Production-Mirror RAG System: Implementation Steps

Project Overview

Build a production-grade, topic-agnostic RAG (Retrieval-Augmented Generation) system on AWS that processes documents (PDFs and screenshots), enables semantic search, and generates formatted reports with citations.

Estimated Timeline: 6 weeks **Estimated Cost:** ~$48/month for POC

Phase 1: Foundation & Setup

Step 1: AWS Account & Region Setup

- Create or access AWS account
- Choose AWS region (us-east-1 recommended for Bedrock availability)
- Navigate to AWS Bedrock console
- Request model access for Amazon Titan Embeddings V2 and Claude 3 Haiku
- Wait for model access approval
- Verify model access in Bedrock console
- Install AWS CLI locally
- Configure AWS credentials
- Test AWS access

Step 2: Terraform Backend Configuration

- Create S3 bucket for Terraform state (manual or bootstrap script)
- Enable versioning on state bucket
- Enable encryption on state bucket
- Block all public access on state bucket
- Create DynamoDB table for state locking
- Create backend.tf file with S3 and DynamoDB configuration
- Test backend configuration with terraform init

Step 3: Project Structure Setup

- Create project directory structure (terraform/, lambda/, frontend/, docs/, scripts/)
- Create subdirectories for Terraform modules (networking/, database/, lambda/, api/, storage/, monitoring/)
- Create subdirectories for Lambda functions (textract_processor/, embedding_processor/, query_handler/)
- Initialize git repository
- Create .gitignore file (exclude Terraform state, secrets, build artifacts)
- Create initial README.md
- Make first git commit

Phase 2: Networking Infrastructure

Step 4: VPC and Networking (Terraform)

- Create VPC with CIDR block 10.0.0.0/16
- Enable DNS hostnames and DNS support
- Create 2 public subnets in different availability zones
- Create 2 private subnets for database in different availability zones
- Create 2 private subnets for Lambda in different availability zones
- Create Internet Gateway
- Create NAT Gateway in public subnet (1 for cost optimization)
- Allocate Elastic IP for NAT Gateway
- Create public route table with route to Internet Gateway
- Create private route table with route to NAT Gateway
- Associate subnets with appropriate route tables
- Enable VPC Flow Logs to CloudWatch or S3
- Create outputs for VPC ID, subnet IDs, and NAT Gateway ID

Step 5: Security Groups (Terraform)

- Create security group for Lambda functions
- Add outbound rule: HTTPS (443) to 0.0.0.0/0 for Bedrock/S3/Textract access

- Add outbound rule: PostgreSQL (5432) to database security group
- Create security group for database
- Add inbound rule: PostgreSQL (5432) from Lambda security group only
- Create security group for VPC endpoints (if using)
- Add descriptive tags to all security groups
- Output security group IDs

Phase 3: Storage Infrastructure

Step 6: S3 Buckets (Terraform)

- Create upload bucket for user files (PDFs and images)
- Enable versioning on upload bucket
- Enable encryption (AES256 or KMS)
- Block all public access on upload bucket
- Configure CORS for web uploads
- Add lifecycle policy to archive to Glacier after 90 days (optional)
- Create processed bucket for extracted text
- Enable versioning, encryption, block public access on processed bucket
- Add lifecycle policy to delete after 7 days
- Create reports bucket for generated PDF reports
- Enable versioning, encryption, block public access on reports bucket
- Add lifecycle policy to delete after 30 days
- Create website bucket for frontend
- Enable static website hosting
- Configure index.html and error.html
- Add bucket policy for public read access (website content only)
- Enable S3 access logging for all buckets
- Add comprehensive tags to all buckets

- Output bucket names and ARNs

Step 7: Database Setup - Aurora PostgreSQL (Terraform)

