6] [7] [8] [9] [13] [14] [24] Automate Document Classification in Azure - Azure Architecture Center | Microsoft Learn
https://learn.microsoft.com/en-us/azure/architecture/ai-ml/architecture/automate-document-classification-durable-functions

[10] [11] [12] [18] [29] Vector search on Azure Database for PostgreSQL | Microsoft Learn
https://learn.microsoft.com/en-us/azure/postgresql/extensions/how-to-use-pgvector

[15] [16] [17] Networking overview with private access (virtual network) | Microsoft Learn
https://learn.microsoft.com/en-us/azure/postgresql/network/concepts-networking-private

[19] The limited availability of AzureOpenAI models is really annoying : r/AZURE
https://www.reddit.com/r/AZURE/comments/1e97a7c/the_limited_availability_of_azureopenai_models_is/

[20] [21] [22] [23] [44] [45] [46] Use Azure Key Vault secrets in a GitHub Actions workflow | Microsoft Learn
https://learn.microsoft.com/en-us/azure/developer/github/github-actions-key-vault

[25] [26] [47] Durable Functions Overview - Azure | Microsoft Learn
https://learn.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview

[30] [31] Amortization Calculator Online – Calculate Loan Schedule, EMI & Table
https://www.vedantu.com/calculator/amortization

[36] GitHub - imaNNeo/fl_chart: FL Chart is a highly customizable Flutter chart library that supports Line Chart, Bar Chart, Pie Chart, Scatter Chart, Radar Chart and Candlestick Chart.
https://github.com/imaNNeo/fl_chart

[39] How to encrypt the SQLite database in Flutter? - Stack Overflow
https://stackoverflow.com/questions/50232418/how-to-encrypt-the-sqlite-database-in-flutter

[40] [41] [42] Encryption
https://drift.simonbinder.eu/platforms/encryption/

[43] offline_sync_kit | Flutter package - Pub.dev
https://pub.dev/packages/offline_sync_kit

48 49 50 51 52 Overview of immutable storage for blob data - Azure Storage | Microsoft Learn
https://learn.microsoft.com/en-us/azure/storage/blobs/immutable-storage-overview