

Solo Vault: Cloud-Native Retrieval Infrastructure for the Solo IDE

Sachin Adlakha, Salauat Kakimzhanov, Krish Kuchroo, Michael Wang

Computer Science Department, New York University

{sa9082, sk12953, kk5846, mzw2010}@nyu.edu

Abstract—Modern software development increasingly relies on large language models (LLMs) for code generation, and retrieval quality depends on access to the developer's own files through Retrieval-Augmented Generation (RAG) or the Model Context Protocol (MCP). Solo is a desktop Integrated Development Environment (IDE) that ships a built-in vault providing offline keyword and semantic search over files dropped into the workspace. The local design works well on a single machine but cannot share state across devices, scale beyond the capabilities of a laptop, or support shared retrieval across teams. We present Solo Vault, an AWS serverless backend that extends this vault into a cloud-native sync and retrieval service while preserving exact schema parity with the desktop store, so cloud rows act as direct sync targets rather than transformations. The system is deployed across twelve declarative CloudFormation stacks, driven by a TypeScript orchestration layer, with strict per-user data isolation, dual customer-managed KMS encryption, and an event-driven five-stage indexing pipeline that streams progress to the IDE in real time. We organize the discussion around three design axes: schema parity between the local SQLite store and its cloud mirror, hybrid keyword and vector retrieval matching the IDE's existing offline behavior, and the engineering trade-offs across the deployment toolchain and security posture.

Index Terms—Cloud computing, serverless architecture, retrieval-augmented generation, vector databases, AWS, PostgreSQL, pgvector, hybrid search, infrastructure as code, knowledge management.

I. PROBLEM STATEMENT

Solo's local vault is fast and private, but its design limits usage to a single user on one machine. The SQLite index and blob store exists exclusively on the disk of the device where the files were added, with no replication or synchronization. It does not support multi-device access or shared retrieval among collaborators, meaning vault data is permanently restricted to the machine on which it was created.

Local indexing is bound by the user's CPU, memory, and disk. File size is capped at 50 MB, and the indexed knowledge is unrecoverable in the event of disk failure. The agent's retrieval surface at any moment is exactly what the user has active in the open workspace; anything outside that boundary is invisible.

Adding a cloud backend resolves these limitations but introduces problems that must be addressed carefully. The cloud schema must be compatible with the local store and must provide synchronization capabilities to ensure data consistency across devices. Multi-tenant data isolation, authenticated access, durable storage, and a retry-tolerant indexing pipeline must all be functional without affecting existing offline behavior.

II. MOTIVATION

Software developers and knowledge workers increasingly rely on retrieval-augmented workflows. Finding the right document, code snippet, file, or

note—both for personal use and as context for AI assistants—has become a routine part of modern development. The volume of knowledge for any given workflow has grown significantly, yet most of it lives unindexed across cloud drives, chat histories, and local filesystems, making retrieval slow and tedious.

Since most code development happens within an IDE, the natural place to surface retrieval is within the IDE itself. Solo follows this approach, using the local disk to index and store a variety of data that is queried by both the user and AI agents integrated within the IDE. The local-first design offers low latency and strong privacy; files never leave the user's machine.

However, the problems with multi-device workflows, team-shared knowledge, and capped device capabilities make a supplemental cloud-based store that extends the local vault an evident necessity. Retrieval-augmented generation over a user's personal knowledge corpus has demonstrated meaningful improvements in the accuracy and relevance of LLM outputs [1], and a cloud-backed vault makes this corpus portable, durable, and queryable from any device or agent session.

III. EXISTING SOLUTIONS

Several existing platforms and tools support cloud-based retrieval for developers, each with distinct capabilities that illuminate the design space.

Cursor [2]: Cursor is an AI-augmented IDE known for its autocomplete feature and deep AI integrations. It delivers

retrieval over a codebase, documentation, and the web, but the index lives in Cursor's managed cloud, meaning offline use is not supported. Solo Vault preserves local data storage while adding cloud support, enabling operation in both directions.

Obsidian [3]: Obsidian is a local-first markdown note application whose search and semantic indexing runs on-device. The cloud store provides backups but is not directly queryable by external agents. Solo Vault guarantees the cloud is a live, queryable mirror of the local store, reachable from any authorized device or AI agent session.

Continue.dev [4]: Continue.dev is an open-source IDE assistant with local codebase indexing and optional cloud hosting. Solo Vault extends this concept to arbitrary file types—PDFs, images, spreadsheets, configuration files, and source code—with full multi-device support and team knowledge sharing.

