Architectural Blueprint for Contextual, Composable Frontline Applications: Deconstructing the RAG-Powered PDF-to-App Workflow

Part I: The Strategic Role of AI in Frontline Operations (Tulip Analysis)

1.1. Contextualizing Frontline AI: The Shift from Legacy Systems to Human-Centric Augmentation

The implementation of Artificial Intelligence (AI) within the manufacturing sector has transitioned from a theoretical discussion to a strategic necessity. The primary challenge facing industrial organizations is no longer whether to adopt AI, but how to scale it effectively across real-world operational workflows.[1] Traditional industrial systems, such as Manufacturing Execution Systems (MES) and Enterprise Resource Planning (ERP), often act as monolithic barriers, requiring specialized developers and extensive time for process digitization, thereby hindering the deployment of scalable AI solutions.[1]

Tulip Interfaces addresses this challenge by championing a philosophy of augmentation rather than full automation. The core mission of the Frontline Copilot is to assist and enhance the capabilities of human operators and engineers, empowering them to execute their tasks faster, smarter, and with greater efficiency.[2] This human-centric approach dictates that the underlying system architecture must provide instantaneous, context-aware decision support, thereby eliminating the need for repetitive searches and guesswork that traditionally plague manual work instructions.[1]

This augmentation strategy is built upon a foundation of composability. The Tulip platform utilizes a cloud-based editor that enables non-developers—those closest to the operation—to create user-friendly, digital applications through an intuitive drag-and-drop interface.[4] This architectural feature is vital, as it allows for the rapid digital transformation of processes and enables high standardization without reliance on dedicated data scientists or external developers.[1] The platform supports inherent modularity, edge connectivity, and integration with third-party systems, establishing the framework for highly flexible, non-linear workflows.[4]

1.2. The Tulip AI Frontline Copilot Ecosystem: Capabilities and Grounding

The Frontline Copilot is a set of Generative AI tools engineered to integrate seamlessly into the operator's interface, providing direct support where and when it is needed most.[2] The system offers real-time contextual decision support by leveraging various data sources, including production data and historical issue reports, to recommend troubleshooting steps, identify similar defects, and highlight where previous problems occurred.[1]

A critical functionality of the Copilot is its capability for Document Question-and-Answer (Q&A), which allows operators to ask freeform questions about internal policy, procedure, and process documentation (typically stored as PDFs).[6] This confirms the system's reliance on a **Retrieval-Augmented Generation (RAG) architecture**, which strengthens the Large Language Model (LLM) by injecting retrieved, trusted data from authoritative knowledge bases into the prompt, ensuring responses are relevant and grounded.[7]

The essential difference between this enterprise RAG implementation and consumer models is the requirement for **trust and visual verification**. To enhance reliability and meet compliance standards prevalent in complex manufacturing and regulated industries [4], the RAG system must be sophisticated enough to not only return a text-based answer but also to **highlight specific pages or sections on the source document that pertain to the answer**.[6] This visual grounding capability reduces LLM hallucinations and allows the operator to instantly verify the source of the information.[3] This requirement—linking an AI-generated text response back to a precise location within a source PDF—is the primary driver for the highly complex PDF parsing and coordinate extraction architecture necessary for the system's operation.

1.3. Feature Deep Dive: AI Composer and the Synchronized UX

The AI Composer is a specialized tool within the Tulip ecosystem designed to dramatically accelerate the digitization of standard operating procedures (SOPs) and work instructions.[9] The workflow involves uploading a PDF, which the Composer then processes by automatically extracting all relevant text, images, steps, checklists, and variable mappings.[1] The result is a fully functional, near-production-ready digital workflow or app, ready for rapid deployment and customization.[10]

The Role of Composable, Editable Questionnaires

The output of the Composer is a **composable, editable questionnaire or app** structure. The system transforms the static instructions within the PDF into dynamic inputs and workflow steps—such as checklists, required inputs, and media prompts—which can be quickly customized and integrated with connected devices or third-party systems.[1] The inherent flexibility of the platform's composable foundation allows the AI to generate a highly adaptive digital workflow, satisfying the user's need for functional, editable forms.