- Create DB subnet group with both private database subnets
- Create Aurora Serverless v2 cluster
- Configure engine: aurora-postgresql, version 15.3 or later
- Set database name, master username
- Configure Serverless v2 scaling: min 0.5 ACU, max 2 ACU
- Enable Multi-AZ deployment (2 instances across availability zones)
- Assign VPC and DB subnet group
- Assign database security group
- Enable encryption at rest with KMS key
- Configure backup retention: 7 days
- Set preferred backup window and maintenance window
- Enable Performance Insights (7-day retention)
- Enable Enhanced Monitoring (60-second granularity)
- Enable deletion protection
- Create database credentials in Secrets Manager
- Auto-generate password (32 characters)
- Enable automatic rotation (90 days)
- Output cluster endpoint, reader endpoint, secret ARN, database name

Step 8: Database Schema Setup - pgvector

- Connect to Aurora cluster using PostgreSQL client
- Install pgvector extension
- Create document_embeddings table with appropriate columns
- Define embedding column as vector type (1024 dimensions for Titan V2)
- Include metadata columns: document_id, document_name, chunk_index, chunk_text, source_file_type, original_file_s3_path, upload_date, metadata JSONB

- Create indexes: document_id, upload_date, file_type
- Create HNSW vector index for cosine similarity search
- Create optional documents table for additional metadata
- Create helper function for similarity search with filters
- Grant appropriate permissions to application user (if using separate user)
- Test schema with dummy data
- Verify extension installed, tables created, indexes created
- Test helper function with sample queries

Phase 4: Security & Secrets Management

Step 9: KMS Keys and Secrets (Terraform)

- Create KMS key for application encryption
- Set key alias and description
- Enable automatic key rotation (annual)
- Define key policy allowing root account and specific IAM roles
- Create KMS key for database (if not using default)
- Update Secrets Manager secret to use KMS key
- Create additional secrets for API keys and configuration
- Store secrets in Secrets Manager with KMS encryption
- Output KMS key IDs, ARNs, and secret ARNs
- Verify keys created in KMS console
- Test encryption/decryption with AWS CLI
- Verify secrets can be retrieved

Phase 5: Lambda Functions - Ingestion Pipeline

Step 10: IAM Roles for Lambda Functions (Terraform)

- Create IAM role for Textract Lambda with trust relationship for lambda.amazonaws.com

- Attach AWSLambdaVPCAccessExecutionRole managed policy
- Create inline policy with permissions for: S3 read (upload bucket), S3 write (processed bucket), Textract invoke, CloudWatch Logs, SQS (DLQ), X-Ray, KMS decrypt
- Create IAM role for Embedding Lambda
- Attach VPC execution policy
- Create inline policy with permissions for: S3 read (processed and upload buckets), Bedrock InvokeModel, Secrets Manager GetSecretValue, CloudWatch Logs, SQS (DLQ), X-Ray, KMS decrypt
- Create IAM role for Query Lambda
- Attach VPC execution policy
- Create inline policy with permissions for: Bedrock InvokeModel (both models), Secrets Manager, S3 read (upload bucket), S3 write (reports bucket), CloudWatch Logs, SQS (DLQ), X-Ray, KMS decrypt
- Output all role ARNs
- Verify roles created in IAM console
- Use IAM Policy Simulator to test permissions

Step 11: Lambda 1 - Textract Processor (Python Code)

- Create handler.py file for Textract processing
- Implement Lambda handler function
- Parse S3 event to get bucket and file key
- Detect file type (.pdf, .png, .jpg, .jpeg) from file extension
- Call Textract API (DetectDocumentText for simple extraction)
- Extract text from Textract response (concatenate LINE blocks)
- Create metadata dictionary with original S3 path, file type, upload date, processing date
- Save extracted text to processed bucket with metadata as JSON
- Implement comprehensive error handling (unsupported files, Textract errors, S3 errors)
- Add structured CloudWatch logging with context
- Create requirements.txt with boto3
- Test locally with sample PDF and image files

Step 12: Deploy Lambda 1 - Textract Processor (Terraform)