None of these solutions simultaneously supports offline-first operation, direct cloud retrieval parity, multi-device access, and team-level knowledge sharing across arbitrary content types.

IV. SYSTEM ARCHITECTURE

A. Overview and Component Layout

Solo Vault's backend is organized as twelve independent CloudFormation stacks, each owning a clear slice of the system: shared-network, secrets, cognito, rds, storage, lambdas, api-gateway, sqs-pipeline, pipeline-lambdas, step-functions-pipeline, notifications-pipeline, and lambda-artifacts. Every stack publishes its outputs as CloudFormation exports so downstream stacks resolve dependencies via `Fn::ImportValue` rather than parameter passing. This decoupling ensures failures in any single stack do not cascade, and partial teardowns in development can target the failing stack in isolation.

Data flows through three logical planes. The synchronous plane handles authenticated REST calls from the desktop—reading, writing, and uploading vault entries through API Gateway and the vault-crud Lambda. The asynchronous plane processes uploaded files through a five-stage Step Functions pipeline that produces searchable chunks in PostgreSQL with pgvector. The notification plane feeds pipeline progress back to the IDE in real time via WebSocket. Figs. 1 and 2 illustrate these planes separately.

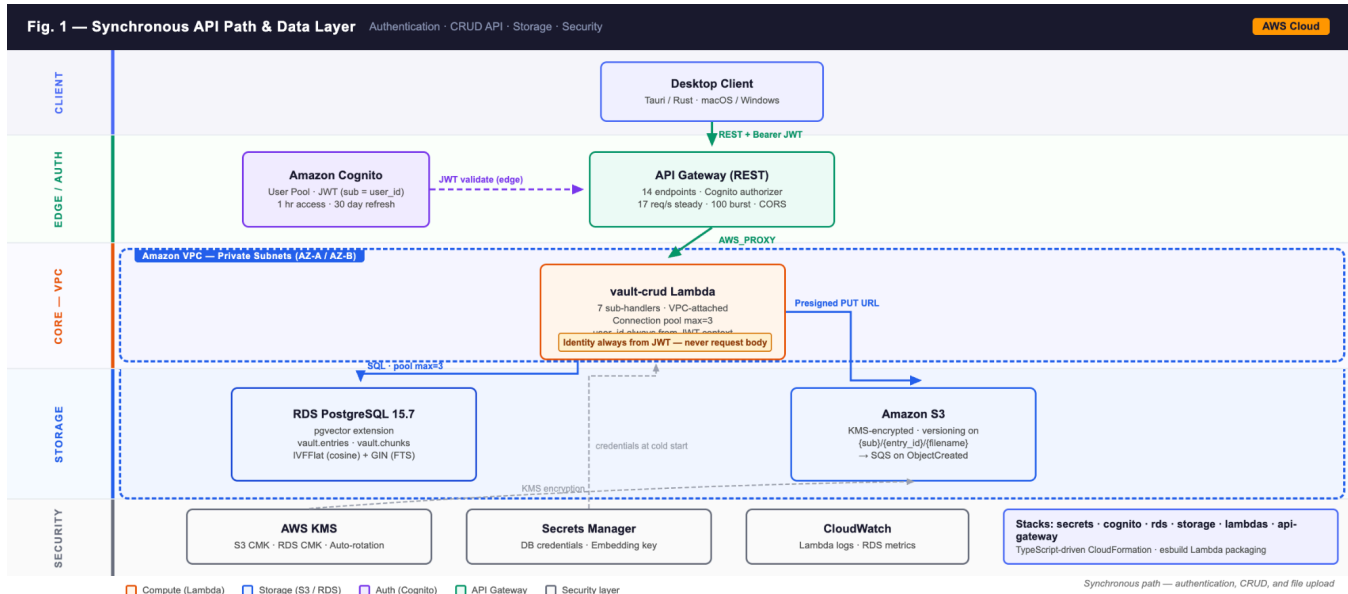


Fig. 1. Synchronous API path and data layer. The desktop client authenticates via Cognito, reaches vault-crud Lambda through API Gateway, and reads/writes to RDS PostgreSQL and S3. All compute touching the database runs inside private VPC subnets with no internet route. KMS provides independent encryption keys for S3 and RDS.