The Synchronization Mechanism: Side-by-Side Reference

The digitization process includes a review and editing phase where the user can refine the extracted content in an "intuitive side-by-side view with the source" document.[12] This interface is the foundation of the synchronous user experience. The most advanced implementation of this feature is the ability to link dynamic form fields in the digitized workflow to the specific, corresponding steps in the PDF reference panel, often achieved through synchronized scrolling and context highlighting.

The seamless conversion of a static PDF document into an editable application requires deep technical integration between three major architectural components:

1. **A sophisticated document parser:** This component must interpret the document's structure, not just its text, and extract geometric metadata like bounding box coordinates for all relevant elements.
2. **A Generative LLM and Inference Engine:** This layer infers the functional schema (what kind of data field is needed) from the extracted text structure (e.g., an instruction followed by a blank space should become a text input field).
3. **A Dynamic Form Builder and Renderer:** This component consumes the inferred schema (often in JSON format) and renders the interactive form on the front end.

The interoperability between these layers mandates the use of a **Unified Metadata Model**. This model must ensure that the geometric bounding box coordinates, extracted during the initial ingestion stage, are consistently associated with the derived text, carried through the RAG retrieval process, and accurately used by the front-end PDF viewer to synchronize scroll positions and render the required visual highlight.

Table 1 provides a technical breakdown of how the platform's advertised features map to necessary architectural elements.

Table 1: Frontline AI Composer Feature Decomposition and Technical Translation

Tulip AI Feature	User Benefit	Technical Translation
AI Composer (PDF-to-App)	Instantly digitize SOPs; rapid workflow creation.	Layout-aware document parsing; LLM structural inference; JSON Schema generation.
Composable, Editable Forms	Flexibility for custom data capture; no-code development.	Dynamic Frontend Renderer (e.g., SurveyJS); JSONB data persistence.
Document Grounding	Trust, accuracy, and reduced hallucinations in responses.	RAG pipeline (Retrieval + Augmentation); context injection.
Synchronized Reference UX	Instant visual verification; eliminates searching and guesswork.	Coordinate extraction (bounding boxes); front-end scroll/highlight synchronization logic.

Part II: Comprehensive RAG Pipeline Architecture for Document-Grounded Generation

Achieving the high-fidelity synchronization and grounding demonstrated by the Frontline Copilot requires a specialized, production-grade RAG pipeline that is optimized for geometric

data transfer and structural document handling.

2.1. The Ingestion and Pre-processing Layer: The Foundation of Geometric Metadata

The primary differentiator of this specialized RAG system is the depth of document understanding in the ingestion stage. Simple PDF text extraction or fixed-size token chunking is wholly inadequate.[13] To support highlighting specific content and synchronize scrolling to the source text, the parser must perform **Layout-Aware Parsing**.[14]

This process involves using advanced tools to interpret the structural elements of the PDF—identifying headings, tables, lists, and paragraphs—and extracting the **geometric coordinates** (bounding boxes) for every significant text block relative to the document page.[15] Specialized libraries like PyMuPDF (fitz) offer strong low-level capabilities for extracting text positions and metadata [15], while enterprise solutions like Unstructured.io or cloud-native layout parsers (such as Google's Vertex AI Search layout parser) are recommended for complex, mixed-content technical documentation, as they can reliably output structured JSON containing both the content hierarchy and location.[14]

Chunking Strategy and Embedding

The chunking strategy must align with the document's semantic flow, which is crucial for technical manuals and SOPs. **Recursive Chunking** is recommended, as it applies a hierarchy of separators (e.g., section, then paragraph, then sentence) until the content fits within a target size, ensuring that chunks remain semantically complete and relevant to a specific instruction or step.[13]

Once the semantically rich chunks are created, they are converted into dense numerical arrays (embeddings) using a suitable model. The quality of the embedding model directly correlates with the accuracy of retrieval.[19] For high-stakes industrial applications, high-performance models like the E5 Family, BGE, or the OpenAI Text-Embedding 3 series are generally preferred, though system architects must balance the trade-off between higher-dimensional vectors (often yielding better accuracy) and the resulting higher computational resources, storage costs, and latency incurred during comparison and retrieval.[19]