- Package Lambda code into ZIP file
- Create Lambda function resource in Terraform
- Set function name, runtime (Python 3.11), handler, timeout (90s), memory (512MB)
- Assign IAM role created in Step 10
- Configure environment variables (processed bucket, upload bucket, log level)
- Configure VPC settings (subnet IDs, security group ID)
- Configure dead letter queue (SQS)
- Enable X-Ray tracing
- Set reserved concurrent executions to 5
- Create Dead Letter Queue (SQS) with 14-day retention
- Create CloudWatch Log Group with 7-day retention
- Create Lambda permission for S3 to invoke function
- Add S3 event notifications on upload bucket for .pdf, .png, .jpg, .jpeg files
- Output Lambda ARN and function name
- Deploy with terraform apply
- Upload test PDF to S3 and verify Lambda triggers
- Check CloudWatch logs for execution
- Verify text file created in processed bucket
- Test with PNG/JPG image
- Test error cases

Step 13: Lambda 2 - Embedding Processor (Python Code)

- Create handler.py file for embedding processing
- Implement Lambda handler function
- Parse S3 event to get processed text file
- Read text file and metadata from processed bucket
- Implement text chunking function (500 characters with 50 character overlap)

- Implement function to call Bedrock Titan Embeddings API
- Implement database connection function (retrieve credentials from Secrets Manager)
- Implement function to store embeddings in PostgreSQL with batch insert
- Connect to database and prepare INSERT statement
- For each chunk: generate embedding and prepare data tuple
- Execute batch insert of all chunks with metadata
- Commit transaction and close connection
- Add comprehensive error handling and retries
- Add structured logging with context
- Create requirements.txt with boto3 and psycopg2-binary
- Test locally with sample text file

Step 14: Deploy Lambda 2 - Embedding Processor (Terraform)

- Package Lambda code into ZIP file (consider Lambda Layer for psycopg2 if package is large)
- Optionally create Lambda Layer with psycopg2-binary
- Create Lambda function resource in Terraform
- Set function name, runtime, handler, timeout (180s), memory (1024MB)
- Assign IAM role
- Configure environment variables (processed bucket, upload bucket, DB secret ARN, Bedrock model ID, chunk size, chunk overlap)
- Configure VPC settings
- Configure dead letter queue
- Enable X-Ray tracing
- Set reserved concurrent executions to 5
- Attach Lambda Layer if using
- Create Dead Letter Queue (SQS)
- Create CloudWatch Log Group with 7-day retention
- Create Lambda permission for S3 to invoke function

- Add S3 event notification on processed bucket for .json files
- Output Lambda ARN
- Deploy with terraform apply
- Upload PDF to test complete pipeline
- Check embedding Lambda logs
- Query database to verify embeddings stored
- Verify vector dimensions (1024 for Titan V2)
- Test with multiple files
- Monitor Lambda duration and memory usage

Step 15: Test Complete Ingestion Pipeline

- Prepare test documents (3 PDFs, 2 screenshots, 1 corrupted file)
- Upload PDF to S3 upload bucket
- Monitor CloudWatch Logs for both Lambda functions
- Verify text extracted successfully
- Verify embeddings generated and stored
- Query database to check document count and chunk count
- Upload remaining test files one by one
- For each upload verify both Lambdas executed, processed file created, embeddings in database, metadata correct
- Test screenshot processing (verify OCR, file type, S3 path stored)
- Test error case with corrupted file (verify error logged, check DLQ)
- Perform performance testing (note Lambda duration, check memory usage, identify cold starts)
- Verify database data integrity with SQL queries
- Validate success criteria: all PDFs processed, all screenshots processed, correct chunk counts, embeddings are 1024-dimension vectors, metadata populated, vector search works, reasonable execution times, no errors in logs, DLQ empty

---

Phase 6: Query Pipeline - Lambda 3

Step 16: Lambda 3 - Query Handler Part 1: Intent Extraction (Python Code)

- Create handler.py file for query handling
- Implement intent extraction function using Claude
- Create system prompt for structured filter extraction (JSON output)
- Define output format: topic, file_filter, date_from, date_to, file_type
- Include examples in prompt for few-shot learning
- Set low temperature (0.1) for consistent structured output
- Call Bedrock with Claude Haiku
- Parse JSON response
- Implement fallback for malformed JSON
- Add validation for extracted filters (sanitize file names, validate dates)
- Test with various query types