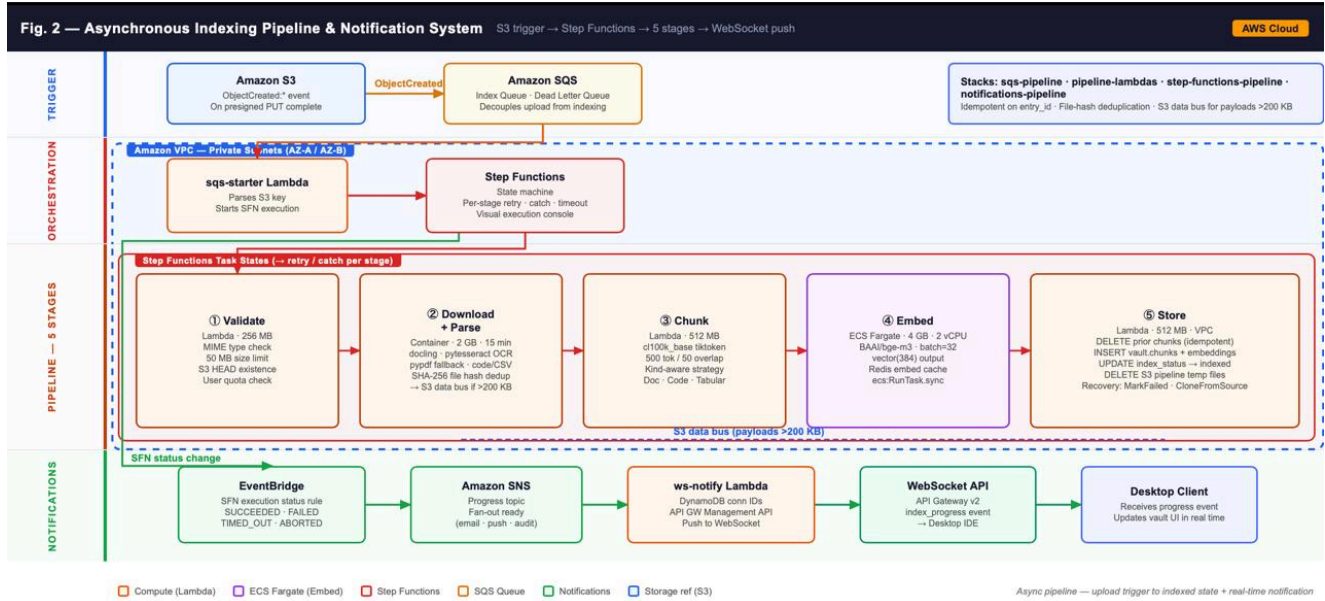


Fig. 2. Asynchronous indexing pipeline and notification system. An S3 ObjectCreated event triggers SQS, which launches a Step Functions execution across five Lambda/ECS stages (Validate → Download+Parse → Chunk → Embed → Store). Terminal state changes flow through EventBridge → SNS → ws-notify Lambda → WebSocket API back to the desktop IDE.

B. Identity and Per-User Isolation

Authentication is handled by an Amazon Cognito User Pool with email as the unique identifier and the Cognito sub-claim (a UUID) as the durable user ID used throughout the system. ID and access tokens are valid for one hour; refresh tokens for thirty days. The desktop authenticates out-of-band through federated identity providers—Google directly, and GitHub wrapped through a custom OIDC Lambda because GitHub is an OAuth 2.0 provider rather than a native OIDC provider.

Cognito is configured as a native authorizer on every API Gateway route. JWT signature, audience, expiry, and issuer are validated at the edge before any compute is billed. The vault-crud Lambda reads identity exclusively from the API Gateway authorizer context through a shared authentication module that extracts the Cognito sub-claim. Every SQL statement parameterizes user_id from the auth context, never from the HTTP request body, making cross-user data access structurally impossible regardless of handler bugs.

C. Data Layer

1) *Object Storage*: Amazon S3 is the durable store for vault file bytes. The bucket is encrypted with a customer-managed KMS key, has versioning enabled, and is locked behind a public-access-block. Object keys follow the strict per-user prefix {cognito_sub}/{entry_id}/{filename}, making per-user IAM scoping straightforward and preventing cross-user object access even in the event of an application-level bug. S3 emits ObjectCreated:* events to an SQS queue, forming the boundary between the synchronous upload path and the asynchronous indexing pipeline.

