RAG Workshop

.NET 10 + Qdrant + OpenAI

Build a complete **Retrieval-Augmented Generation** solution from scratch

📦 `github.com/PeterMilovcik/Qdrant.Demo`

Agenda

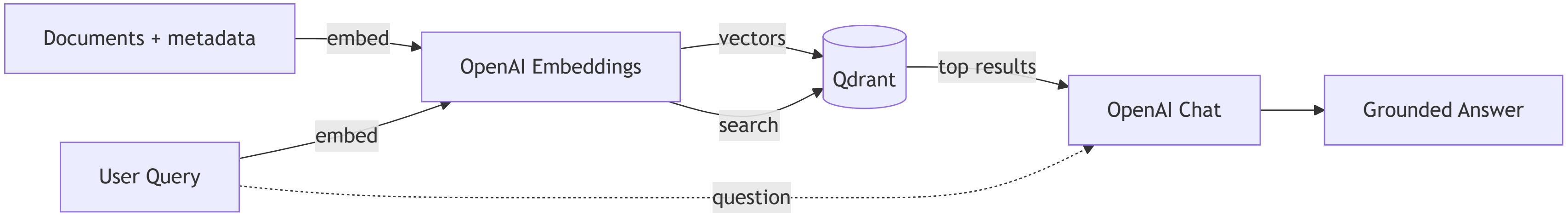
#	Module	Topic	⏱️
0	Setup	Docker, Qdrant Dashboard, Swagger UI	20 min
1	Index	Embeddings, deterministic IDs, batch indexing	25 min
2	Retrieval	Cosine similarity, metadata, tag filters, threshold	45 min
3	Generation	RAG pipeline, system prompts, filtered chat, score threshold	40 min
4	Chunking	Text splitting, overlap, sentence boundaries	30 min
5	User Interface	Static frontend, visual RAG experience	20 min

Total ≈ 3 hours at a comfortable pace

What is RAG?

Retrieval-Augmented Generation — a three-step pattern:

1. **Index** — Turn documents into vectors, store in a vector database
2. **Retrieve** — Embed the user’s query, find most similar documents
3. **Generate** — Feed retrieved documents into an LLM for a grounded answer



Why RAG?

Without RAG	With RAG
LLM answers from training data only	LLM answers from **your documents**
May hallucinate facts	Grounded — cites what's actually indexed
Generic answers	Specific to your domain
Training cutoff	Always current (re-index anytime)

Key insight: Don't hope the AI "knows" — **feed it the right documents first**.

Prerequisites

Tool	Version	Why
Docker Desktop	4.x+	Runs the Qdrant vector database
.NET 10 SDK	10.0+	Build & run the API locally
OpenAI API key	—	Embeddings + chat
`curl` or Swagger	—	Test the endpoints

Quick check

```
docker --version
```

```
dotnet --version
```

> **Cost note:** Expected cost is well under **$1** per participant for the full workshop.

Tech Stack

Component	Role
Qdrant (Docker)	Open-source vector database — stores embeddings + metadata
.NET 10 Minimal API	Exposes indexing, search, and chat endpoints
OpenAI Embeddings	`text-embedding-3-small` — 1536 dimensions
OpenAI Chat	`gpt-4o-mini` — generates grounded answers
Docker Compose	Runs Qdrant in a container

Each module folder (`module-XX/`) is **self-contained** — its own `README.md`, solution, source, and tests.

Module 0

Setup

~20 min · No LLM needed · No code to write

Module 0 — Qdrant Concepts

Qdrant is an open-source vector database. It stores **points**:

Component	Description
id	UUID or integer — uniquely identifies each point
vector	Array of floats (1536 dimensions for our model)
payload	Key/value metadata (text, timestamps, tags)

> Points live inside **collections** — like a database table. All vectors in a collection share the same dimensionality.

