Covasant

AI-Driven. Human-Inspired.

Semantic Documentation Search for Agentic Platforms

Executive Summary

AI coding agents, Claude Code, GitHub Copilot, Gemini Code Assist, now routinely navigate large monorepo codebases on behalf of developers. They read files, trace dependencies, and propose changes. But they share a common weakness: **documentation discovery.** When an agent needs to understand "how does tenant isolation work?" or "what is the authentication model?", it typically falls back to grep, file-tree browsing, or reading every file until it finds something relevant. This is slow, noisy, and semantically blind.

Text embeddings solve this by converting documentation into dense numerical vectors where **semantic similarity maps to geometric proximity**. An agent's natural-language question becomes a vector, and the most relevant documentation chunks are retrieved by cosine similarity, not keyword overlap. The result: agents find the right documentation in one query instead of dozens of file reads.

The challenge is cost. Dedicated vector databases (Pinecone, Weaviate, Qdrant) charge $70–400+/month in always-on infrastructure. Databricks Vector Search endpoints cost approximately $400/month with no scale-to-zero option. For documentation search, an episodic, low-throughput workload, this is economically disproportionate.

Covasant Auraa's approach eliminates this cost by using components already present in a Databricks workspace. Delta tables store embeddings alongside documentation chunks, no separate vector store. The SQL Warehouse computes cosine similarity on demand via higher-order array functions, scaling to zero when idle. The Foundation Model API generates embeddings pay-per-token, with no model hosting required.

The result is a fully functional semantic search system at **$2–10/month**, a 40–200x cost reduction compared to dedicated vector infrastructure, with zero idle costs and no external dependencies.

The Journey: From Grep to Semantic Search

Each stage solved a real problem and revealed the next bottleneck. The progression was not planned top-down, it emerged from watching agents struggle with real queries and asking "why did that fail?"

Stage 1, Grep

The starting point was the tool every developer already has: grep -r. Agents searching for documentation would shell out to grep with keywords extracted from the user's question, read the top-matching files, and synthesize an answer.

What worked: For exact-match queries, "find the principal repository", "where is the SQL warehouse config defined", grep is hard to beat. It searches everything (markdown and source code), returns precise line-level matches, and requires zero infrastructure.

What broke: Conceptual queries. An agent asking "how does tenant isolation work?" would grep for "tenant isolation" and get either zero results (the phrase doesn't appear verbatim) or a flood of tangentially related files. Grep returns mentions, not answers. Agents compensated by issuing multiple grep queries with different keywords, reading dozens of files, and consuming enormous context windows, often 40,000–50,000 tokens per lookup.

Grep's failure on conceptual queries pointed to a deeper problem: the tool has no understanding of what documents are about. It can find strings, but it cannot reason about topics. The next stage attempted to add that reasoning layer by hand.

Stage 2, Directory-Level Indexes

The insight was that documentation has structure that grep ignores. Each directory in the monorepo represents a logical grouping, authentication docs, architecture decisions, deployment guides. If we could describe each directory's purpose and contents, agents could navigate the hierarchy instead of searching blindly.

We introduced structured index files at the directory level. Each file contained LLM-generated summaries describing the directory's contents, manually curated keywords and topics augmented by LLM tagging, and explicit entry points, pointers to the most important files in each directory. Agents could read a directory's index, decide whether it was relevant, and follow entry points to the actual documents. A two-hop process: read the index, then read the document.

What worked: For navigational queries ("where is the auth documentation?"), this was dramatically better than grep. The LLM-generated summaries captured document purpose, not just content. An agent could match "tenant isolation" to a directory summary that said "design documents for Unity Catalog-based multi-tenancy" even though the phrase "tenant isolation" never appeared.

What broke: Scale and maintenance. With dozens of directories, each needing a curated index, the system required ongoing human attention. More fundamentally, agents had to read the full index file (often 25,000+ tokens for deeply nested directories) to find relevant entries, trading grep's token cost for a different kind of token cost.

The directory-level indexes proved that pre-computed reasoning about documents was the right idea, but the flat structure meant agents had no way to narrow the search without reading every index. The next step was to add a layer above the directory indexes, a table of contents for the table of contents.

Stage 3, Top-Level Manifest

The next iteration added a root-level manifest that referenced all directory-level indexes. Agents could start at the manifest, scan high-level descriptions to identify promising directories, then drill into specific index files.

This created a three-level navigation hierarchy: manifest → directory index → document. Machine-readable hints in each manifest entry provided entry points, related documents, and recommended reading order that agents could follow without parsing free text.

What worked: The structured index approach achieved the highest precision in our benchmark (P@5=0.200, MRR=0.261). Pre-computed LLM reasoning at build time, summarizing documents, grouping them into clusters, annotating entry points, consistently outperformed real-time similarity matching. The reasoning was done once and reused across every agent query.

What broke: Two things. First, the token cost was still high (~39,000 tokens per query) because agents had to read through structured metadata files to navigate. Second, the system was fundamentally a curated knowledge base, someone had to write the summaries, choose the keywords, and maintain the hierarchy. It didn't scale to a fast-moving codebase with hundreds of documents changing weekly.

The manifest experiment revealed a fundamental limitation of navigation-based approaches: no matter how well you organize the metadata, agents still have to read it, and reading costs tokens. What if we could skip the navigation entirely and let agents ask a question and get answers directly?

Stage 4, Semantic Search with Flat Content Chunking

The semantic search approach inverted the paradigm. Instead of building a navigation structure that agents traverse, we built an index that agents query. The pipeline:

1. Chunks every markdown file by heading hierarchy
2. Embeds each chunk via Databricks Foundation Model API
3. Stores chunks + embeddings in a Delta table
4. Agents query via SQL Warehouse cosine similarity, a single API call

This eliminated the navigation overhead entirely. An agent asking "how does tenant isolation work?" gets back the 5 most semantically similar chunks in ~1,100 tokens, with no intermediate reads.

What worked: Token efficiency was transformative, 42x fewer tokens than grep or the structured index. The system required zero curation. New documents were automatically indexed on the next pipeline run. And it worked for vocabulary-gap queries: "permission resolution" matched documents about "access policy evaluation" because the embeddings captured semantic proximity.

What broke: Precision was lower than the structured index (P@5=0.117 vs 0.200). Content chunks capture what paragraphs say, but agents often need to know what documents are about. A 200-word chunk from the middle of a design document doesn't convey that this is the authoritative reference for tenant isolation, it just contains a paragraph that mentions tenant isolation alongside other topics. The structured index's document-level summaries provided exactly this "aboutness" signal.

The precision gap between semantic search and the structured index pointed back to the same insight from Stage 2: LLM-generated summaries capture document purpose in a way that raw content does not. The question became: could we bring the summarization approach into the embedding pipeline itself?

Stage 5, Hierarchical Chunking with LLM Summaries

The final stage brought the structured index insight, pre-computed LLM reasoning about document purpose, into the embedding pipeline. Alongside content chunks, the pipeline now generates **document summaries,** where Llama 3.3 70B distills each document into a 2–3 sentence description of what it covers and why an agent would need it. It also generates **section summaries,** each major H2 section gets its own summary for finer-grained matching.

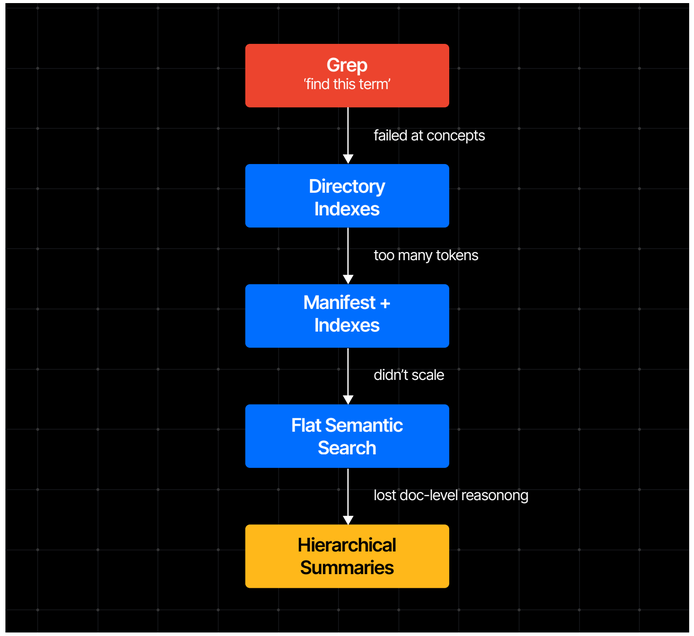
These summary chunks are embedded alongside content chunks and participate in the same cosine similarity search. An agent's query now matches against both "what the paragraph says" (content chunks) and "what the document is about" (summary chunks).

