Here is the Agile implementation plan for the backend. I have broken the work into **3 Sprints**, with small, isolated tasks suitable for individual branches.

Sprint 1: Foundation & Data Ingestion (The Producer)

Goal: Build the Core domain and the "Write" side of the system (Downloading, Parsing, and Saving data).

Task ID	Branch Name	Description & Files	Testing Strategy
1.1	feature/core-domain	**Implement Domain Entities & Enums** Create the foundational data structures. **Files:** Dataset.cs, MetadataRecord.cs, MetadataFormat.cs, Result.cs.	**Unit Tests:** Verify that entity constructors validation works (e.g., ensuring a Dataset cannot exist without an ID).
1.2	feature/infra-sqlite	**Implement SQLite Repository** Set up Entity Framework Core and the database schema. **Files:** AppDbContext.cs, SqliteMetadataRepository.cs.	**Integration Tests:** Use SqliteMetadataRepositoryTests.cs with an **In-Memory** SQLite connection to ensure CRUD operations work without a real disk file.

1.3	feature/parsing-strategies	**Implement Metadata Strategies** Create the parsers for XML, JSON-LD, etc. **Files:** MetadataParserFactory.cs, Iso19115XmlParser.cs, JsonExpandedParser.cs.	**Unit Tests:** Create ParserTests.cs. Feed raw sample XML/JSON strings into each parser and assert that the output MetadataRecord matches expected values.
1.4	feature/download-extract	**Implement Download & Zip Extraction** Handle HTTP streams and Zip file inspection. **Files:** CehDatasetDownloader.cs, ZipExtractionService.cs.	**Integration Tests:** Test ZipExtractionService by creating a small dummy .zip in the test setup, passing it to the service, and verifying it identifies the correct files inside.
1.5	feature/etl-orchestrator	**Implement ETL Workflow** Wire everything together: Download -> Unzip -> Parse -> Save. **Files:** EtlOrchestrator.cs, DatasetProcessor.cs.	**Unit Tests:** Mock all interfaces (IDownloader, IRepository). Test the *flow*: "If download succeeds, does it call Parse? If Parse succeeds, does it call Save?"

Sprint 2: Search & Intelligence (The Consumer)

Goal: Build the "Read" side of the system (Vector Storage, Search Logic, and RAG).

Task ID	Branch Name	Description & Files	Testing Strategy
2.1	feature/infra-vector	**Implement Vector Store Adapter** Connect to the Qdrant instance. **Files:** QdrantAdapter.cs, EmbeddingVector.cs.	**Integration Tests:** Use the Docker Qdrant container. Index a dummy vector and try to retrieve it via SearchAsync.
2.2	feature/search-service	**Implement Search Logic** Combine keyword search (SQLite) with semantic search (Qdrant). **Files:** SearchService.cs, SearchResultDto.cs.	**Unit Tests:** Mock the Repositories. Test logic like "If vector store returns ID 5, does the service fetch Metadata for ID 5 from SQLite?"
2.3	feature/rag-agent	**Implement Conversational Agent**	**Unit Tests:** Mock the ILlmClient. Verify that RagService constructs the prompt string correctly (e.g., "Context:

		Build the prompt engineering logic. **Files:** RagService.cs, OpenAiLlmClient.cs.	[Data]... Question: [User Query]").

Sprint 3: API & Delivery

Goal: Expose functionality to the web and harden the system.

Task ID	Branch Name	Description & Files	Testing Strategy
3.1	feature/api-controllers	**Implement Web API Endpoints** Connect HTTP requests to the Application Core. **Files:** SearchController.cs, ChatController.cs, AdminEtlController.cs.	**End-to-End Tests:** Use WebApplicationFactory. Spin up the full API in memory and send a real HTTP GET request to /api/search to ensure the full pipeline works.
3.2	feature/api-caching	**Implement Response Caching** Add middleware to cache frequent search results. **Files:** ResponseCachingMiddleware.cs.	**Integration Tests:** Send the same request twice. Verify the second request returns faster or has a specific "Cached" header.

3.3	chore/documentation	**Swagger & Readme** Finalize API documentation. **Files:** Program.cs (Swagger Config), README.md.	**Manual Verification:** Run the container and verify the Swagger UI is accessible.

Suggested First Step

I recommend we start with **Task 1.1 (Domain Foundation)**.

Command to start:

Bash

git checkout -b feature/core-domain

Would you like me to generate the **C# code for Task 1.1** (The Entity classes and Enums) so you can push the first branch?