Module 0 — QdrantBootstrapper (1/2)

A background service that creates the collection at startup with retries:

```csharp
sealed class QdrantBootstrapper(QdrantClient qdrant, string collection, int dim)
    : BackgroundService
{
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        for (var attempt = 1; attempt <= 30; attempt++)
        {
            try
            {
                await EnsureCollectionAsync(stoppingToken);
                Console.WriteLine($"[bootstrap] Collection '{collection}' ready.");
                return;
            }
            catch (Exception ex) when (!stoppingToken.IsCancellationRequested)
            {
                Console.WriteLine($"[bootstrap] attempt {attempt} failed: {ex.Message}");
                await Task.Delay(TimeSpan.FromSeconds(1), stoppingToken);
            }
        }
    }
```

30 retries × 1 s delay — tolerates Qdrant starting slowly.

Module 0 — QdrantBootstrapper (2/2)

```
    private async Task EnsureCollectionAsync(CancellationToken ct)
    {
        try {
            await qdrant.CreateCollectionAsync(collection,
                new VectorParams { Size = (uint)dim, Distance = Distance.Cosine },
                cancellationToken: ct);
        }
        catch (RpcException ex) when (ex.StatusCode == StatusCode.AlreadyExists) { }
    }
}
```

- `AlreadyExists` catch → idempotent — safe to restart the API
- `Distance.Cosine` → similarity metric for semantic search
- **1536 dimensions** → matches `text-embedding-3-small`

Module 0 — Program.cs (Skeleton)

```csharp
var qdrantHost     = config["QDRANT_HOST"]       ?? config["Qdrant:Host"]       ?? "localhost";
var qdrantGrpcPort = int.Parse(config["QDRANT_GRPC_PORT"] ?? config["Qdrant:GrpcPort"] ?? "6334");
var collectionName = config["QDRANT_COLLECTION"] ?? config["Qdrant:Collection"] ?? "documents";

// Register Qdrant client
builder.Services.AddSingleton(_ => new QdrantClient(qdrantHost, qdrantGrpcPort));

// Start bootstrapper — creates collection with retries
builder.Services.AddHostedService(sp =>
    new QdrantBootstrapper(
        sp.GetRequiredService<QdrantClient>(),
        collectionName,
        embeddingDim));

// Endpoints
app.MapInfoEndpoints(new { service = "Qdrant.Demo.Api", ... });
```

- Config reads from `appsettings.json` with **env-var overrides**
- Pattern: `ENV_VAR → appsettings key → default value`

Module 0 — Steps

1. Start Qdrant

```
cd module-00
```

```
docker compose up -d
```

```
curl http://localhost:6333/healthz
```

2. Run the API

```
dotnet run --project src/Qdrant.Demo.Api
```

3. Explore

- Swagger UI → `http://localhost:8080/swagger`
- Qdrant Dashboard → `http://localhost:6333/dashboard`
- Verify: `documents` collection exists, 0 points, 1536 dim, Cosine distance

Module 1

Index

~25 min · Requires OpenAI API key

Module 1 — What Are Embeddings?

An **embedding** is a list of floats that captures the **meaning** of text.

`text-embedding-3-small` → **1536 floats** per input

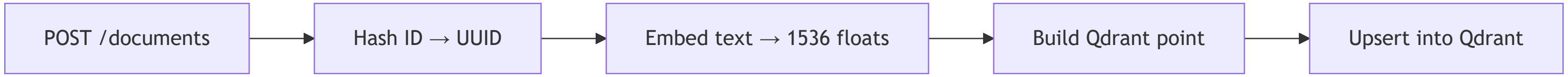
Key insight

> Texts with **similar meaning** produce vectors that are **close together** in vector space.

Text A	Text B	Similar?
"The cat sat on the mat"	"A kitten was sitting on a rug"	✅ Very close
"The cat sat on the mat"	"Stock prices rose yesterday"	❌ Very far

This is what makes **semantic search** possible — compare meaning, not keywords.

Module 1 — The Indexing Pipeline



Deterministic point-IDs

- If caller provides `id` → `SHA256("article-001")` → same UUID every time
- If no `id` → `SHA256(text)` → same text = same point

> **Re-indexing the same document is safe** — it overwrites, never duplicates. This is an **idempotent upsert**.