Results: P@5 improved from 0.117 to 0.127 (+8.5%), with the largest gains on medium-difficulty conceptual queries (+18%). The improvement was modest because summaries help a specific query class, conceptual "what is X?" questions, while easy queries (keyword lookups) and hard queries (cross-cutting architecture questions) need different interventions.

With five stages behind us, each solving one problem and exposing the next, a pattern emerges about what works, what doesn't, and why.

What We Learned

Each stage was the right solution for the problem we understood at the time. The progression reveals a recurring pattern:



The fundamental tension is between **pre-computed reasoning** (expensive to build, cheap to query) and **real-time similarity** (cheap to build, limited in what it can capture). The best system combines both: build-time LLM reasoning (summaries) embedded into a query-time similarity index.

The Problem: Agent Documentation Discovery at Scale

The journey traced above tells us what happened. The following sections explain why, starting with the nature and scale of the problem that motivated each stage.

The Documentation Discovery Gap

Modern data platforms are complex. Covasant Auraa's monorepo contains **600+ markdown documents** spread across **13 component directories,** covering architecture decisions, authentication patterns, deployment guides, API references, and troubleshooting procedures. Each component maintains its own documentation subtree. AIGENE handles identity and governance, principals, tenant role bindings, access policies. Foundra manages platform initialization, tenant onboarding, schema provisioning, operations logging. AIREKA powers documentation search and vector indexing. DataDuct orchestrates source-to-lakehouse data pipelines. AiCura monitors data quality, drift detection, and contract enforcement. Nexara manages the agent registry, tool definitions and orchestration workflows. Infraia executes Unity Catalog grants, storage configuration, and infrastructure provisioning.

Seven components. Seven documentation subtrees. Seven different vocabularies for overlapping concepts.

When a developer asks an AI coding agent to "add a new tenant onboarding step," the agent must first understand the existing onboarding flow. That information is scattered: platform initialization documentation describes provisioning procedures, identity and governance docs cover principal management, architecture docs explain the project organization strategy, infrastructure docs detail Unity Catalog provisioning, and authentication docs define the tenant access design.

No single file contains the complete answer. The agent must discover, read, and synthesize information across all of these locations. The question is: **how does it find the right files?**

Why Keyword Search Falls Short

The conventional approach, grep or ripgrep, searches for literal string matches. This fails in several important ways:

The most fundamental failure is **vocabulary mismatch.** A developer asks "how do I set up permissions?" but the documentation uses "RBAC binding configuration" and "Unity Catalog grants." Keyword search misses the connection entirely.

Then there is context loss. grep returns individual lines. An agent searching for "authentication" gets 200+ line-level matches with no sense of which sections are most relevant or how they relate. All matches are treated equally, a passing mention of "authentication" in a troubleshooting guide is returned alongside the definitive authentication design document.

Keyword search is also blind across component boundaries. An agent working in one component directory typically searches only that subtree, missing critical documentation in sibling directories that implements the other half of the same workflow. And the scaling costs are punishing: with 600+ documents, an exhaustive keyword search requires the agent to read dozens of files, consuming context window tokens and increasing response latency. In practice, agents give up after 5–10 file reads and work with incomplete information.

The Cost Barrier to Semantic Search

Embedding-based search solves all of these problems, but introduces a cost problem of its own. The standard architecture requires an embedding model to convert text into vectors, a vector database to store and query them efficiently, and always-on infrastructure to serve low-latency queries.

For high-throughput production workloads, e-commerce search, recommendation engines, RAG chatbots, this infrastructure pays for itself. But documentation search is fundamentally different. Usage is episodic: agents query documentation when they encounter unfamiliar territory, perhaps 10–50 times per day during active development, and zero times on quiet days. Throughput is low, a single query returns 5–10 results, with no need for millisecond latency. And the corpus is stable, changing weekly at most, with no real-time indexing requirement.

Paying $400/month for an always-on vector search endpoint to handle 50 queries per day is like renting a warehouse to store a bookshelf. The capacity exists for a reason, it is simply the wrong reason for this workload.

Embeddings: Turning Documentation into Searchable Vectors

The three-part problem, documentation scale, keyword blindness, and the cost barrier to always-on vector infrastructure, points to a specific kind of solution: one that captures semantic meaning without requiring dedicated search infrastructure. That solution starts with text embeddings.

What Are Text Embeddings?

A text embedding is a function that maps a passage of text to a fixed-length vector of floating-point numbers. Passages with similar meaning produce vectors that are close together in the embedding space; passages with different meaning produce vectors that are far apart.

For example, given a 1024-dimensional embedding model:

- "How do I configure tenant permissions?" → [0.032, -0.118, 0.247, ...]
- "RBAC binding setup for new tenants" → [0.029, -0.121, 0.251, ...] (very close)
- "Installing Python dependencies with pip" → [-0.445, 0.082, -0.031, ...] (distant)

Cosine similarity measures the angle between two vectors, producing a score between -1 and 1. Values close to 1 indicate high semantic similarity. In practice, documentation search results with scores above 0.7 are typically relevant; scores above 0.85 are highly relevant.

Why Embeddings Work for Documentation

Embeddings are particularly effective for documentation search because they perform **semantic bridging,** the model learns that "authentication," "identity management," "RBAC," and "access control" are related concepts, even when the exact words differ. An agent asking about "permissions" will find documents about "grants" and "role bindings."

Rather than embedding entire documents (which dilute meaning), we split documentation by **heading hierarchy,** H1, H2, H3. Each chunk represents a coherent section, a concept, a procedure, or a decision, and carries its own embedding. Search results point to the specific section, not just the file.

Each chunk also inherits metadata from YAML frontmatter: the component it belongs to, the domain, its status, and keywords. Agents can filter results by component or domain before ranking by similarity. And unlike keyword search, embedding-based search returns results **ordered by relevance score.** The agent reads the top 5 results and gets the best available answer, no manual triage required.

The Embedding Model Choice

With the conceptual case for embeddings established, the practical question becomes: which model, hosted where, at what cost?

Covasant Auraa uses **databricks-gte-large-en,** a General Text Embedding model hosted via the Databricks Foundation Model API (FMAPI):

Model	Dimensions	Token Window	Dimensions	Dimensions
databricks-gte-large-en	1024	8,192 tokens	Databricks FMAPI	Pay-per-token
databricks-bge-large-en	1024	8,192 tokens	Databricks FMAPI	Pay-per-token
all-MiniLM-L6-v2	384	512 tokens	Self-hosted (PyTorch)	Free (compute only)
OpenAI text-embedding-3-small	1536	8,191 tokens	OpenAI API	$0.02/1M tokens

GTE-large-en was selected for three reasons:

1. **No model hosting.** FMAPI provides embeddings as a serverless API call. There is no GPU instance to provision, no model to download, no container to manage.
2. **Sufficient capacity.** The 8,192-token window accommodates even the longest documentation sections without truncation. The 1024-dimensional output provides strong discriminative power.
3. **Native integration.** FMAPI authenticates via the same Databricks credentials (host + token) used by every other Covasant Auraa component. No additional API keys or accounts.

The cost of embedding the entire Covasant Auraa documentation corpus (~15,000 chunks, ~2–3 million tokens) is approximately **$0.03–0.05 per full index build.**

Architecture: Delta-Native Semantic Search

Embeddings and a suitable model provide the raw ingredients. The question is how to compose them into a system that is both functional and economical, without introducing the always-on infrastructure costs that make traditional vector search prohibitive for this workload.

Design Principles

The system is built on four principles.

Zero idle cost. No component runs continuously. The SQL Warehouse suspends when idle; the FMAPI charges nothing when unused; the Delta table is passive storage. Monthly cost at rest: $0.

Databricks-native. Every component, storage, compute, embedding, authentication, is provided by the Databricks workspace. No external services, no API keys, no network egress.