Step 17: Lambda 3 - Query Handler Part 2: Vector Search (Python Code)

- Implement database connection function with credential caching
- Implement function to generate query embedding using Bedrock Titan
- Implement document search function with filters
- Build dynamic SQL query based on available filters
- Use parameterized queries to prevent SQL injection
- Apply filters for file name (ILIKE), date range, file type
- Filter by similarity threshold
- Order by vector distance and limit to top K results
- Generate pre-signed S3 URLs for image sources (1-hour expiry)
- Return list of matching chunks with metadata and image URLs
- Add connection reuse across invocations
- Implement error handling for database failures and no results

Step 18: Lambda 3 - Query Handler Part 3: Report Generation (Python Code)

- Implement report generation function using Claude
- Build context from retrieved chunks (include source citations)
- Create system prompt for report structure (Executive Summary, Key Findings, Detailed Analysis, Sources, Confidence Assessment)
- Include instructions for inline citations
- Set temperature to 0.3 for factual but readable output
- Call Bedrock with Claude Haiku
- Implement response formatting function
- Separate image sources for frontend display
- Build metadata object with timestamp, filters, processing time
- Format sources list with document names, file types, similarity scores
- Implement main Lambda handler to orchestrate all three stages
- Parse API Gateway event and extract query
- Call intent extraction
- Generate query embedding
- Search documents with filters
- Handle no results case gracefully
- Generate report from retrieved chunks
- Calculate processing time
- Format final JSON response with CORS headers
- Add comprehensive error handling and logging
- Create requirements.txt
- Test locally with mock events

Step 19: Deploy Lambda 3 - Query Handler (Terraform)

- Package Lambda code into ZIP file
- Create Lambda function resource in Terraform
- Set function name, runtime, handler, timeout (300s), memory (2048MB)

- Assign IAM role
- Configure environment variables (DB secret ARN, Bedrock model IDs, upload bucket, top K, similarity threshold)
- Configure VPC settings
- Configure dead letter queue
- Enable X-Ray tracing
- Set reserved concurrent executions to 10 (user-facing)
- Create Dead Letter Queue
- Create CloudWatch Log Group with 7-day retention
- Output Lambda ARN and function name
- Deploy with terraform apply
- Invoke directly with test event using AWS CLI or Console
- Verify all three stages execute
- Check CloudWatch Logs for execution flow
- Measure end-to-end latency
- Test error handling cases

Phase 7: API Layer

Step 20: API Gateway Setup (Terraform)

- Create HTTP API in API Gateway
- Configure CORS (allow origins, methods, headers, max age)
- Create default stage with auto-deploy
- Configure access logging to CloudWatch
- Set default route throttling (100 burst, 50 steady-state)
- Create CloudWatch Log Group for API logs with 7-day retention
- Create Lambda integration for query endpoint (AWS_PROXY type)
- Set integration timeout to 30 seconds

- Create route: POST /query
- Create Lambda permission for API Gateway to invoke query Lambda
- Create upload handler Lambda (simple function to generate pre-signed S3 URLs)
- Create Lambda integration for upload endpoint
- Create route: POST /upload
- Create Lambda permission for API Gateway to invoke upload Lambda
- Output API endpoint URL
- Deploy with terraform apply
- Test query endpoint with curl
- Test upload endpoint with curl
- Use returned pre-signed URL to upload file
- Verify CORS headers in response
- Check API Gateway logs in CloudWatch

Step 21: API Authentication (Terraform)

- For POC, implement simple API key authentication
- Create Lambda authorizer function (checks X-Api-Key header)
- Create IAM role for authorizer Lambda
- Deploy authorizer Lambda
- Create API Gateway authorizer resource (REQUEST type, identity source: header)
- Attach authorizer to query and upload routes
- Store API key in Secrets Manager
- Use random password generator for secure key
- Output API key ARN (mark as sensitive)
- Alternative: Skip authentication for POC and document as limitation
- Test authentication with valid and invalid API keys
- Verify unauthorized requests are blocked