💡 **Upsert** = **update** + **insert** — if the ID exists, update it; if not, insert it.

Module 1 — StringExtensions.cs

Deterministic UUID via SHA-256:

```
public static Guid ToDeterministicGuid(this string input)
{
    var hash = SHA256.HashData(Encoding.UTF8.GetBytes(input));

    Span<byte> g = stackalloc byte[16];
    hash.AsSpan(0, 16).CopyTo(g);

    g[6] = (byte)((g[6] & 0x0F) | 0x50);  // version 5
    g[8] = (byte)((g[8] & 0x3F) | 0x80);  // RFC 4122 variant

    return new Guid(g);
}
```

Same input → same GUID, every time. **No duplicates, ever.**

Module 1 — EmbeddingService.cs

A thin wrapper around Microsoft.Extensions.AI:

```
public sealed class EmbeddingService(
    IEmbeddingGenerator<string, Embedding<float>> generator) : IEmbeddingService
{
    public async Task<float[]> EmbedAsync(string text, CancellationToken ct = default)
    {
        var embedding = await generator.GenerateAsync(
            [text], cancellationToken: ct);
        return embedding[0].Vector.ToArray();
    }
}
```

- Depends on the `IEmbeddingGenerator` **interface** (not the OpenAI SDK directly)
- Easy to mock in unit tests
- One text in → one 1536-float vector out

Module 1 — DocumentIndexer.cs

The orchestrator: hash → embed → build point → upsert:

```
var idSource = !string.IsNullOrWhiteSpace(request.Id)
    ? request.Id! : request.Text;
var pointId = idSource.ToDeterministicGuid().ToString("D");

var vector = await embeddings.EmbedAsync(request.Text, ct);

var point = new PointStruct
{
    Id = new PointId { Uuid = pointId },
    Vectors = vector,
    Payload =
    {
        [Text] = request.Text,
        [IndexedAtMs] = DateTime.UtcNow.ToUnixMs()
    }
};

await qdrant.UpsertAsync(collectionName, [point], wait: true, cancellationToken: ct);
```

`wait: true` → Qdrant confirms the write is durable before returning.

Module 1 — DocumentEndpoints.cs

```
app.MapPost("/documents", async (
    [FromBody] DocumentUpsertRequest req,
    IDocumentIndexer indexer,
    CancellationToken ct) =>
{
    if (string.IsNullOrWhiteSpace(req.Text)) return Results.BadRequest("Text is required and cannot be empty.");
    try
    {
        var response = await indexer.IndexAsync(req, ct);
        return Results.Ok(response);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"[documents] Error: {ex.Message}");
        return Results.Problem(detail: ex.Message, statusCode: 500, title: "Indexing failed");
    }
});
```

Request: `{ "id": "article-001", "text": "Photosynthesis is..." }`

Response: `{ "pointId": "a1b2c3d4-..." }`

Module 1 — Batch Endpoint

```csharp
app.MapPost("/documents/batch", async (
    [FromBody] IReadOnlyList<DocumentUpsertRequest> batch,
    IDocumentIndexer indexer, CancellationToken ct) =>
{
    List<string> errors = [];
    var succeeded = 0;
    foreach (var req in batch)
    {
        if (string.IsNullOrWhiteSpace(req.Text))
        {
            errors.Add($"[{req.Id ?? "(empty)"}]: Text is required.");
            continue;
        }
        try { await indexer.IndexAsync(req, ct); succeeded++; }
        catch (Exception ex)
        {
            errors.Add($"[{req.Id ?? req.Text[..Math.Min(req.Text.Length, 40)]}]: {ex.Message}");
        }
    }
    return Results.Ok(new BatchUpsertResponse(
        batch.Count, succeeded, errors.Count, errors));
});
```

Partial failure: one bad document doesn’t block the rest. Errors are collected with labels for easy debugging.