Coordinate Normalization Pipeline

A fundamental technical challenge inherent in this architecture is the mismatch between the coordinate systems. The parser extracts coordinates in the PDF's native units (typically PostScript points relative to the page origin), while the front-end PDF viewer renders the document using pixels (relative to the browser viewport, modified by zoom and scale).[21]

To solve this, the stored coordinates must be normalized at the time of ingestion to a standardized format and carried as metadata. The system must then incorporate a robust, client-side **Coordinate Transformation Service**. This service dynamically translates the stored PDF points into the current screen pixel geometry when the document is rendered, accounting precisely for the PDF viewer's current scroll offset and zoom level. This translation service is a central point of engineering complexity and requires rigorous validation to ensure accurate highlight placement across various devices and scaling factors.

2.2. The Retrieval Service Design: Hybrid Search and Structured Payload

The Retrieval Service must be optimized for speed and context delivery.

Hybrid Retrieval and Indexing

Effective retrieval requires a system capable of combining multiple search techniques. **Hybrid Retrieval** marries semantic search (using the vector index to find conceptually similar content) with traditional keyword search (using sparse indices to find exact matches for technical specifications or part numbers).[23] This dual approach maximizes retrieval effectiveness for diverse industrial queries.

The Vector Database (e.g., Qdrant, Weaviate) is where the indexed documents reside. For this architecture, each indexed entry must store four critical components:

1. The vector embedding of the text chunk.
2. The raw text chunk content.
3. Essential RAG metadata (e.g., document_id, document title).

4. The precise **geometric metadata** (page_number, bounding_box_coordinates) extracted during ingestion.[16]

RAG API Structured Response

The API endpoint that serves the RAG results must deliver more than just a text answer; it must return a fully structured JSON payload to the front-end application.[24] This structured response must include the LLM-generated answer *and* the precise geometric coordinates of the grounding context. This structure allows the front end to simultaneously display the answer and trigger the required visual actions (highlighting and scrolling).

2.3. The Generation and Augmentation Layer: Schema Inference and Context Injection

The generative component of the system serves two distinct functions: initial workflow creation and real-time operator assistance.

LLM Role in Schema Generation

During the AI Composer's PDF-to-app conversion, the LLM utilizes the structured text and layout data retrieved from the document store to perform **Schema Inference**.[10] The LLM analyzes the semantic content (e.g., a "Warning" section should become an alert component, a "Required measurement" should become a numeric input field) and generates a structured JSON schema that defines the inputs, logic, and flow of the new digital application.

Runtime Augmentation and Orchestration

For real-time Copilot assistance, the LLM receives an augmented prompt containing the user query, any relevant chat history, and the contextual data—including coordinates—retrieved by the RAG service.[7] This enhanced context allows the LLM to generate responses that are

accurate and specifically grounded in the company's knowledge base.[8]

Furthermore, a true enterprise Copilot often requires the capability to handle requests that involve both knowledge retrieval (RAG) and specific transactional actions (e.g., querying real-time inventory or processing an order). This demands a sophisticated orchestration layer that integrates RAG with **Function Calling** capabilities.[23] The orchestration layer must manage complex, multi-step user requests by intelligently determining when to fetch contextual data (RAG) and when to execute a defined external function, ensuring seamless navigation between knowledge-based responses and operational transactions.

Table 2: Retrieval-Augmented Generation (RAG) Pipeline Tooling Stack

RAG Stage	Functionality Requirement	Recommended Technologies/Libraries	Relevance to Synchronization
Parsing & Ingestion	Layout-aware PDF extraction (coordinates, tables, hierarchy).	Unstructured.io, Google Layout Parser, PyMuPDF.	Must extract Bounding Box Coordinates for visual linking.
Chunking	Semantic and Recursive splitting optimized for complex manuals.	LangChain/LlamaIndex, utilizing structural separators.	Ensures chunks are semantically whole and linkable to distinct form steps.
Embedding	Semantic vectorization of text for high-fidelity retrieval.	BGE, Cohere Embed v3, OpenAI Text-Embedding 3.	Determines the accuracy of retrieving the * correct* text snippet and its associated coordinate metadata.
Vector Database	High-speed similarity search and metadata storage.	Qdrant, Weaviate, Pinecone (supporting hybrid search).	Must efficiently index and return coordinate metadata with low latency.