Phase 8: Monitoring & Observability

Step 22: CloudWatch Dashboards and Alarms (Terraform)

- Create main CloudWatch dashboard
- Add widget for Lambda invocations (all three functions)
- Add widget for Lambda duration averages
- Add widget for Lambda errors and throttles
- Add widget for API Gateway requests and errors
- Add widget for Aurora database metrics (CPU, connections)
- Create SNS topic for alerts
- Create SNS email subscription (requires email confirmation)
- Create alarm for Lambda errors (threshold: 5 errors in 5 minutes) for each Lambda
- Create alarm for Lambda duration approaching timeout (query Lambda)
- Create alarm for API Gateway 5xx errors (threshold: 10 errors)
- Create alarm for database high CPU (threshold: 80%)
- Create alarm for NAT Gateway high data transfer (cost control)
- Create AWS Budget for monthly spending (threshold: $50, alert at 80% and 100%)
- All alarms send notifications to SNS topic
- X-Ray tracing already enabled in Lambda functions
- Deploy with terraform apply
- Confirm SNS email subscription
- View dashboard in CloudWatch console
- Trigger test alarm by temporarily lowering threshold
- Verify alert emails received
- Generate traffic and view X-Ray traces

Step 23: End-to-End Testing

- Prepare 50 test documents (30-35 PDFs across various domains, 15-20 screenshots)
- Ensure diversity: industries, file sizes, page counts, date ranges

- Upload all documents via API using script or manually
- Monitor CloudWatch Logs for both ingestion Lambdas
- Verify all files processed successfully
- Check for any DLQ messages (should be zero)
- Query database for total document and chunk counts
- Test diverse queries: basic (no filters), file-specific, date-filtered, image-specific, topic-agnostic (cross-domain)
- Validate query responses: report structure, citations, image URLs, similarity scores, processing time, metadata
- Run performance testing with 10 concurrent queries
- Monitor Lambda concurrency in CloudWatch
- Check for throttling errors
- Verify no database connection issues
- Measure P50, P90, P99 latencies
- Test error handling: empty query, very long text, special characters, unsupported file type, oversized file, corrupted PDF
- Verify graceful error messages and system stability
- Validate data integrity in database with SQL queries (check for orphaned chunks, null embeddings, duplicates, incorrect dimensions)
- Analyze costs during testing in AWS Cost Explorer
- Calculate cost per document processed and cost per query
- Project monthly cost for full usage
- Document all issues found and resolutions
- Verify success criteria met

Step 24: Security Hardening

- Review all Lambda IAM roles for least privilege
- Verify no wildcard permissions where avoidable
- Use IAM Access Analyzer to identify issues

- Verify database credentials in Secrets Manager (not environment variables)
- Verify API keys in Secrets Manager
- Enable automatic rotation for database credentials
- Verify KMS encryption enabled on all secrets
- Check secret access logs in CloudTrail
- Verify all public S3 buckets are intentional (only website bucket)
- Verify database in private subnets only
- Review security group rules (no unnecessary 0.0.0.0/0 inbound)
- Verify VPC Flow Logs enabled
- Check NACLs for proper configuration
- Verify S3 bucket settings: Block Public Access (all except website), Versioning, Encryption, Access Logging
- Run S3 bucket policy analyzer
- Check for public objects
- Verify CORS configuration restrictive
- Verify database encryption at rest and in transit
- Review database user permissions
- Verify no public database accessibility
- Review CORS policy on API Gateway
- Verify rate limiting/throttling configured
- Consider WAF for production
- Verify API authentication working
- Test for injection vulnerabilities
- Verify Lambda functions in VPC
- Review Lambda environment variables (no secrets)
- Verify CloudTrail enabled and logging
- Verify log file validation enabled
- Check encryption status on all resources using AWS CLI

- Scan Lambda dependencies for vulnerabilities
- Update packages to latest secure versions
- Verify input validation in Lambda functions
- Check for SQL injection prevention (parameterized queries)
- Document security posture: controls implemented, known risks, mitigation strategies