Module 1 — Request & Response Models

```
public record DocumentUpsertRequest(
    string? Id,
    string Text,
    Dictionary<string, string>? Tags = null,
    Dictionary<string, string>? Properties = null
);

public record DocumentUpsertResponse(
    string PointId,
    int TotalChunks = 1,
    IReadOnlyList<string>? ChunkPointIds = null
);
```

- `Id` is optional — if omitted, the text itself is hashed
- `Tags` and `Properties` are declared now but used starting in Module 2
- `TotalChunks` and `ChunkPointIds` are used starting in Module 4

> Forward-compatible design — the DTOs grow with the workshop.

Module 1 — Program.cs (DI Setup)

```csharp
var openAi = new OpenAIClient(openAiApiKey);

// Embedding generator
builder.Services.AddSingleton<IEmbeddingGenerator<string, Embedding<float>>>(
    openAi.GetEmbeddingClient(embeddingModel).AsIEmbeddingGenerator());

// Services
builder.Services.AddSingleton<IEmbeddingService, EmbeddingService>();

// Document indexer — needs QdrantClient, EmbeddingService, collection name
builder.Services.AddSingleton<IDocumentIndexer>(sp =>
    new DocumentIndexer(
        sp.GetRequiredService<QdrantClient>(),
        sp.GetRequiredService<IEmbeddingService>(),
        collectionName));

// Register endpoints
app.MapDocumentEndpoints();
```

- `AsIEmbeddingGenerator()` adapts the OpenAI SDK to the `Microsoft.Extensions.AI` interface
- The indexer is registered with a factory to inject the collection name

Module 1 — Try It

1. Set your OpenAI API key

```
$env:OPENAI_API_KEY = "sk-..."
```

2. Start & run

```
cd module-01
docker compose up -d
dotnet run --project src/Qdrant.Demo.Api
```

3. Index a single document via Swagger (`POST /documents`)

```
{ "id": "article-001", "text": "Photosynthesis converts sunlight into energy." }
```

4. Batch index via Swagger (`POST /documents/batch`)

```
[
  {"id": "article-002", "text": "Quantum entanglement links particles instantly."},
  {"id": "article-003", "text": "Machine learning finds patterns in data."}
]
```

5. Check the Qdrant Dashboard — **3 points** in the collection

Module 2

Retrieval

~45 min

Module 2 — Cosine Similarity

Measures the **angle** between two vectors:

Score	Meaning
1.0	Identical meaning
0.7+	Highly relevant
0.4–0.7	Somewhat related
< 0.3	Likely unrelated

Semantic ≠ Keyword

Query	Best match	Why?
"How do plants make food?"	Photosynthesis article	Same **meaning**, different words
"training algorithms on datasets"	Machine learning article	Same **concept**

> Top-K returns exactly K results ranked by similarity (highest first).

Module 2 — SearchEndpoints.cs (Top-K)

```
app.MapPost("/search/topk", async (
    [FromBody] TopKSearchRequest req,
    QdrantClient qdrant, IEmbeddingService embeddings, CancellationToken ct) =>
{
    var vector = await embeddings.EmbedAsync(req.QueryText, ct);

    var hits = await qdrant.SearchAsync(
        collectionName: collectionName,
        vector: vector,
        limit: (ulong)req.K,
        payloadSelector: true,
        cancellationToken: ct);

    return Results.Ok(hits.ToFormattedHits());
});
```

Pipeline: Embed query → Search Qdrant → Return ranked hits

`payloadSelector: true` includes the stored payload (text, timestamp) in each result.