2) *Relational Store*: RDS runs PostgreSQL 15.7 with the pgvector extension on a db.t3.micro instance across two

availability zones. Encryption at rest uses a separate CMK from the S3 key, so the IAM principal authorized to decrypt the database differs from the one authorized to decrypt vault files. The security group accepts inbound traffic only from the Lambda security group on port 5432.

3) *Schema Design*: The schema is organized in a vault namespace and designed for strict parity with the local SQLite desktop store—cloud rows are sync targets, not transformations. The vault.entries table holds the full entry record with twenty-two columns including kind, subkind, scope_type, memory_type, cloud_sync_state, classifier_confidence, and millisecond-epoch timestamps. The single cloud-only addition on every table is user_id for per-user IDOR scoping.

The vault.chunks table stores flat content chunks with a 384-dimensional embedding column indexed using IVFFlat under cosine distance for approximate nearest-neighbor search. The vault.chunks_fts table is a materialized full-text search mirror indexed with GIN on a tsvector generated from chunk content. Hybrid retrieval executes a UNION across both indexes—dense vector cosine similarity for semantic queries and BM25-family tsvector matching for rare-token queries—providing complementary coverage that neither approach offers alone [5].

D. Synchronous API

1) *API Gateway*: A regional REST API in API Gateway fronts every authenticated call. Fourteen endpoints are defined and deployed in CloudFormation before Lambda implementations are complete, locking the URL surface so desktop development can proceed against a stable contract. Throttling is set at 17 req/s steady-state with a burst of 100.

2) *vault-crud Lambda*: vault-crud is the single Lambda backing all CRUD and upload routes. It runs inside the VPC

to access RDS. A dispatcher routes on HTTP method and API Gateway resource path to seven sub-handlers: list, create, get, update, delete, upload, and an idempotent ensureUser. Database access uses a module-level connection pool sized at max=3 so Lambda concurrency stays within the db.t3.micro connection limit. Database credentials are pulled from Secrets Manager at cold start and cached for the lifetime of the execution environment.

3) *Presigned URL Upload*: Every file upload uses the presigned-URL pattern. The desktop POSTs metadata to POST /vault/entries/{id}/upload, receives a presigned S3 PUT URL valid for five minutes, and uploads bytes directly to S3. This keeps API Gateway and Lambda entirely out of the data plane, bypassing the 6 MB Lambda payload limit.

E. Asynchronous Indexing Pipeline

When S3 emits an ObjectCreated event, the SQS IndexQueue receives it and a sqs-starter Lambda starts a Step Functions execution. Step Functions was chosen because each stage is independently retryable with its own backoff policy, the execution graph is visible in the AWS console, and per-stage timeouts protect the pipeline from hung stages [6].

Parsed text output and embedding vectors frequently exceed the Step Functions 256 KB payload limit. The pipeline uses S3 as a data bus: outputs exceeding 200 KB are written to s3://bucket/pipeline/{entry_id}/{stage}.json and only an S3 reference is passed in the payload. The Store Lambda deletes all intermediate pipeline objects after a successful write.

1) *Validate (Lambda, 256 MB)*: Checks MIME type against an allowlist of twenty-two types, validates size against the 50 MB ceiling, and verifies user quota. If the S3 object has been deleted between event emission and validation, the state machine routes to MarkDeleted rather than treating the run as a failure.

2) *Download and Parse (Container, 2 GB, 15 min)*: Downloads the file, computes a SHA-256 hash, and checks for prior indexed entries with the same hash. On a match, the state machine routes to CloneFromSource, copying prior chunks without re-parsing or re-embedding. Otherwise a kind-aware parser registry dispatches to docling, pytesseract OCR, pypdf, or code and tabular extractors as appropriate.

3) *Chunk (Lambda, 512 MB)*: Produces content chunks using cl100k_base tiktoken at 500 tokens with a 50-token overlap. Chunking strategy is kind-aware: paragraph-aligned for documents, logical-block for code, schema-narrative for tabular data.

4) *Embed (ECS Fargate, 8 GB, 2 vCPU)*: Runs BAAI/bge-m3 on ECS Fargate rather than Lambda because the model requires approximately 8 GB of RAM at inference time [7]. Step Functions uses the ecs:RunTask.sync integration to wait for task completion. The task runs in batches of 32, consults a Redis cache for previously-seen chunk hashes, and writes 384-dimensional vectors back to S3.