---

Phase 9: Documentation & Polish

Step 25: Architecture Documentation

- Create architecture diagram using draw.io, Lucidchart, or CloudCraft
- Include all components: User, Web UI, API Gateway, Lambda Functions (3), VPC, Aurora, Bedrock, S3 buckets (3), Textract, CloudWatch, NAT Gateway
- Show data flow for ingestion and query paths
- Indicate security boundaries (VPC, subnets)
- Label all components with AWS service names
- Update README.md with: project overview, features, architecture diagram, technology stack, prerequisites, setup instructions, usage examples, API documentation link, cost estimate, monitoring info, security summary, known limitations, future enhancements, troubleshooting link
- Create API Documentation with: base URL, authentication, endpoint specifications (request/response examples), error codes
- Create Troubleshooting Guide with common issues and solutions
- Create deployment guide for team members
- Document maintenance procedures: updating Lambda code, adding documents, querying logs, responding to alarms, backup/restore
- Verify all documentation is complete, accurate, and tested by someone else

Step 26: Cost Optimization Review

- Review actual costs in AWS Cost Explorer
- Compare to estimates and identify unexpected costs
- Review CloudWatch metrics for Lambda memory usage

- Right-size Lambda memory allocations based on actual usage
- Check for excessive cold starts
- Evaluate Provisioned Concurrency if needed
- Check actual Aurora ACU usage
- Verify auto-pause working (if applicable)
- Review database query performance
- Consider index optimization
- Evaluate Multi-AZ necessity for POC
- Implement S3 lifecycle policies: delete old processed files after 7 days, archive old reports to Glacier after 30 days, delete reports after 90 days
- Evaluate NAT Gateway alternatives (VPC Endpoints for S3 and Bedrock)
- Verify 7-day log retention on all CloudWatch log groups
- Consider exporting old logs to S3
- Review Bedrock token usage patterns
- Optimize prompts for token efficiency
- Consider caching frequent queries
- Verify AWS Budgets alerts working
- Tag all resources for cost allocation
- Enable Cost Anomaly Detection
- Document cost optimization opportunities and savings
- Update final monthly cost projection

Phase 10: Web UI (Final Step)

Step 27: Frontend Development

- Create index.html with structure: header, upload section, query section, results section
- Add file input accepting .pdf, .png, .jpg, .jpeg with multiple file support
- Add upload button and status display

- Add query textarea and search button
- Add results section with report display, download PDF button, image gallery, sources info
- Include external libraries: marked.js for markdown rendering, html2pdf.js for PDF generation
- Create styles.css with modern, responsive design
- Style with gradient backgrounds, clean cards, hover effects, loading animations
- Ensure mobile responsiveness
- Create app.js with application logic
- Implement upload handler: get pre-signed URL from API, upload file to S3, show progress, handle errors
- Implement query handler: call query API endpoint, display loading state, render report, show images, display sources
- Use marked.js to convert markdown report to HTML
- Display image gallery for screenshots with clickable thumbnails
- Display sources list with similarity scores
- Implement PDF download with html2pdf.js
- Add status message helper function
- Add keyboard shortcut (Enter to search)
- Create config.js.example template for API endpoint and API key
- Test on Chrome, Firefox, Safari
- Test on mobile devices
- Test file upload (multiple files)
- Test various queries
- Test PDF download
- Test image viewing
- Verify CORS working
- Check browser console for errors

Step 28: Deploy Frontend (Terraform)

- Update API endpoint and API key in app.js

- Create website.tf in storage module
- Create S3 bucket for website
- Configure static website hosting (index.html, error.html)
- Configure public access block to allow public reads
- Create bucket policy for public read access to website content
- Upload HTML, CSS, and JS files to S3 (manually or via Terraform)
- If using Terraform, create S3 object resources for each file type with correct content types
- Optional: Create CloudFront distribution for HTTPS and better performance
- Configure CloudFront with S3 origin, caching behavior, HTTPS redirect
- Output website URL (S3 endpoint or CloudFront domain)
- Deploy with terraform apply
- Open website URL in browser
- Test all functionality (upload, query, report display, PDF download, images)
- Verify API calls working
- Check browser console for errors