Generation	LLM to interpret query/context and format structured output.	GPT-4/5, LLama 3, Mixtral (API access).	Converts retrieved context/coordinates into a structured response for UI consumption.

Part III: Frontend Engineering for Dynamic Forms and UX Synchronization

The complexity of this system is heavily concentrated on the client-side, where dynamic form generation must intersect with precise, annotated PDF rendering and scroll synchronization.

3.1. The Composable App Builder Framework: Runtime Rendering Architecture

The front-end framework must be architected for maximal flexibility and performance. This begins with a decoupled architecture where the UI rendering is driven entirely by a configuration layer (the JSON Schema).

Dynamic Renderer and Metadata Embedding

Libraries such as SurveyJS or formio.js provide the necessary infrastructure to dynamically ingest the flexible, nested JSON schemas generated by the AI Composer and render complex forms compatible with common modern JavaScript frameworks (React, Vue, Angular).[27] These libraries facilitate the creation of custom input fields and workflows, fulfilling the requirement for composable, editable questionnaires.

Crucially, the architecture must support **pre-grounding** to reduce runtime latency. Performance is paramount in industrial settings, where sluggish AI interactions erode operator trust.[3] To avoid a costly RAG query for every field interaction, the bounding box coordinates (page_number, chunk_ID, coordinates) of the corresponding source text should be embedded

directly as non-rendered metadata within the JSON definition of each generated form field during the initial AI Composer run. This means that focusing on a form field only requires a low-latency metadata lookup, reserving real-time RAG calls only for complex, free-form Q&A sessions.

3.2. High-Fidelity PDF Rendering and Annotation

The PDF panel must function as a synchronized, interactive viewer.

Viewer Technology and Highlighting

The PDF viewer requires high-performance rendering and the capability for external programmatic manipulation. Open-source solutions like PDF.js are viable for rendering PDFs in a browser environment [15], though commercial or custom SDKs are often necessary for advanced features like seamless server-side editing or optimized WebAssembly (Wasm) rendering.[29]

Programmatic highlighting is achieved by creating dynamic HTML elements (e.g., transparent, absolute-positioned div overlays) on a layer positioned directly above the PDF canvas.[21] These overlays are defined by the returned bounding box coordinates.

Execution of Coordinate Transformation

The front-end must execute the **Coordinate Transformation Service** immediately upon receiving the stored PDF points from the form field metadata or the RAG API. This client-side function references the PDF viewer's current state (scroll offset, zoom level, and scale) and calculates the precise pixel coordinates for positioning the HTML overlay and triggering the scroll operation. This service is a major engineering effort, requiring precise handling of document scaling to ensure the highlight rectangle accurately overlays the target text despite user interaction.

3.3. Scroll Synchronization Mechanisms: Linking Semantic Flow to

Visual Position

The most sophisticated feature is the real-time synchronization between the dynamic form panel and the static PDF reference panel. This synchronization must be bi-directional.

Targeted Deep Link (Form-Driven Scroll)

When an operator focuses or clicks on a specific form input field, the system performs a **Targeted Deep Link**. It retrieves the pre-embedded coordinate metadata for that field, uses the Coordinate Transformation Service to calculate the target pixel position, and triggers a smooth scroll operation within the PDF viewer's frame to bring the corresponding source instruction into the operator's immediate viewport.[30]

Passive Scroll Sync (Form Panel to PDF)

Synchronizing the movement of two independent scrollable containers presents a challenge, as simple linear percentage syncing often fails due to divergent content heights—the dynamic form content is variable, while the PDF content is fixed.[31]

The highly effective approach is **Semantic Section Synchronization**. When the operator scrolls the form panel, the JavaScript event listener identifies which **semantic chunk** (e.g., "Step 3: Verification Check") of the dynamic form is currently in the viewport. The system then forces the PDF viewer to scroll and display the page and section coordinates associated with that semantically relevant chunk, ensuring that the visual reference panel always aligns with the current operational instruction, regardless of minor scroll discrepancies.