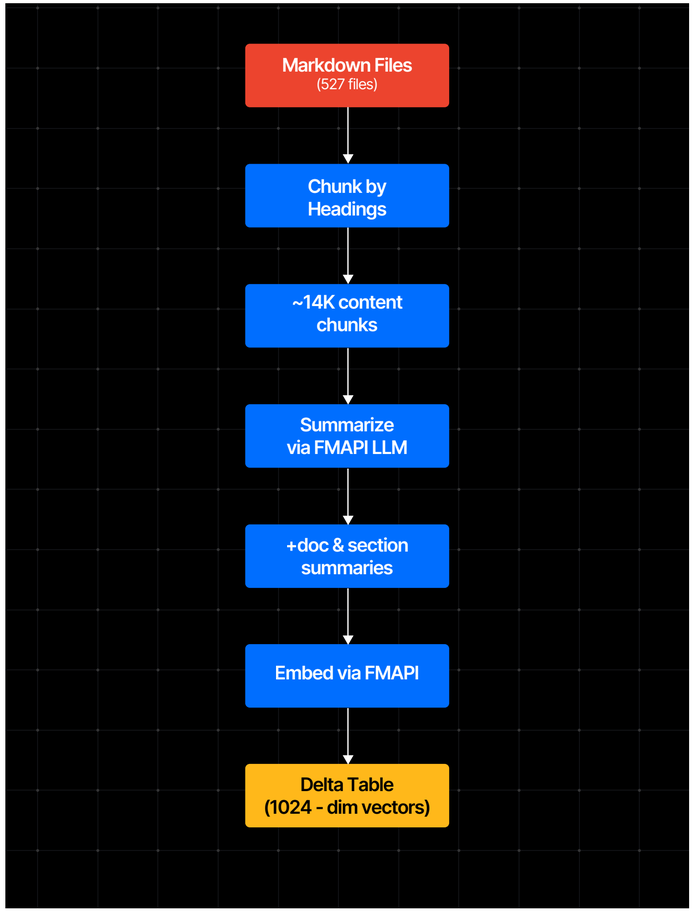
Agent-first. The primary consumers are AI coding agents, not humans. The system exposes a CLI and Python API designed for programmatic invocation and JSON output, not a web interface.

Pluggable. The vector search abstraction supports multiple backends. Organizations with existing Pinecone or Databricks Vector Search infrastructure can swap providers without changing application code.

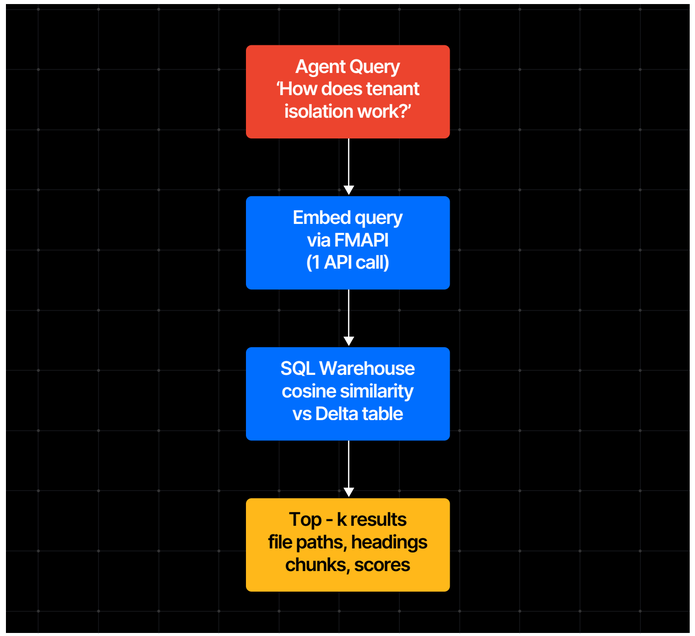
System Architecture

The system has two operational modes: **indexing** (periodic batch) and **searching** (on-demand per query).

Indexing Pipeline (weekly Databricks job)



Search Flow (on-demand, per agent query)



The Delta table serves **dual purpose:** it is both the persistent store of record (audit, lineage, change detection) and the search index. There is no separate vector database. At query time, the SQL Warehouse computes cosine similarity using **higher-order array functions** (zip_with, aggregate, transform) directly against the ARRAY<FLOAT> embedding column, no dedicated vector search function is required.

Component Roles

Component	Role	Databricks Service	Cost Model
Document chunks + embeddings	Storage and search index	Delta table (Unity Catalog)	Included with workspace
Embedding generation	Convert text to 1024-dim vectors	Foundation Model API	Pay-per-token (~$0.01/1M)
Similarity search	Cosine similarity via higher-order SQL	Serverless SQL Warehouse	Pay-per-query (scales to zero)
Pipeline orchestration	Weekly index refresh	Databricks Job (python_wheel_task)	Serverless compute
Agent search CLI	doc-search "query"	Local Python (databricks-sdk)	Free (SDK only)

Implementation: The Four-Stage Pipeline

The indexing pipeline runs as a Databricks job task, scheduled weekly, and executes four stages sequentially. The entire pipeline completes in 5–10 minutes at a cost of less than $0.50 per run.

Stage 1, Document Discovery

The pipeline walks component documentation directories plus the root docs/ folder, collecting all markdown files. In Covasant Auraa's monorepo, this includes 13

```
docs/                # Platform-level architecture and design documentation
<identity>/docs/     # Identity model, principals, tenant role bindings
<ingestion>/docs/    # Data ingestion engine design and configuration
<quality>/docs/      # Data quality rules, drift detection, contracts
<governance>/docs/   # Governance policies and access control
<search>/docs/       # Search, explainability, and AI-assisted discovery
<platform>/docs/     # Platform initialization, tenant onboarding
<agents>/docs/       # Agent registry, tool definitions, orchestration
... (13 directories total)
```

Directories matching archive/, reference_projects/, __pycache__/, .venv/, and node_modules/ are excluded. The current corpus contains **620 markdown files.**

Stage 2, Heading-Based Chunking

Each markdown file is split into **content chunks** based on its heading hierarchy. A chunk corresponds to a section: everything under a ## or ### heading, including its body text, code blocks, and sub-content.

The chunking process:

1. **Extract YAML frontmatter** (title, component, domain, status, keywords, audience)
2. **Parse heading hierarchy** (H1 through H6)
3. **Build section breadcrumbs,** e.g., "Architecture / Tenant Isolation / Unity Catalog Grants"
4. **Split at heading boundaries,** each section becomes one chunk
5. **Subdivide oversized sections,** chunks exceeding 4,000 characters are split at paragraph boundaries

Each chunk carries rich context. Identity fields include a chunk ID (MD5 hash), document ID, file path, and content hash. Hierarchy fields capture the document title, section breadcrumb, heading text and level, and the chunk's position index. The chunk text itself is stored alongside a code-block flag, and metadata from YAML frontmatter, component, domain, status, keywords, audience, travels with every chunk. Timestamps record when the document was last updated and when it was indexed. Finally, a chunk type field distinguishes content chunks (the default) from doc_summary and section_summary chunks (see §6.3).

The 527 indexed documents yield approximately **13,858 content chunks,** an average of 26 chunks per document.

Limitations of Flat Content Chunking

Heading-based chunking produces semantically coherent fragments, but it introduces two retrieval problems that became apparent during benchmarking (see §10):

Chunk dilution. A 20-section architecture document produces ~20 chunks. When an agent queries "how does tenant isolation work?", only 1–2 of those chunks may score highly for similarity. The document's signal is diluted across its many chunks. A shorter, less comprehensive document with a single high-scoring chunk can outrank the definitive reference.

Missed holistic matches. Some queries ask about what a document covers, not what a specific paragraph says. "What is the complete authentication flow?" requires understanding a comprehensive auth index document as a whole, but no individual content chunk captures the full scope. The query matches weakly against many chunks rather than strongly against one.

These are not hypothetical problems. In benchmark testing (§10), flat content chunking achieved P@5=0.138 across 15 evaluation queries. The structured index approach, which operates at the document level via LLM-generated semantic summaries, achieved P@5=0.200 on the same queries. The difference is attributable to pre-computed document-level reasoning: it knows what each document is about, not just what text it contains.