Step 29: Final Demo Preparation

- Organize 50 demo documents by category
- Create spreadsheet documenting each file
- Upload all documents via UI
- Prepare 10-15 demo queries showcasing various features and domains
- Create demo script with timing: introduction, upload demo, query demo, topic-agnostic demo, technical deep dive, cost & scale, Q&A
- Prepare backup demo: screen recordings, screenshots
- Practice entire demo flow and time each section
- Identify potential issues and troubleshooting steps
- Prepare supporting materials: architecture diagram, cost breakdown, technical specs, future roadmap
- Final system check: all documents uploaded, database populated, API working, website accessible, monitoring dashboard green, no alarms, recent queries working, PDF download working, images displaying

- Prepare demo environment: stable internet, browser tabs ready, credentials accessible, demo queries in text file, screen sharing tested, audio tested, backup plan ready

---

Project Completion Checklist

Infrastructure

- All VPC and networking resources deployed
- All S3 buckets created and configured
- Aurora PostgreSQL database running with pgvector
- All Lambda functions deployed and working
- API Gateway configured and accessible
- CloudWatch monitoring active
- X-Ray tracing enabled
- SNS alerts configured and tested

Security

- All credentials in Secrets Manager
- KMS encryption enabled on all applicable resources
- Security groups properly configured
- IAM roles follow least privilege
- S3 buckets properly secured
- VPC Flow Logs enabled
- CloudTrail logging enabled

Functionality

- Document upload working for all file types
- OCR text extraction working
- Vector embeddings generated correctly
- Database storing all data properly
- Query pipeline functioning end-to-end

- Intent extraction accurate
- Vector search returning relevant results
- Report generation working with citations
- Image URLs working and accessible
- PDF download working

Testing

- All 50 documents processed successfully
- Multiple test queries executed successfully
- Error handling tested and working
- Performance acceptable
- Cross-domain queries working
- Screenshot queries working

Documentation

- README complete and accurate
- Architecture diagram created
- API documentation written
- Setup instructions tested
- Troubleshooting guide created
- Security documentation complete
- Cost analysis documented

Demo Ready

- Demo script prepared and practiced
- Test queries ready
- System stable and tested
- Backup plan prepared
- Supporting materials ready

Success Metrics

- System uptime: 99%+
- Query response time: <10 seconds average
- Document processing time: <2 minutes per document
- Error rate: <1%
- Documents processed: 50
- Successful queries: >95%
- Cost efficiency: Within $50/month budget

Estimated Costs

Monthly (POC with 50 documents)

- VPC & NAT Gateway: $33.00
- Aurora PostgreSQL (Multi-AZ): $11.00
- Lambda: $0.10
- Bedrock: $0.10
- Textract: $0.08
- S3: $0.02
- Secrets Manager: $0.40
- KMS: $1.00
- CloudWatch: $1.50
- **Total: ~$48/month**

Production Scaling Estimate

- 1,000 documents, 1,000 queries/month: $200-300/month
- 10,000 documents, 10,000 queries/month: $600-800/month

Next Steps After Completion

Phase 2 (Production Ready)

- Add second NAT Gateway (Multi-AZ HA)
- Implement WAF for API protection
- Add Cognito user authentication
- Custom domain with Route53
- Enhanced monitoring and dashboards
- Automated testing pipeline
- CI/CD pipeline

Phase 3 (Advanced Features)

- Document versioning
- User-specific document collections
- Advanced chunking strategies
- Multi-modal embeddings
- Query caching
- Analytics dashboard
- Feedback loop for relevance tuning
- Export to multiple formats

Phase 4 (Scale)

- Multi-region deployment
- CDN for global performance
- Elasticsearch for hybrid search
- Fine-tuned embedding models
- Batch processing pipelines
- Data warehousing for analytics