5) *Store (Lambda, 512 MB, VPC)*: Deletes prior chunks for the entry, batch-inserts new chunks with embeddings into vault.chunks via pgvector, updates index_status to indexed, and cleans up intermediate S3 pipeline files. Three terminal recovery states—MarkDeleted, MarkFailed, CloneFromSource—all route through this same Lambda with different action parameters.

The pipeline is idempotent on entry_id: the same upload retried produces the same chunks. It is also idempotent on file hash across entries: identical content is parsed and embedded once, then cloned for all subsequent entries referencing the same bytes.

F. Notification System

An EventBridge rule listens for Step Functions Execution Status Change events filtered to terminal states (SUCCEEDED, FAILED, TIMED_OUT, ABORTED). Matching events route to an SNS topic. The ws-notify Lambda looks up the user's active WebSocket connection ID from DynamoDB and pushes a structured progress message through the API Gateway Management API. Decoupling the notification path from the pipeline allows additional destinations to be added by subscribing to the SNS topic without modifying the state machine.

G. Security Architecture

1) *Network Isolation*: Every compute resource accessing the database lives in private subnets with no internet route. The VPC has two private subnets with no public subnets and no NAT gateway. The absence of internet egress is enforced at the network level.

2) *Encryption*: Two customer-managed KMS keys are maintained. The S3 CMK encrypts vault file objects; the RDS CMK encrypts database storage and associated Secrets Manager entries. Separate keys mean independent IAM grant control. Both keys have automatic annual rotation and provide a full audit trail through CloudTrail.

3) *IAM*: Each Lambda has its own minimal IAM execution role. The vault-crud role may read the database credentials secret, decrypt with the RDS CMK, and operate on vault-files bucket objects only. The ECS embed task uses split execution and task roles, keeping platform-level and application-level permissions separate.

4) *Secrets Management*: Three Secrets Manager entries cover database credentials, an embedding-provider API key, and a CloudFront key pair. Database credentials are auto-generated at stack creation and attached to the RDS instance via SecretTargetAttachment. Lambda functions retrieve credentials at cold-start and cache them for the lifetime of the execution environment.

H. Infrastructure as Code and Deployment

1) *TypeScript-Driven CloudFormation*: Every deployment is driven by a TypeScript script executed with tsx against the AWS SDK v3 CloudFormation client. The script reads per-environment configuration, resolves parameters from CloudFormation exports, packages Lambda code through esbuild, and submits create-or-update

requests. This preserves templates exactly as written in source control while moving orchestration logic into typed, testable code without the generated-template overhead of frameworks like CDK.

2) *Stack Dependency Graph*: shared-network and secrets carry no dependencies and deploy in either order. rds and storage depend on both. lambdas and api-gateway depend on the data layer. sqs-pipeline, pipeline-lambdas, step-functions-pipeline, and notifications-pipeline form a sequential dependency chain. cognito is independent. All outputs are exported under consistent names for Fn::ImportValue resolution.

3) *VPC-Isolated Schema Migrations*: Running npm run migrate triggers a one-shot Lambda pattern: the script creates a short-lived IAM role with VPC access, deploys the migration Lambda into private subnets, invokes it synchronously with the SQL payload, and deletes both the Lambda and role in a finally block regardless of outcome.

VI. RESULTS

The system was successfully deployed and exercised end-to-end in a development AWS environment. All twelve CloudFormation stacks were deployed in their correct dependency order, and the stack isolation was validated: redeploying a single stack after a configuration change did not require rebuilding dependent stacks.

Authentication, object storage, and the relational database layer are fully operational. Cognito User Pool authentication was exercised with Google federated login and GitHub OIDC, producing valid ID and access tokens consumed by all API Gateway routes. The presigned URL upload flow was tested end-to-end: a desktop client authenticated with a Cognito JWT, retrieved a five-minute presigned PUT URL, uploaded a file directly to S3 with the correct per-user prefix, and received confirmation without API Gateway or Lambda handling the file bytes.

The vault-crud Lambda handles all five CRUD operations plus presigned upload through a single dispatch handler. All six live AWS_PROXY routes in API Gateway return correct responses, and the eight MOCK routes return structured 501 NOT_IMPLEMENTED responses, enabling downstream consumers to develop against a stable contract before backing handlers are written.