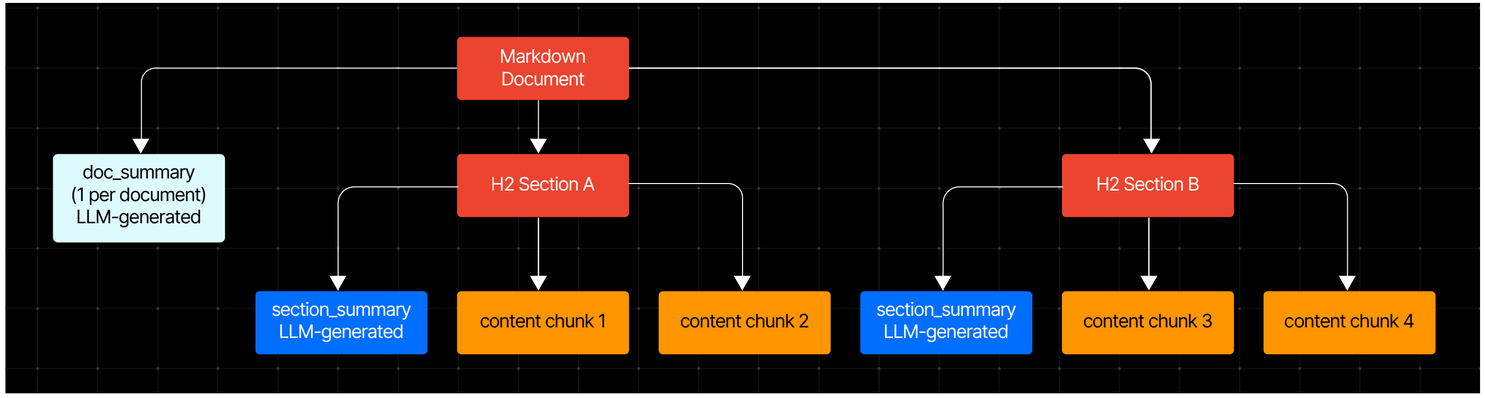
This insight motivates the hierarchical chunking strategy described in §6.3.

Stage 2b, Hierarchical Summary Generation

To address chunk dilution and missed holistic matches, the pipeline generates **summary chunks** at two levels of the document hierarchy. These are embedded alongside content chunks and participate in the same cosine similarity search. This stage is conceptually an extension of Stage 2, it operates on the same documents but produces a different kind of output: LLM-distilled descriptions that capture document and section purpose rather than raw content.

The Three Chunk Types

Chunk Type	Scope	Generated By	Purpose
content	Section (H2/H3)	Heading-based split (§6.2)	Detailed retrieval, specific concepts, procedures, code
doc_summary	Entire document	LLM summarization (FMAPI)	Holistic matching, "what is this document about?"
section_summary	H2 section	LLM summarization (FMAPI)	Section-level matching, bridges document and paragraph granularity



Document Summary Chunks

For each document with 2 or more content chunks, the pipeline generates a single doc_summary chunk. The process:

1. **Build extractive input:** Document title + heading outline + opening paragraph. This structural sketch is similar to what structured index systems distill at build time.
2. **LLM summarization:** The extractive input is sent to a Databricks FMAPI chat model with the instruction: "Summarize this technical document in 2-3 sentences. Describe what it covers, which system component it relates to, and what questions a developer would consult it to answer."
3. **Fallback:** If the LLM call fails, the extractive input is used directly as the summary text.

The resulting summary is phrased to match the kind of natural-language queries agents produce. For example, a document about tenant data isolation would produce a summary like: "This document describes how the platform implements tenant data isolation using Unity Catalog. It covers catalog-per-tenant strategy, schema-level permissions, and managed storage locations. It is the primary reference for understanding how data is separated between tenants."

This summary chunk will score highly against the query "how does tenant isolation work?", higher than any individual content chunk from the same document, because it captures the document's entire scope in language that aligns with the query's intent.

Section Summary Chunks

For each H2 section within a document that contains 2 or more content chunks, the pipeline generates a section_summary chunk:

1. **Build section input:** Concatenate the first 500 characters of each content chunk in the section (up to 5 chunks).
2. **LLM summarization:** Sent to FMAPI with: "Summarize this documentation section in 1-2 sentences. Focus on the key concepts, decisions, or instructions it contains."
3. **Fallback:** First sentence of each chunk, concatenated as bullet points.

Section summaries bridge the gap between document-level (too broad for specific questions) and paragraph-level (too narrow for section-scoped questions).

Pre-Computed Semantic Distillation

This hierarchical approach is inspired by the structured index system's success in benchmark testing. LLM-generated metadata, document summaries, topic groupings, and entry points, represents a form of **pre-computed semantic distillation:** the LLM reasons about document purpose and relationships at build time, compressing its understanding into structured metadata.

Hierarchical chunking brings this same strategy into the embedding space. Instead of relying on keyword matching against pre-computed summaries, the summaries are embedded and retrieved via cosine similarity, inheriting semantic search's ability to match conceptual queries that keyword matching would miss.

The cost is modest: for a corpus of 527 documents, approximately 527 document summaries and an estimated 500–1,000 section summaries are generated. At ~500 input tokens and ~200 output tokens per LLM call, the total is approximately 700K–1M tokens, roughly **$0.50–1.00** per full index build using FMAPI pricing.

Additional Chunking Recommendations

Several further improvements to chunking quality are under consideration. The current splitter may separate a code block from its explanatory text if they fall across a heading boundary; a code-aware splitter would keep code blocks with their immediately preceding description. Similarly, markdown tables should be treated as atomic units and kept with their nearest heading rather than split across chunk boundaries.

Adding 1–2 sentences of overlap between adjacent chunks, a sliding window approach, would prevent information loss at chunk boundaries, at the cost of ~10% more chunks. The highest-impact improvement, however, is **source code indexing:** extending the pipeline to chunk .py files by class and function boundaries, with docstrings as natural chunk text. This would close the coverage gap that prevents semantic search from finding implementation-level answers (see §10.3).

Stage 3, Embedding Generation

Each chunk's text is sent to the Databricks Foundation Model API in batches of 32:

```
response = workspace_client.serving_endpoints.query(
    name="databricks-gte-large-en",
    input=batch_of_texts,  # Up to 32 texts per call
)
embedding = response.data[i].embedding  # 1024-dim float array
```

The model processes approximately 2–3 million tokens across all 15,000 chunks. At FMAPI's pay-per-token pricing, this costs **less than $0.05** per full index build. Progress is logged every batch.

Stage 4, Delta Table Storage

Chunks and their embeddings are written to a single Delta table in Unity Catalog:

```
<catalog>.<schema>.doc_search_index
```

The table schema includes all chunk metadata plus the embedding column (ARRAY<FLOAT>, 1024 elements). The write uses **overwrite mode,** each pipeline run produces a complete, fresh index. This is idempotent: running the pipeline twice produces identical results.

The Delta table is the search index. There is no ETL step to load embeddings into a separate vector store. When an agent queries, the SQL Warehouse reads directly from this table.

Agent Search
How AI Assistants Query the Index

The indexing pipeline runs silently in the background. From an agent's perspective, none of that machinery is visible, the documentation search system is a single command, or a single API call.

The Search Flow

When an AI coding agent needs documentation, it executes a single command:

```
doc-search "how does tenant isolation work?"
```

Behind the scenes, three operations execute:

Embed the query. The query text is sent to FMAPI (databricks-gte-large-en), which returns a 1024-dimensional vector. This single API call takes approximately 100–200ms.

Search the Delta table. The SQL Warehouse computes cosine similarity using higher-order array functions:

```
SELECT * EXCEPT(embedding),
    aggregate(
        zip_with(embedding, query_vec, (a, b) -> CAST(a AS
DOUBLE) * CAST(b AS DOUBLE)),
        0.0D, (acc, x) -> acc + x
    ) / (
        sqrt(aggregate(transform(embedding, x -> CAST(x AS
DOUBLE) * x), 0.0D, (acc, x) -> acc + x)) *
        sqrt(aggregate(transform(query_vec, x -> CAST(x AS
DOUBLE) * x), 0.0D, (acc, x) -> acc + x))
    ) AS score
FROM <catalog>.<schema>.doc_search_index
WHERE embedding IS NOT NULL
ORDER BY score DESC
LIMIT 5
```

This computes dot(a,b) / (norm(a) * norm(b)), the standard cosine similarity formula, using Databricks SQL's built-in higher-order functions (zip_with, aggregate, transform). Note that cosine_similarity() is a Spark SQL function not available on SQL Warehouse, so the manual computation is necessary. The EXCEPT(embedding) clause excludes the large embedding array from results, returning only the metadata and text. The SQL Warehouse resumes from suspension in 5–10 seconds if idle; subsequent queries execute in 1–3 seconds.