Module 2 — QdrantPayloadExtensions.cs

Converts gRPC protobuf values to clean API responses:

```csharp
public static IEnumerable<SearchHit> ToFormattedHits(this IReadOnlyList<ScoredPoint> hits)
{
    return hits.Select(h => new SearchHit(
        Id: h.Id?.Uuid ?? h.Id?.Num.ToString(),
        Score: h.Score,
        Payload: h.Payload.ToDictionary()
    ));
}

private static object? FromProto(Value v) => v.KindCase switch
    {
        Value.KindOneofCase.StringValue  => v.StringValue,
        Value.KindOneofCase.DoubleValue  => v.DoubleValue,
        Value.KindOneofCase.IntegerValue => v.IntegerValue,
        Value.KindOneofCase.BoolValue    => v.BoolValue,
        Value.KindOneofCase.StructValue  => v.StructValue.Fields.ToDictionary(f => f.Key, f => FromProto(f.Value)),
        Value.KindOneofCase.ListValue    => v.ListValue.Values.Select(FromProto).ToList(),
        _ => null
    };
```

Module 2 — Try It

Search via Swagger — `POST /search/topk`:

```
{ "queryText": "How do plants produce energy from sunlight?", "k": 3 }
```

Rank	Document	Score
1	Photosynthesis article	~0.64
2	Machine learning article	~0.10
3	Quantum article	~0.09

Also try:

- `"spooky action at a distance"` → matches quantum (Einstein's phrase!)
- `"training algorithms on datasets"` → matches ML
- `"best pizza recipe"` → low scores on everything (no minimum filter yet)

Module 2 — Exercises

Exercise 2.1 — Different queries

Try `"spooky action at a distance"` and `"training algorithms"`. Observe which document scores highest.

Exercise 2.2 — Change K

K=1 → only the best match. K=10 → at most 3 (your collection size).

Exercise 2.3 — Pizza test

`"best pizza recipe"` → still returns 3 results, but scores are very low. Top-K **always** returns K results — no minimum filter yet.

> **Problem:** How do we exclude irrelevant results? → Threshold search, later in this module.

Module 2 — Tags vs Properties

	Tags	Properties
Prefix	`tag_{key}`	`prop_{key}`
Purpose	Filtering during search	Displayed with results
Indexed?	✅ Yes — used in filter clauses	❌ No — stored only
Example	`"category": "science"`	`"source_url": "https://…"`

Example payload after indexing

```json
{
  "text": "Photosynthesis converts sunlight…",
  "indexed_at_ms": 1718500000000,
  "tag_category": "biology",  "tag_level": "introductory",
  "prop_source_url": "https://example.com/bio",  "prop_author": "Dr. Green"
}
```

Why prefixes? Qdrant uses a flat payload. Prefixes avoid collisions and make filter building automatic.

Module 2 — Storing Metadata

PayloadKeys.cs — Constants

```
public static class PayloadKeys
{
    public const string Text           = "text";
    public const string IndexedAtMs    = "indexed_at_ms";
    public const string TagPrefix      = "tag_";
    public const string PropertyPrefix = "prop_";
}
```

DocumentIndexer.cs — Storage loop

```
if (request.Tags is not null)
    foreach (var (key, value) in request.Tags)
        point.Payload[$"{TagPrefix}{key}"] = value;

if (request.Properties is not null)
    foreach (var (key, value) in request.Properties)
        point.Payload[$"{PropertyPrefix}{key}"] = value;
```

Backward compatible — existing callers that don’t send tags/properties are unaffected.

Module 2 — Try It (Metadata)

Index with metadata (`POST /documents`)

```
{
  "id": "bio-001",
  "text": "Photosynthesis converts sunlight into energy in plants.",
  "tags": { "category": "biology", "level": "introductory" },
  "properties": { "source_url": "https://example.com/bio", "author": "Dr. Green" }
}
```

Search and verify (`POST /search/topk`)

```
{ "queryText": "photosynthesis", "k": 1 }
```

You should see `tag_category`, `tag_level`, `prop_source_url` in the payload.

> Tags are stored but search still returns **all** documents ranked by similarity. **Filtering comes next.**

Module 2 — Three Search Strategies

Endpoint	How it works
`POST /search/topk`	Fixed K results + optional tag filter
`POST /search/threshold`	All results above a minimum score
`POST /search/metadata`	Tag-only browse — no vectors involved

Pre-filtering

Qdrant applies tag filters **before** computing similarity:

```
{ "queryText": "energy", "k": 5, "tags": { "category": "biology" } }
```

Only biology documents are scored. Physics is excluded before similarity is even calculated.