Table 3: Frontend Synchronization Implementation Matrix

Mechanism	Trigger Event	Client-Side Implementation	Challenges & Mitigation
Form Field Deep Link	Operator focuses/clicks specific form input	JS scrollIntoView on PDF viewer; programmatic highlight of	**Challenge:** Latency if RAG call is required. **Mitigation:**

	field.	associated text.	Pre-cache bounding box metadata in the form schema (low-latency lookup).
Passive Scroll Sync	Operator scrolls the content within the Form Panel.	JavaScript onscroll listener; calculates active semantic section/step ID.	**Challenge:** Divergent scroll ratios between dynamic form and fixed PDF. **Mitigation:** Sync based on semantic section (Step ID) rather than linear scroll percentage.
Coordinate Transformation	RAG API returns bounding box (PDF units) or form metadata accessed.	Custom JS function to convert PDF points to current browser pixels (accounting for zoom/scale).	**Challenge:** Inaccuracies due to variable zoom/device resolution. **Mitigation:** Use WASM PDF viewer with explicit scale management and device calibration testing.
Highlight Rendering	Coordinate transformation complete.	Dynamic HTML overlay ( ) using absolute positioning (z-index above PDF canvas).	**Challenge:** Ensuring highlights disappear upon context change. **Mitigation:** Implement clear and efficient state management to remove overlays on context shift.

Part IV: Backend, Data Persistence, and Scalability

While the AI pipeline and front-end synchronization are the most visible components, the success of a composable industrial platform relies heavily on a robust and flexible backend infrastructure.

4.1. Data Modeling for Composable Applications

Industrial applications must reconcile the need for structured data (required for traceability, analytics, and GxP compliance) [4] with the inherent flexibility of AI-generated, composable workflows.[5] Traditional relational modeling struggles when schema needs to change rapidly, or when complex, nested data structures are involved, often leading to performance-impacting join operations.[32]

The optimal approach is **Hybrid Persistence** utilizing a modern relational database with advanced NoSQL capabilities. PostgreSQL's JSONB datatype is highly recommended.[33] JSONB allows the system to store the variable, nested definitions of the AI-generated form schemas and the resulting form instance data within a single column.[33] This maintains the transactional reliability and indexing power of a relational database while providing the schema flexibility and simplified querying typically associated with document databases. This adherence to a Unified Data Architecture (UDA) is critical for handling complex, hierarchical data structures efficiently for subsequent analytics.[34]

4.2. API and Orchestration Layer

The middleware layer must be designed to handle orchestration, data management, and low-latency RAG requests.

A dedicated **Orchestration/Composer API** is required to manage the multi-stage, asynchronous workflow triggered by the PDF upload: handling the handoff between the parser, the chunking/embedding services, and the LLM schema generator. This API ensures

that the entire PDF-to-app conversion process is tracked and completed reliably.[11]

A **Low-Latency RAG API** is essential for all real-time operator queries, designed for high throughput and sub-second response times. Finally, a standard **Data API** handles the CRUD operations for the JSONB-stored form definitions and execution data.

Given that frontline operations often occur in environments prone to network disruptions [4], the architecture must incorporate mechanisms for **concurrency and resilience**. This includes specifying client-side data persistence strategies, such as utilizing libraries like Room or DataStore for local caching on operator devices, which allows work to continue offline.[36] The orchestration layer must then manage the complex synchronization and transaction management required to reconcile locally stored data once network connectivity is restored.

4.3. Strategic Recommendations for Implementation and Validation

The replication of a system like the Tulip Frontline Copilot, specifically its PDF-to-app conversion with synchronized scrolling, necessitates focused investment in three strategic areas:

1. **Prioritization of Ingestion Quality:** The quality of the synchronized user experience is directly proportional to the accuracy of the initial document parsing. Investing in, or developing, highly accurate layout parsers capable of extracting precise geometric metadata and semantic structure is the single highest-return architectural decision. If the initial coordinate extraction is flawed, the front-end synchronization and highlighting will fail, regardless of the quality of the LLM or the front-end code.
2. **Performance Benchmarking of RAG Latency:** Since the Copilot's success relies on augmenting the human operator in real-time, RAG query latency must be aggressively benchmarked and optimized to ensure sub-second response times for crucial interactions.[3] Any delay introduces friction and compromises operator trust, potentially leading to system abandonment.
3. **Phased Deployment Strategy:** Given the significant engineering complexity of the Coordinate Transformation Service and bi-directional scroll synchronization, a phased implementation is advisable. Initial deployment should focus on the core RAG Q&A functionality that provides text answers and page number references.[6] Subsequent phases can introduce the full geometric coordinate retrieval and visualization features (highlighting and scroll synchronization) once the foundational RAG pipeline is proven stable and reliable.