Return ranked results. The agent receives results ordered by cosine similarity score:

```
Search: "how does tenant isolation work?"
Results: 5
  1. [0.892] docs/architecture/tenant_isolation.md
     Heading: Unity Catalog Grants
     Component: platform
  2. [0.847] identity/docs/identity_model.md
     Heading: Tenant Role Bindings
     Component: identity
  3. [0.831]
docs/architecture/ADR-001-configurable-project-organization.md
     Heading: Strategy 2, Catalog Per Project
     Component: architecture
```

The agent now knows exactly which files to read and which sections to focus on.

CLI and Python API

Command-line interface (for agent tool invocation):

```
# Basic search
doc-search "authentication setup"
# Filter by component
doc-search "data quality rules" --component quality --top-k 3
# Machine-readable output
doc-search "connector configuration" --json
```

Python API (for programmatic integration):

```
from your_search_package import search
results = search(
    "how does tenant isolation work?",
    top_k=5,
    component="platform",
)
for r in results:
    print(f"{r['score']:.3f}  {r['file_path']} , {r['heading']}")
```

Both interfaces return identical data: cosine similarity score, file path, heading text, section breadcrumb, document title, component, domain, and a 500-character text snippet.

Filtering and Ranking

Filter	Example	Effect
--component	identity	Only chunks from identity/governance docs
--domain	architecture	Only architecture-domain docs
Status	active	Exclude archived/draft content
Namespace	(tenant ID)	Multi-tenant isolation

Filters are applied as SQL WHERE clauses before the cosine similarity computation, reducing the scan set and improving query performance.

Cost Analysis

The economic argument for this architecture rests on a single observation: **documentation search is an episodic workload.** Infrastructure that charges for idle time is structurally mismatched to this usage pattern.

Cost Comparison

Cost Component	Delta + SQL Warehouse	Databricks Vector Search	Pinecone Serverless	ChromaDB Cloud
Index build (weekly)	~$0.50/run	~$0.50/run	~$0.50/run + egress	~$0.50/run + egress
Monthly indexing	~$2	~$2	~$2	~$2
Search (per query)	~$0.001	Included in endpoint	~$0.0001	Variable
Idle cost	$0	~$400/month	$0 (starter)	$0 (free tier)
Monthly minimum	~$2	~$402	~$2 (starter)	~$2 (free tier)
External dependencies	None	None	API key, network egress	API key, network egress
Data residency	In workspace	In workspace	External cloud	External cloud
Auth integration	Databricks native	Databricks native	Separate credentials	Separate credentials

Annual Projection

For a team with moderate agent usage (50 queries/day average, 5 days/week):

Solution	Annual Cost	Notes
Delta + SQL Warehouse	**$24–120**	Indexing + query compute
Databricks Vector Search	**$4,800+**	Always-on endpoint
Pinecone (Starter)	**$24–60**	Comparable, but external
Pinecone (Standard)	**$840–1,800**	Higher tiers for production

The Delta + SQL Warehouse approach achieves cost parity with the cheapest external options while keeping all data and compute within the Databricks workspace, no vendor dependencies, no API key management, no data egress.

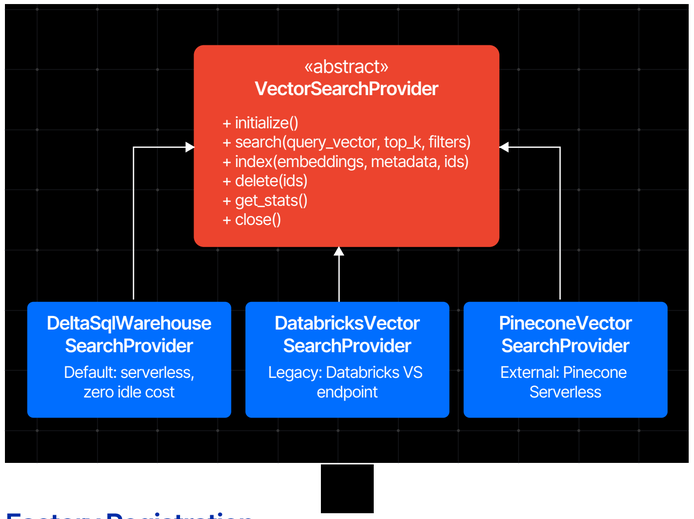
Abstraction Layer Supporting Multiple Backends

The cost analysis makes a clear case for the Delta + SQL Warehouse approach, but not every organization shares the same infrastructure constraints. While the Delta implementation is the default, the system is designed to be **provider-agnostic.** The vector search abstraction layer defines a set of abstract base classes that any backend can implement, allowing the application layer to remain unchanged as backend requirements evolve.

The Provider Pattern

```
class VectorSearchProvider(ABC):
    """Abstract base class for vector search backends."""
    def initialize(self) → None: ...
    def search(self, query_vector, top_k, filters) → List[SearchResult]: ...
    def index(self, embeddings, metadata, ids) → Dict: ...
    def delete(self, ids) → Dict: ...
    def get_stats(self) → Dict: ...
    def close(self) → None: ...
```

Three implementations are provided:



Factory Registration

Providers are registered via a factory pattern with lazy loading:

```
from vector_search import VectorSearchFactory
config = VectorSearchConfig(
    provider_type="delta_warehouse",  # or "databricks", "pinecone"
    embedding_model="databricks-gte-large-en",
    index_name="doc_search_index",
    dimension=1024,
    provider_options={
        "source_table": "<catalog>.<schema>.doc_search_index",
    },
)
provider = VectorSearchFactory.create(config)
```

Switching backends requires changing a single configuration value. Application code, the pipeline, the search CLI, and the discovery service, does not change. This allows organizations to start with the zero-cost Delta approach and migrate to a dedicated vector database if query volume or latency requirements increase.

Empirical Evaluation: Three Discovery Approaches

Architecture and cost models are necessary but not sufficient, the system must actually find the right documents. We conducted a controlled benchmark comparing three approaches across 15 evaluation queries with human-curated ground truth.

Methodology

Corpus. The Covasant Auraa monorepo contains 937 markdown files across 13 component directories. The semantic search index contains 13,858 content chunks from 527 documents (the remaining ~410 files fall outside the workspace bundle sync path). The structured index MANIFEST catalogs 370 documents across 56 section indexes.

Query set. 15 queries spanning three difficulty tiers, with human-curated ground truth (gold and silver files). The five **Easy** queries test literal symbol lookup, specific class names, configuration variables, file paths, and function entry points. The five **Medium** queries are conceptual: tenant isolation, engine abstraction, permission resolution, embedding model choice, deployment workflow. These require understanding what a document describes, not just what text it contains. The five **Hard** queries are cross-component: tenant onboarding end-to-end, project organization strategies, adding a REST endpoint, authentication token flow, ingestion-to-transformation data flow. These require synthesizing information across multiple files and components.

Approaches compared

Approach	Mechanism	File types	Cost model
Grep	Python-native keyword search across .md and .py files, ranked by keyword hit count	Markdown + source code	Listing output + full content of top-5 files
Structured index	MANIFEST keyword match → follow entry points → read matched docs	Markdown only	MANIFEST (100KB) + matched index files + entry point content
Grep	Python-native keyword search across .md and .py files, ranked by keyword hit count	Markdown + source code	Listing output + full content of top-5 files
Semantic search	CLI → FMAPI embedding → SQL Warehouse cosine similarity	Markdown only (Delta index)	JSON output only (~1K tokens per query)

Metrics: Precision@5 (fraction of top-5 results in gold+silver set), MRR (reciprocal rank of first gold file), token consumption (chars / 4), wall-clock time.

Results, Baseline (Flat Content Chunking)

These results represent the baseline system with heading-based content chunking only (no hierarchical summaries). The full corpus contains 20,576 content chunks from 937 markdown files.