Module 2 — QdrantFilterFactory.cs

The bridge between tag dictionaries and Qdrant filter objects:

```
public Filter? CreateGrpcFilter(Dictionary<string, string>? tags)
{
    if (tags is null || tags.Count == 0) return null;

    var filter = new Filter();
    foreach (var (key, value) in tags)
        filter.Must.Add(MatchKeyword($"tag_{key}", value));
    return filter;
}
```

- `null` tags → `null` filter → no filtering (search all documents)
- Multiple tags → AND logic (`Must` clause)
- Each tag becomes a `MatchKeyword` condition on `tag_{key}`

Module 2 — Threshold Search

```
app.MapPost("/search/threshold", async ( … ) ⇒
{
    var vector = await embeddings.EmbedAsync(req.QueryText, ct);
    var filter = filters.CreateGrpcFilter(req.Tags);

    var hits = await qdrant.SearchAsync(
        collectionName, vector, limit: (ulong)req.Limit,
        filter: filter,
        scoreThreshold: req.ScoreThreshold,
        payloadSelector: true, cancellationToken: ct);

    return Results.Ok(hits.ToFormattedHits());
});
```

Returns **all** documents with similarity ≥ threshold (default 0.4).

`Limit` (default 100) acts as a safety cap.

Module 2 — Metadata-Only Search

```csharp
app.MapPost("/search/metadata", async ( … ) ⇒
{
    var filter = filters.CreateGrpcFilter(req.Tags);

    var scroll = await qdrant.ScrollAsync(
        collectionName, filter: filter,
        limit: (uint)req.Limit,
        payloadSelector: true, cancellationToken: ct);

    var results = scroll.Result.Select(p ⇒ new SearchHit(
        Id: p.Id?.Uuid ?? p.Id?.Num.ToString(),
        Score: 0f,
        Payload: p.Payload.ToDictionary()));

    return Results.Ok(results);
});
```

- **No vectors** — uses Qdrant's `ScrollAsync`
- Tag-only browse/export
- Score is always `0f` (no similarity computed)

Module 2 — Try It (Filtered Search)

Filtered top-K

```
{ "queryText": "energy", "k": 5, "tags": { "category": "biology" } }
```

Physics is excluded even if "energy" is relevant to it.

Threshold search

```
{ "queryText": "biological processes", "scoreThreshold": 0.4 }
```

Try `0.8` (almost nothing passes) vs `0.2` (everything passes).

Metadata browse

```
{ "tags": { "category": "biology" } }
```

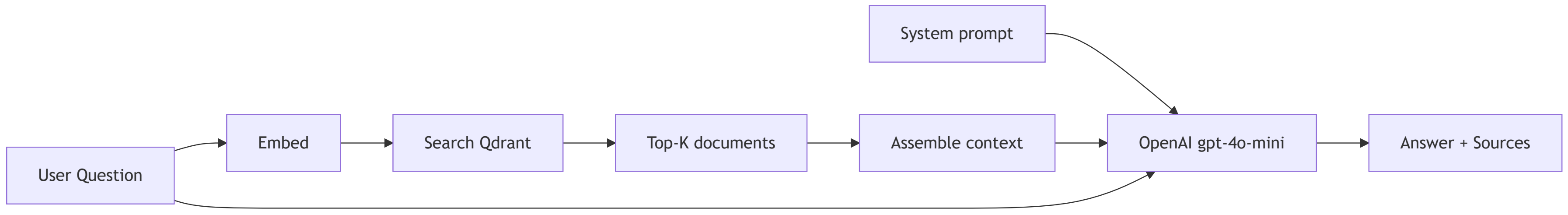
All biology documents, no vector search involved.