Conclusions

The Tulip AI Frontline Copilot and AI Composer represent an advanced application of Retrieval-Augmented Generation (RAG) technology, moving beyond simple chatbot Q&A into the realm of structured workflow automation. The unique complexity lies in the seamless integration of LLM-driven schema generation with high-fidelity, coordinate-based visual grounding of the source material.

To replicate the core feature set—composable, editable forms synchronized with a scrolling, referenced PDF—requires a highly specialized technical stack:

- A **layout-aware RAG pipeline** that stores geometric bounding box coordinates in its vector database metadata.
- A **hybrid persistence layer (e.g., JSONB)** to manage the flexible schemas of the generated applications while maintaining transactional integrity.
- A **decoupled front-end architecture** that leverages dynamic form renderers and implements a custom, client-side **Coordinate Transformation Service** to accurately translate stored PDF points into actionable screen pixel coordinates for highlighting and scroll operations.
- **Pre-grounding strategies** that embed coordinate metadata directly into the form schema at the time of creation, significantly reducing the latency of reference lookups during active use.

The successful implementation of such an architecture provides a powerful competitive advantage by delivering high-trust, grounded, and composable operational guidance directly to the frontline workforce.

Works cited

1. Live Demo: AI Co-pilot & Composer for Manufacturing, accessed November 18, 2025, https://www.frontlineoperations.ai/post/live-demo-ai-co-pilot-composer-for-manufacturing
2. Announcing Frontline Copilot™ – AI Tools in Tulip To Assist... | Tulip, accessed November 18, 2025, https://tulip.co/blog/announcing-frontline-copilot/
3. AI Copilot for Manufacturing: Enhancing Operations with Artificial Intelligence - Tulip Co, accessed November 18, 2025, https://tulip.co/blog/ai-manufacturing-copilot/
4. Tulip Frontline Operations Platform - Microsoft Marketplace, accessed November 18, 2025, https://marketplace.microsoft.com/zh-cn/product/web-apps/tulipinterfacesinc.off

er_frontline_operations?tab=overview
5. What Is a Composable Tech Stack and How Do You Build One? - Webstacks, accessed November 18, 2025, https://www.webstacks.com/blog/composable-tech-stack
6. Document Q&A with Copilot - Tulip Knowledge Base, accessed November 18, 2025, https://support.tulip.co/docs/document-qa-with-copilot
7. What is RAG? - Retrieval-Augmented Generation AI Explained - AWS, accessed November 18, 2025, https://aws.amazon.com/what-is/retrieval-augmented-generation/
8. What is Retrieval Augmented Generation (RAG)? - Databricks, accessed November 18, 2025, https://www.databricks.com/glossary/retrieval-augmented-generation-rag
9. Transform PDFs into Interactive Apps: Tulips AI Composer Demo - YouTube, accessed November 18, 2025, https://www.youtube.com/watch?v=53E3Cjq9yYA
10. Introducing AI Composer: Instantly Turn SOPs into Interactive Apps - Tulip Co, accessed November 18, 2025, https://tulip.co/blog/introducing-ai-composer/
11. Tulip AI composer, accessed November 18, 2025, https://support.tulip.co/docs/tulip-ai-composer
12. Build Apps from SOPs with AI Composer – Now with Templates - Tulip Co, accessed November 18, 2025, https://tulip.co/blog/build-apps-from-sops-with-ai-composer-now-with-templates/
13. 25 chunking tricks for RAG that devs actually use | by
14. Parse and chunk documents | Vertex AI Search - Google Cloud Documentation, accessed November 18, 2025, https://docs.cloud.google.com/generative-ai-app-builder/docs/parse-chunk-documents
15. Best PDF Parsers for RAG Applications - DEV Community, accessed November 18, 2025, https://dev.to/biomathcode/best-pdf-parsers-for-rag-applications-26j8
16. RAG technique using Document Extraction API. | LandingAI Community, accessed November 18, 2025, https://community.landing.ai/c/questions-insights-ade/rag-technique-using-document-extraction-api
17. Implement RAG chunking strategies with LangChain and watsonx.ai - IBM, accessed November 18, 2025, https://www.ibm.com/think/tutorials/chunking-strategies-for-rag-with-langchain-watsonx-ai
18. Mastering Chunking Strategies for RAG: Best Practices & Code Examples - Databricks Community, accessed November 18, 2025, https://community.databricks.com/t5/technical-blog/the-ultimate-guide-to-chunking-strategies-for-rag-applications/ba-p/113089
19. 5 Best Embedding Models for RAG: How to Choose the Right One - GreenNode, accessed November 18, 2025, https://greennode.ai/blog/best-embedding-models-for-rag
20. Develop a RAG Solution - Generate Embeddings Phase - Azure Architecture