Overall Performance

Metric	Grep	Structured Index	Semantic Search
Avg Tokens	46,442	39,594	1,058
Avg Time (sec)	1.35	0.04	4.50
Avg Precision@5	0.107	0.200	0.117
Avg MRR	0.172	0.261	0.067

Performance by Difficulty Tier

Tier	Grep P@5	Structured Index P@5	Semantic P@5	Winner
Easy	0.280	39,594	0.133	Grep
Medium	0.000	0.04	0.167	Semantic
Hard	0.040	0.200	0.050	Structured Index

Per-Query Detail

Query	Tier	Grep P@5	Structured Index P@5	Semantic P@5
Q01: Find principal repository implementation	easy	0.40	0.00	0.00
Q02: Find SQL warehouse config variable	easy	0.40	0.00	0.00
Q03: Find tenant path resolver	easy	0.40	1.00	0.67
Q04: MANAGED LOCATION usage	easy	0.00	0.00	0.00
Q05: Find doc search pipeline entry point	easy	0.20	0.00	0.00
Q06: Tenant isolation in Unity Catalog	medium	0.00	0.40	0.00
Q07: Engine abstraction layer	medium	0.00	0.00	0.25
Q08: Permission resolution for tenant user	medium	0.00	0.00	0.33
Q09: Embedding model for doc search	medium	0.00	0.00	0.25
Q10: Deploy wheel to Databricks Apps	medium	0.00	0.40	0.00
Q11: Tenant onboarding end-to-end	hard	0.00	0.00	0.00
Q12: Three project organization strategies	hard	0.20	0.20	0.25
Q13: Add REST API endpoint to unified service	hard	0.00	0.40	0.00
Q14: Authentication token to tenant access flow	hard	0.00	0.40	0.00
Q15: Data flow ingestion to transformation	hard	0.00	0.20	0.00

Analysis

Why the Structured Index Outperformed Semantic Search

The structured index approach achieved the highest overall precision despite sharing a fundamental constraint with semantic search: both can only discover markdown documentation, not source code.

The explanation lies in how each approach reasons about documents:

Pre-computed semantic distillation vs. real-time similarity. The structured index metadata is LLM-generated, semantic summaries, document descriptions, and cluster groupings are produced by agents at index-build time. This is a form of pre-computed semantic distillation: the LLM reasons about "what is this document about?" once, and compresses the result into structured metadata that agents navigate via keyword matching.

Semantic search defers all reasoning to query time. Raw document chunks are embedded, and similarity is computed against the query embedding at search time. The embeddings capture surface-level semantic proximity, "tenant isolation" matches chunks that mention tenant isolation, but not the holistic reasoning about which document is the authoritative reference for a given topic.

Document-level vs. chunk-level retrieval. Each structured index entry represents a whole document with a title, description, and tags that capture what the document is for. Semantic search operates on chunks, fragments of documents that capture what a paragraph says. For navigational queries ("where is the auth documentation?"), document-level matching outperforms paragraph-level matching.

Corpus Coverage Asymmetry

A critical benchmark design artifact: **grep searches both .md and .py files**, while the structured index and semantic search can only discover markdown. Of the 15 queries, 7 have .py source code files in their gold answers. Neither approach can find these files.

This explains grep's Easy-tier lead (P@5=0.280): it finds source code files like repository implementations and pipeline entry points that the documentation-only approaches structurally cannot match.

Where Semantic Search Wins

Despite lower overall precision, semantic search demonstrated clear advantages on conceptual queries:

Query	Semantic P@5	Structured Index P@5	Why
Q03: Find tenant path resolver	0.67	1.00	Found design doc + ADR via embedding similarity
Q07: Engine abstraction layer	0.25	0.00	Embeddings captured "abstraction layer" → engine design doc
Q08: Permission resolution	0.33	0.00	Found identity/auth docs via semantic proximity
Q09: Embedding model for doc search	0.25	0.00	Found configuration + whitepaper docs
Q12: Project organization strategies	0.25	0.20	Found ADR via semantic matching

The pattern: semantic search excels at **Medium-tier queries** (P@5=0.167) where the structured index metadata lacks relevant keywords. Embeddings bridge vocabulary gaps that keyword matching cannot. Notably, semantic search outperformed the structured index on medium queries despite operating at the chunk level.

Token Economics

The most striking finding: semantic search consumes **37–44x fewer tokens** (~1,058) than grep (~46,442) or the structured index (~39,594). An agent making 10 documentation lookups per session would consume ~464K tokens with grep vs ~10K with semantic search, a 46x reduction that translates directly to cost and context window savings.

Results, Hierarchical Chunking

After adding LLM-generated doc_summary and section_summary chunks (§6.3) alongside the original content chunks, the index grew from 14,544 content chunks to 16,781 total chunks (491 doc summaries, 1,746 section summaries). Summaries were generated by Llama 3.3 70B via Databricks FMAPI.

Note: The baseline results in §10.2 were measured against an index that contained duplicate content chunks due to a file-discovery bug. The corrected content-only baseline (14,544 unique content chunks from 564 docs) is comparable to the hierarchical results below, as the deduplication fix was applied before this measurement.

Hierarchical Overall Performance

Metric	Grep	Structured Index	Semantic (Hierarchical)
Avg Tokens	46,599	39,594	1,100
Avg Time (sec)	1.79	0.05	5.59
Avg Precision@5	0.107	0.200	**0.127**
Avg MRR	0.172	0.261	**0.083**

Hierarchical Performance by Difficulty Tier

Tier	Grep P@5	Structured Index P@5	Semantic P@5 (Baseline)	Semantic P@5 (Hierarchical)	Delta
Easy	0.280	0.200	0.133	0.133	
Medium	0.000	0.160	0.167	**0.197**	+18%
Hard	0.040	0.240	0.050	0.050	

Per-Query Comparison

Query	Tier	Baseline P@5	Hierarchical P@5	Change
Q03: Find tenant path resolver	easy	0.67	0.67	
Q06: Tenant isolation in Unity Catalog	medium	0.00	**0.20**	+0.20
Q07: Engine abstraction layer	medium	0.25	0.25	
Q08: Permission resolution for tenant user	medium	0.33	0.20	-0.13
Q09: Embedding model for doc search	medium	0.25	**0.33**	+0.08
Q12: Three project organization strategies	hard	0.25	0.25	

Hierarchical Chunking Analysis

Hypothesis partially confirmed. Hierarchical summaries improved Medium-tier queries (P@5: 0.167 → 0.197, +18%) as predicted. The gains came from document-level summary chunks matching conceptual queries more precisely than raw content chunks. Q06 ("How does tenant isolation work in Unity Catalog?") went from zero precision to 0.20, the doc_summary chunk for the tenant isolation design document captured the document's purpose in a way that content chunks did not.

Easy and Hard tiers unchanged. Easy queries (keyword lookups) continued to match via content chunks, summaries neither helped nor hurt. Hard queries (cross-cutting architectural questions) were not helped because they require synthesizing information across multiple documents, which single-document summaries cannot address.

Minor regression on Q08. Permission resolution dropped from 0.33 to 0.20. The doc_summary for an unrelated document scored higher than a content chunk from the correct file, displacing it from the top-5. This is the expected trade-off: summaries are compressed representations that may match broadly, occasionally outranking more specific content chunks.

Token cost stable. Average tokens per query increased marginally (1,058 → 1,100, +4%) due to slightly longer summary chunk text in results. The 42x token advantage over grep is preserved.

Pipeline Economics

Phase	Time	Semantic Search
LLM Summarization (first run)	~30 min	2,237 Llama 3.3 70B calls (~$0.15)
LLM Summarization (incremental)	<1 sec	Cache hit, 0 LLM calls
Embedding (22,960 chunks)	~3 min	FMAPI pay-per-token (~$0.02)
Delta Write	~2 min	Serverless SQL Warehouse
Total (first run)	**~51 min**	**~$0.17**
Total (incremental)	**~8 min**	**~$0.02**

Incremental caching reduced subsequent pipeline runs from 51 minutes to 8 minutes by reusing LLM-generated summaries stored in the Delta table. Only documents with changed content trigger new LLM calls.

Lessons Learned and Future Directions

The benchmark results, combined with the experience of building and operating this system, surfaced several insights that are not obvious from the numbers alone.

Key Takeaways

Embeddings transform documentation from static files into queryable knowledge. The shift from keyword search to semantic search is qualitative, not just quantitative. Agents that previously required 10–20 file reads to find relevant documentation now retrieve it in a single query. This reduces context window consumption, speeds up agent responses, and improves the accuracy of agent-generated code.