Module 3

Generation

~40 min · The core of the workshop

Module 3 — The RAG Pipeline



Steps

1. **Embed** the question → 1536-float vector
2. **Search** Qdrant for the K most similar documents
3. **Assemble** retrieved documents into numbered context
4. **Send** system prompt + context + question to the LLM
5. **Return** the answer + source documents with scores

Module 3 — System Prompt

```
private const string DefaultSystemPrompt =
    """
    You are a helpful assistant. Answer the user's question based **only** on
    the provided context documents. If the context does not contain enough
    information to answer, say so clearly — do not make up facts.
    """;
```

What this does

- **Grounds** the LLM in your documents
- **Prevents hallucination** — if the context doesn't have the answer, it says so
- Hard-coded default, customizable per request via `systemPrompt`

> Try asking "What is the best pizza recipe?" after indexing only science articles — the LLM will refuse to make up an answer.

Module 3 — ChatEndpoints.cs (1/2)

Embed → Search → Assemble context:

```
// 1. Embed the question
var vector = await embeddings.EmbedAsync(req.Question, ct);

// 2. Search Qdrant
var hits = await qdrant.SearchAsync(
    collectionName, vector, limit: (ulong)req.K,
    payloadSelector: true, cancellationToken: ct);

// 3. Assemble numbered context
List<ChatSource> sources = [];
List<string> contextParts = [];
for (var i = 0; i < hits.Count; i++)
{
    var text = hits[i].Payload.TryGetValue(Text, out var v) ? v.StringValue : "";
    sources.Add(new ChatSource(hits[i].Id?.Uuid ?? "?", hits[i].Score, text));
    contextParts.Add($"[{i + 1}] {text}");
}
```

Module 3 — ChatEndpoints.cs (2/2)

Send context + question to the LLM:

```
// 4. Send to LLM
List<ChatMessage> messages =
[
    new ChatMessage(ChatRole.System, DefaultSystemPrompt),
    new ChatMessage(ChatRole.User, $"""
        Context:
        {string.Join("\n\n", contextParts)}

        Question: {req.Question}
        """)
];

var response = await chatClient.GetResponseAsync(messages, cancellationToken: ct);
return Results.Ok(new ChatResponse(response.Text, sources));
```

- **System prompt** grounds the LLM — "only use provided context"
- **Context** = numbered documents from step 3
- Response includes both the **answer** and the **sources** list

Module 3 — Response Shape

```csharp
public record ChatRequest(
    string Question, int K = 5,
    float? ScoreThreshold = null,
    Dictionary<string, string>? Tags = null,
    string? SystemPrompt = null);

public record ChatResponse(
    string Answer,
    IReadOnlyList<ChatSource> Sources);

public record ChatSource(
    string Id, float Score, string TextSnippet);
```

Example response

```json
{
  "answer": "Plants produce energy from sunlight through photosynthesis...",
  "sources": [
    { "id": "6b64...", "score": 0.63, "textSnippet": "Photosynthesis is..." },
    { "id": "4524...", "score": 0.18, "textSnippet": "DNA replication..." }
  ]
}
```

Module 3 — Try It

Index documents, then chat (`POST /chat`)

```json
{ "question": "How do plants produce energy from sunlight?" }
```

The answer is grounded in the photosynthesis document.

Hallucination test

```json
{ "question": "What is the best pizza recipe?" }
```

Response: *"The provided context does not contain information about pizza recipes."*

Cross-domain

```json
{ "question": "Compare biological replication with quantum physics" }
```

The LLM pulls from both biology and physics documents.

Module 3 — Three New Controls

```
public record ChatRequest(
    string Question,
    int K = 5,
    float? ScoreThreshold = null,            // NEW
    Dictionary<string, string>? Tags = null, // NEW
    string? SystemPrompt = null              // NEW
);
```

Control	What it does
`ScoreThreshold`	Exclude low-relevance documents from context
`Tags`	Filter which documents are retrieved as context
`SystemPrompt`	Override the LLM’s persona per request

All optional — existing callers continue to work unchanged.