Center | Microsoft Learn, accessed November 18, 2025, https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/rag/rag-generate-embeddings
21. How to Find and Highlight Text in PDF Using JavaScript | by MESCIUS inc. - Medium, accessed November 18, 2025, https://medium.com/mesciusinc/how-to-find-and-highlight-text-in-pdf-using-javascript-a56ab7b95230
22. How to use PDF.js to highlight text programmatically - Nutrient SDK, accessed November 18, 2025, https://www.nutrient.io/blog/how-to-add-highlight-annotations-to-pdfs-in-javascript/
23. Designing a Custom Chatbot with RAG and Function Calling, accessed November 18, 2025, https://community.openai.com/t/designing-a-custom-chatbot-with-rag-and-function-calling/1098582
24. Doing RAG on PDFs using File Search in the Responses API - OpenAI Cookbook, accessed November 18, 2025, https://cookbook.openai.com/examples/file_search_responses
25. What is RAG (Retrieval Augmented Generation)? - IBM, accessed November 18, 2025, https://www.ibm.com/think/topics/retrieval-augmented-generation
26. What is Retrieval-Augmented Generation (RAG)? A Practical Guide - K2view, accessed November 18, 2025, https://www.k2view.com/what-is-retrieval-augmented-generation
27. The Form.io Core Is An Open Source Form Builder, accessed November 18, 2025, https://form.io/open-source/
28. SurveyJS: Survey and Form Management Software, accessed November 18, 2025, https://surveyjs.io/
29. Embeddable PDF Viewer Guide, accessed November 18, 2025, https://embeddable.co/blog/embeddable-pdf-viewer-guide
30. Smooth Scrolling - CSS-Tricks, accessed November 18, 2025, https://css-tricks.com/snippets/jquery/smooth-scrolling/
31. synchronise scrolling between 2 divs in native javascript (2020) - Stack Overflow, accessed November 18, 2025, https://stackoverflow.com/questions/65117713/synchronise-scrolling-between-2-divs-in-native-javascript-2020
32. Comparing Postgres JSONB With NoSQL Databases | Learn More - Couchbase, accessed November 18, 2025, https://www.couchbase.com/blog/postgres-jsonb-and-nosql/
33. Bridging the Gap Between SQL and NoSQL in PostgreSQL with JSON - DbVisualizer, accessed November 18, 2025, https://www.dbvis.com/thetable/bridging-the-gap-between-sql-and-nosql-in-postgresql-with-json/
34. Working with nested data types using Amazon Redshift Spectrum | AWS Big Data Blog, accessed November 18, 2025, https://aws.amazon.com/blogs/big-data/working-with-nested-data-types-using-

amazon-redshift-spectrum/
35. Model Once, Represent Everywhere: UDA (Unified Data Architecture) at Netflix, accessed November 18, 2025, https://netflixtechblog.com/model-once-represent-everywhere-uda-unified-data-architecture-at-netflix-6a6aee261d8d
36. Data persistence - Android Basics with Compose - Android Developers, accessed November 18, 2025, https://developer.android.com/courses/android-basics-compose/unit-6
37. Persist data with Room - Android Developers, accessed November 18, 2025, https://developer.android.com/codelabs/basic-android-kotlin-compose-persisting-data-room