The most expensive vector database is not always the right one. Documentation search is episodic and low-throughput. The cost-optimal architecture uses infrastructure that scales to zero, not infrastructure that provides millisecond latency at all times. Databricks SQL Warehouse, originally designed for analytics queries, supports higher-order array functions (zip_with, aggregate, transform) that can compute cosine similarity directly on ARRAY<FLOAT> columns. This incidental capability, standard SQL primitives repurposed for vector math, is sufficient for documentation workloads.

Databricks-native means zero additional operational burden. No new credentials to manage. No new vendor contracts. No new monitoring dashboards. The Delta table is governed by Unity Catalog. The SQL Warehouse is managed by the workspace. The FMAPI is authenticated by the same tokens used for every other Databricks operation. The operational surface area of this system is effectively zero.

Chunking strategy matters more than embedding model choice. The benchmark results show that flat content chunking achieves P@5=0.117 on a 14,544-chunk corpus, lower than the structured index approach (P@5=0.200) which uses keyword matching against LLM-distilled document-level summaries. Adding hierarchical summaries (doc + section level) improved semantic search to P@5=0.127 (+8.5%), with the largest gains on medium-difficulty conceptual queries (+18%). The bottleneck is not embedding quality but chunk granularity: content chunks capture what paragraphs say, while agents often need to know what documents are about.

Pre-computed reasoning compounds. The structured index system's advantage comes from LLM reasoning performed at build time, summarizing documents, grouping them into clusters, annotating entry points. This reasoning is done once and reused across every agent query. Hierarchical chunking applies the same principle within the embedding pipeline: summarization at build time produces chunks that match query intent more directly than raw content.

Future Directions

Incremental indexing is already partially implemented. The pipeline caches LLM-generated summaries in the Delta table and reuses them on subsequent runs when document content is unchanged, matched by content hash fingerprint. This reduced incremental run time from ~51 minutes to ~8 minutes and eliminated redundant FMAPI calls. Full re-embedding still occurs each run; a future optimization could skip re-embedding for unchanged chunks.

Hybrid search represents the highest-leverage precision improvement. Combining BM25 keyword scores (30% weight) with semantic similarity scores (70% weight) would improve results for queries that include exact technical terms, function names, error codes, configuration keys, that benefit from literal matching.

Source code indexing would close the corpus coverage gap identified in §10.3. Extending the pipeline to discover and chunk .py files with code-aware chunking, splitting at class/function boundaries, using docstrings as primary text, would enable semantic search to answer implementation-level queries that currently only grep can serve.

Beyond these, three additional enhancements are under consideration. The Model Context Protocol (MCP) would allow agents like Claude Code to invoke doc-search natively without executing shell commands. Cross-repository search could federate documentation indexes across multiple Databricks workspaces via shared Delta tables. And automatic freshness weighting, boosting scores for documents with recent last_updated timestamps, would ensure agents prefer current documentation over outdated content.

Recommendations

Lessons and future directions identify what is possible. Recommendations prioritize what to do first. The core principle: optimize for token efficiency, then incrementally close precision gaps through targeted enhancements rather than wholesale replacement.

Use Semantic Search as the Primary Approach

The 42x token savings (1,100 vs 46,000 tokens per query) is the dominant factor. At 10 documentation lookups per agent session, semantic search consumes ~11,000 tokens vs ~465,000 for grep, the difference between fitting the answer in the agent's context window and exceeding it. P@5=0.127 is sufficient for most queries, and the zero-curation, zero-ops operational model makes it sustainable.

Invest in Hybrid Search Next

The hierarchical chunking experiment showed that better summaries improve medium-tier conceptual queries (+18%) but do not address easy-tier keyword queries where semantic search scores 0.00 (e.g., specific configuration variables). The next precision improvement will come from **combining BM25 keyword scores with cosine similarity,** a hybrid approach that catches literal identifier matches that embeddings miss. Implementation: add a BM25 index column to the Delta table; at query time, compute a weighted score (e.g., 0.7 x cosine + 0.3 x BM25).

Expand Corpus Coverage to Source Code

Seven of fifteen benchmark queries have .py source files in their gold answers. Semantic search structurally cannot find them because the pipeline only indexes markdown. Adding code-aware chunking, splitting at class/function boundaries, using docstrings as the primary embedding text, would close the Easy-tier gap where grep leads (P@5=0.280 vs 0.133). This is the single largest addressable gap in the benchmark

Do Not Abandon Structured Indexes

The structured index system achieved the highest overall precision (P@5=0.200, MRR=0.261) and excels at the hardest cross-cutting architectural queries. Its weakness is maintenance cost, not quality. The long-term play is to **auto-generate structured index metadata from the same LLM pipeline** that produces hierarchical summaries, unifying the two approaches so structured navigation and semantic similarity share the same build-time reasoning.

Do Not Chase Summary Quality

The benchmark showed diminishing returns from richer summaries. The jump from no summaries (P@5=0.117) to hierarchical summaries (P@5=0.127) was +8.5%, meaningful but modest. Further investment in summary sophistication (e.g., multi-document summaries, query-aware summaries) is unlikely to yield comparable returns. The higher-leverage investments are hybrid scoring and corpus expansion, which address structural gaps that no amount of summarization can fix.

Priority Sequence

Priority	Enhancement	Expected Impact	Effort
1	Hybrid BM25 + cosine scoring	+30-50% on Easy-tier queries	Medium
2	Source code (.py) indexing	+20-40% on Easy-tier, new coverage	Medium
3	Auto-generated structured indexes from pipeline	Unifies navigation + search	High
4	MCP tool integration	Better agent UX, no shell execution	Low
5	Cross-repository federation	Multi-workspace search	High

Taken together, these recommendations form a roadmap from the current system (P@5=0.127) toward a hybrid search capability that combines the 42x token efficiency of semantic search with the precision of keyword-aware retrieval and the coverage of source code indexing. The foundation built here, a zero-idle-cost Delta-native index with hierarchical summaries, provides a stable base for all of these enhancements.

Related Work

The architecture, chunking strategies, and evaluation methodology described in this paper draw on established research in information retrieval, dense vector search, and LLM-augmented retrieval. This section situates the system within that literature and notes where the approach deliberately simplifies the standard techniques.

Dense Retrieval and Retrieval-Augmented Generation

The theoretical foundation for embedding-based search lies in the dense passage retrieval literature. Karpukhin et al. (2020) demonstrated with DPR (Dense Passage Retrieval) that bi-encoder models, where queries and documents are embedded independently and relevance computed by dot product, could match or exceed BM25 for open-domain question answering. This established the fundamental pattern used throughout the pipeline: embed at index time, embed the query at search time, rank by cosine similarity.

Lewis et al. (2020) introduced Retrieval-Augmented Generation (RAG) in “Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks,” combining dense retrieval with sequence-to-sequence generation to ground LLM outputs in factual documents. RAG demonstrated that pre-indexed document corpora could be queried at inference time to supplement parametric model knowledge, the same pattern that underlies agent documentation discovery, except that the agent uses the retrieved passages directly rather than fusing them through a generator.

Sentence-BERT (Reimers & Gurevych, 2019) established the siamese fine-tuning approach that produces semantically aligned sentence embeddings suited for symmetric retrieval tasks. The databricks-gte-large-en model used in this system follows in this lineage, optimized for retrieval tasks as measured on the MTEB (Massive Text Embedding Benchmark) suite.

Vector Database Infrastructure and ANN Search

Production vector search typically relies on approximate nearest neighbor (ANN) indexes to avoid exhaustive linear scan over large corpora. FAISS (Johnson, Douze & Jegou, 2019) introduced GPU-accelerated similarity search across multiple index types (flat, IVF, HNSW, PQ), establishing the reference implementation for billion-scale ANN. HNSW (Malkov & Yashunin, 2018) demonstrated that graph-based navigation through a layered small-world structure could achieve sub-linear query time with high recall, and HNSW is now the default index type in most commercial vector databases (Pinecone, Weaviate, Qdrant, Databricks Vector Search).

Covasant Auraa deliberately avoids ANN indexes. For a corpus of 16,000–20,000 chunks, exhaustive cosine similarity over an ARRAY<FLOAT> column executes in 1–3 seconds on a serverless SQL Warehouse, sufficient for documentation search, which does not require millisecond latency. The resulting architecture is analogous to pgvector on PostgreSQL: standard database infrastructure, zero ANN tuning, and no additional operational dependencies. Research on embedding model benchmarks (Muennighoff et al., MTEB, 2023) has shown that at this corpus scale, retrieval quality is dominated by embedding model choice rather than index architecture, making the simplification well justified.