The vault.entries, vault.chunks, and vault.chunks_fts tables were migrated to the development RDS instance using the one-shot Lambda migration pattern, and schema parity with the local desktop SQLite store was verified column by column. Both retrieval indexes were confirmed deployed: the IVFFlat cosine index on vault.chunks.embedding (lists=100) for dense approximate nearest-neighbor search, and the GIN tsvector index on vault.chunks_fts for sparse keyword search.

The indexing pipeline—SQS queue, sqs-starter Lambda, Step Functions state machine, all five pipeline stages, and the ECS Fargate embed task definition—is fully defined in CloudFormation and deployed. The EventBridge

rule, SNS topic, and ws-notify Lambda are live and ready to relay pipeline terminal state events through the WebSocket layer.

VI. FUTURE WORK

Several components are scoped and partially implemented, with full deployment targeted for the next development window.

The /vault/search endpoint is the most immediate deliverable. The hybrid retrieval indexes are in place; the remaining work is the search Lambda, which must execute the UNION across IVFFlat cosine ANN and tsvector GIN queries, normalize scores, and return broader chunk context to the caller.

WebSocket session management requires a DynamoDB table for connection ID mappings and a ws-notify Lambda wired to the notifications SNS topic. The WebSocket API Gateway resource and the DynamoDB table are scoped and templated; the remaining work is the connection lifecycle handlers.

CloudFront signed-URL delivery would replace the current direct S3 presigned URL pattern for downloads. RDS Proxy will be needed as vault-crud and the search Lambda scale under concurrent load, since the current pool-per-Lambda-container pattern will eventually reach the db.t3.micro connection ceiling. RDS Proxy slots in without application code changes.

Two extensions are already shaped in the repository but deliberately out of scope for the initial deployment. LightRAG-style graph retrieval [8] would use entity and entity_relationship tables reserved in the schema to support graph-traversal retrieval over named entities extracted at index time. Multilingual full-text search would replace the English-only tsvector tokenizer configuration once user data demonstrates the need.

Contributions

The system was developed collaboratively, with each team member responsible for a distinct architectural layer.

Krish established the cloud foundation, provisioning a Virtual Private Cloud (VPC) with private subnets, configuring an Amazon Cognito User Pool for authentication, and setting up API Gateway alongside the initial Lambda infrastructure.

Saluat designed and implemented the document indexing pipeline, encompassing input validation, document downloading and parsing, token-aware chunking using the cl100k tokenizer, and dense embedding generation via BAAI BGE-M3 deployed on Amazon ECS Fargate with Redis-backed caching. Processed embeddings were subsequently stored for downstream retrieval. AWS orchestration across pipeline stages was handled through SQS queues, Step Functions workflows, and Lambda/ECS task execution, with an EventBridge-triggered notification path routing alerts through SNS to connected WebSocket

clients.

Michael was responsible for the data persistence layer, including RDS provisioning, schema design, and table migrations. His contributions also included research into retrieval methodologies, establishing secure ECS-to-RDS connectivity, and implementing encryption at rest and in transit using AWS KMS and Secrets Manager.

Sachin led full-system integration with the client application, overseeing integration testing, production and development bug fixes, and ECS performance tuning. In his role as project lead, he coordinated team workflows and managed cross-cutting concerns including hybrid search across cloud and local environments, LLM-based retrieval with MCP integration, VPC connectivity issues, and merge conflict resolution.

REFERENCES

- [1] P. Lewis et al., "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks," Advances in Neural Information Processing Systems (NeurIPS), 2020.
- [2] Anysphere, Inc., "Cursor: The AI Code Editor," 2024. [Online]. Available: <https://cursor.sh>
- [3] Obsidian, "Obsidian: A second brain, for you, forever," 2024. [Online]. Available: <https://obsidian.md>
- [4] Continue, "Continue.dev: Open-source AI code assistant," 2024. [Online]. Available: <https://continue.dev>
- [5] S. Robertson and H. Zaragoza, "The Probabilistic Relevance Framework: BM25 and Beyond," Foundations and Trends in Information Retrieval, vol. 3, no. 4, pp. 333–389, 2009.
- [6] Amazon Web Services, "AWS Step Functions Developer Guide," AWS Documentation, 2024.
- [7] BAAI, "BGE M3-Embedding: Multi-Lingual, Multi-Functionality, Multi-Granularity Text Embeddings Through Self-Knowledge Distillation," arXiv:2402.03216, 2024.
- [8] Edge et al., "From Local to Global: A Graph RAG Approach to Query-Focused Summarization," arXiv:2404.16130, 2024.