Hybrid Retrieval: Sparse and Dense

BM25 (Robertson & Zaragoza, "The Probabilistic Relevance Framework," 2009) remains a competitive baseline for exact-term retrieval, particularly on queries containing technical identifiers, function names, and configuration keys, precisely the Easy-tier queries where semantic search performs poorly (P@5=0.133 vs grep's 0.280). A substantial body of work on hybrid retrieval demonstrates that combining BM25 sparse scores with dense cosine similarity consistently outperforms either method independently, with improvements largest on domain-specific technical corpora.

SPLADE (Formal et al., 2021) extended BM25 toward learned sparse representations, while ColBERT (Khattab & Zaharia, 2020) introduced late interaction, per-token embeddings with MaxSim scoring, achieving dense retrieval precision at efficient re-ranking cost. Both represent the frontier of first-stage retrieval quality, though at greater infrastructure complexity than simple BM25 hybrid weighting.

The recommendation for BM25+cosine hybrid (0.7 x dense + 0.3 x sparse) is a conservative step toward this literature rather than an adoption of its frontier techniques. For a 16,000-chunk corpus with low query volumes, BM25 hybrid implemented as a SQL expression alongside existing cosine similarity provides the largest quality gain for the smallest implementation cost.

Hierarchical Chunking and LLM-Augmented Indexing

The chunk dilution problem identified in §11.1, where a comprehensive document's relevance signal is diluted across many small content chunks, is well recognized in the RAG literature. Parent-child chunking (implemented in frameworks such as LlamaIndex and LangChain) addresses this by indexing small chunks for precision but returning their parent documents for context. Covasant Auraa's approach is analogous: doc_summary and section_summary chunks are embedded for precision matching while the associated file path and heading breadcrumb guide agent navigation.

HyDE (Gao et al., 2022, "Precise Zero-Shot Dense Retrieval without Relevance Labels") proposed embedding a hypothetical document generated by the LLM as the query representation, improving alignment between query intent and document embedding space. The hierarchical summarization approach inverts this strategy: rather than transforming the query, the documents are transformed at index time, LLM-generated purpose descriptions are embedded alongside raw content, pre-computing the alignment that HyDE achieves dynamically.

The structured index approach evaluated in §10 represents an independent discovery of the same principle: pre-computed LLM reasoning at build time produces better query-time alignment than raw content embeddings. That this pattern was arrived at empirically through the journey described in §2, and later confirmed by its parallel in the academic literature, supports its validity as a general design principle for LLM-assisted retrieval systems.

Documentation Search for Software Engineering

The broader problem of code and documentation search has received attention through models such as CodeBERT, UniXcoder, and CodeT5, which train on code-documentation pairs to support natural-language-to-code retrieval. These models target finding implementations given intent descriptions, the inverse of the problem addressed here, which is finding documentation that explains existing implementations.

The BEIR benchmark (Thakur et al., 2021) established that general-purpose dense retrieval models transfer well to domain-specific retrieval tasks without fine-tuning, including technical question answering and fact verification. This motivates the choice of a general-purpose hosted model over a fine-tuned code documentation retrieval model: at the current corpus scale and query volume, the operational simplicity of a hosted general model outweighs the marginal accuracy gain from task-specific fine-tuning.

AI Usage Disclosure

This whitepaper was produced through active collaboration between human authors and AI coding and writing assistants. AI tools contributed to research, code generation, benchmark execution, analysis, and document drafting throughout the project. Specific contributions are noted below.

Tools and Contributions

Claude Code (Anthropic) The primary development environment for this project. Claude Code assisted with pipeline architecture, Python implementation (chunking, summarization, embedding, Delta write), debugging multi-stage failures, benchmark script development, and the drafting and revision of this whitepaper. The benchmark results, code, and analysis were reviewed and directed by the human author at each stage.

GitHub Copilot (Microsoft) Used during editor-integrated coding sessions for code completion, inline suggestions, and refactoring assistance across Python and YAML files in the monorepo.

Microsoft 365 Copilot Used for document drafting, structural review, and prose refinement at various stages of whitepaper preparation.

Nature of AI Collaboration

The research questions, architectural decisions, evaluation methodology, and conclusions in this whitepaper reflect the judgment of the human author. AI assistants were directed to implement, test, and document specific approaches, they did not independently determine what to measure or what conclusions to draw. All benchmark numbers were produced by running code against live Databricks infrastructure and are reproducible.

The use of AI coding assistants is itself relevant to the subject matter of this paper: the experience of directing Claude Code to build and evaluate a documentation search system while Claude Code simultaneously used (and was constrained by the limitations of) the search system it was building informed several of the design decisions described in §2.

References

Dense Retrieval and RAG

- Karpukhin, V., Oguz, B., Min, S., Lewis, P., Wu, L., Edunov, S., Chen, D., & Yih, W. (2020). "Dense Passage Retrieval for Open-Domain Question Answering." EMNLP 2020. https://arxiv.org/abs/2004.04906
- Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2020). "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks." NeurIPS 2020. https://arxiv.org/abs/2005.11401
- Reimers, N., & Gurevych, I. (2019). "Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks." EMNLP 2019. https://arxiv.org/abs/1908.10084
- Gao, L., Ma, X., Lin, J., & Callan, J. (2022). "Precise Zero-Shot Dense Retrieval without Relevance Labels." ACL 2023. https://arxiv.org/abs/2212.10496

Vector Search and ANN

- Johnson, J., Douze, M., & Jégou, H. (2019). "Billion-Scale Similarity Search with GPUs." IEEE Transactions on Big Data. https://arxiv.org/abs/1702.08734
- Malkov, Y. A., & Yashunin, D. A. (2018). "Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs." IEEE TPAMI. https://arxiv.org/abs/1603.09320

Hybrid Retrieval

- Robertson, S., & Zaragoza, H. (2009). "The Probabilistic Relevance Framework: BM25 and Beyond." Foundations and Trends in Information Retrieval, 3(4).
- Formal, T., Piwowarski, B., & Clinchant, S. (2021). "SPLADE: Sparse Lexical and Expansion Model for First Stage Ranking." SIGIR 2021. https://arxiv.org/abs/2107.05720
- Khattab, O., & Zaharia, M. (2020). "ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT." SIGIR 2020. https://arxiv.org/abs/2004.12832

Benchmarks

- Muennighoff, N., Tazi, N., Magne, L., & Reimers, N. (2023). "MTEB: Massive Text Embedding Benchmark." EACL 2023. https://arxiv.org/abs/2210.07316
- Thakur, N., Reimers, N., Rücklé, A., Srivastava, A., & Gurevych, I. (2021). "BEIR: A Heterogeneous Benchmark for Zero-shot Evaluation of Information Retrieval Models." NeurIPS 2021. https://arxiv.org/abs/2104.08663

Software Engineering Code Search

- Feng, Z., Guo, D., Tang, D., et al. (2020). "CodeBERT: A Pre-Trained Model for Programming and Natural Languages." EMNLP Findings 2020. https://arxiv.org/abs/2002.08155

Databricks Platform Documentation

- Delta Lake Documentation
- Foundation Model API
- Serverless SQL Warehouse
- Unity Catalog Overview

Companion Documents

- Semantic Documentation Search (Internal Reference), Implementation-level reference with proprietary code paths and internal schemas.
- Discovery Benchmark Results, Detailed benchmark data supporting §10.

Document Version: 1.0 **Status:** Production **Last Updated:** April 4, 2026 **Authors:** Covasant Platform Team **Audience:** Architects, Data Engineers, Platform Engineers, Technical Leaders, Executive Stakeholders

Covasant

AI-Driven. Human-Inspired.

About Covasant

Covasant Technologies is an emerging global leader in delivering Agentic AI-led services that address complex, industry-specific challenges. Through its pioneering Services-as-Software model, Covasant brings together capabilities in data engineering, digital and cloud, AI engineering, and Enterprise Risk Management (ERM). Strategically located in innovation hubs including Hyderabad, Plano, New York, Los Angeles, London, and Dubai, Covasant leverages a premier talent ecosystem to deliver scalable, autonomous solutions. Our "Governance-Led" approach ensures that global enterprises can automate decision-making and orchestrate business processes with the reliability and speed required in today's market.

Plano, TX (USA) • London, UK • Hyderabad, India • Dubai, UAE

kathy.harnois@Covasant.com

alan.dennis@Covasant.com