Module 3 — Custom System Prompts

```
var systemPrompt = req.SystemPrompt ?? DefaultSystemPrompt;

List<ChatMessage> messages =
[
    new ChatMessage(ChatRole.System, systemPrompt),
    new ChatMessage(ChatRole.User, $"""
        Context:{context}
        Question: {req.Question}
        """)
];
```

Same question, different personas

System prompt	Answer style
(default)	Neutral, factual
"You are a children's science teacher..."	Simple words, fun analogies
"You are a pirate..."	Pirate language, still grounded
"Answer in haiku format"	Three-line poem

Module 3 — Filtered + Threshold Chat

```
// Tag-filtered retrieval
var filter = filters.CreateGrpcFilter(req.Tags);

var hits = await qdrant.SearchAsync(
    collectionName, vector, limit: (ulong)req.K,
    filter: filter,
    scoreThreshold: req.ScoreThreshold,
    payloadSelector: true, cancellationToken: ct);
```

Combine all controls

```
{
  "question": "How do plants get energy?",
  "k": 3,
  "scoreThreshold": 0.4,
  "tags": { "category": "biology" },
  "systemPrompt": "Answer in exactly one sentence."
}
```

Only biology documents with score ≥ 0.4, answered in one sentence.

Module 3 — Exercises

Exercise 3.6 — Persona switch

Same question, three different system prompts:

Prompt	Expected tone
"You are a formal academic..."	Scholarly, citations
"You are a pirate..."	Arrr, matey! (still grounded)
"Answer in haiku format"	5-7-5 syllable poem

Exercise 3.7 — Combine all controls

```
{ "question": "How do plants get energy?", "k": 3, "scoreThreshold": 0.4,
  "tags": { "category": "biology" }, "systemPrompt": "Answer in exactly one sentence." }
```

Module 4

Chunking

~30 min

Module 4 — Why Chunk?

Problem

- Embedding models have a **token limit** (`text-embedding-3-small` → 8,191 tokens)
- Long documents produce **lower-quality embeddings** (too much meaning in one vector)
- Search returns the **whole document**, even if only one section is relevant

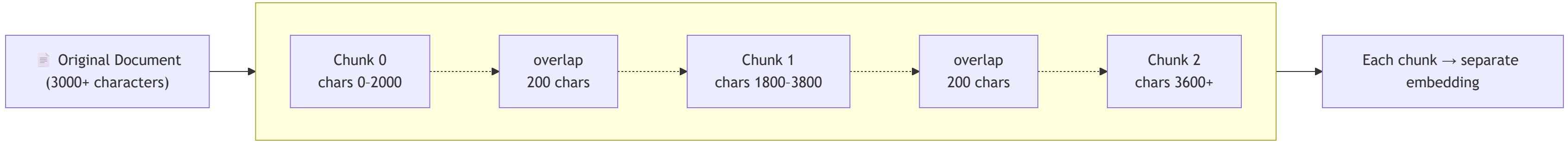
Solution: Chunking

Split long documents into smaller pieces, each gets its own vector.

Approach	Our implementation
Max chunk size	2,000 characters (~500 tokens)
Overlap	200 characters between adjacent chunks
Boundary detection	Prefer `\n` , `.` , `?` , `!` , whitespace

Result: Search returns the most relevant **section**, not the whole document.

Module 4 — Chunking Visual



Overlap ensures context at boundaries isn't lost — sentences near edges appear in multiple chunks.

Module 4 — TextChunker.cs

```csharp
public IReadOnlyList<TextChunk> Chunk(string text)
{
    if (text.Length <= options.MaxChunkSize)
        return [new TextChunk(text, Index: 0, StartOffset: 0, EndOffset: text.Length)];

    List<TextChunk> chunks = [];
    var chunkIndex = 0;
    var start = 0;

    while (start < text.Length)
    {
        var remaining = text.Length - start;
        var length = Math.Min(options.MaxChunkSize, remaining);

        // Not the last chunk? Try to find a sentence boundary.
        if (start + length < text.Length)
            length = FindSentenceBoundary(text, start, length);

        var chunkText = text.Substring(start, length).Trim();
        if (chunkText.Length > 0)
        {
            chunks.Add(new TextChunk(chunkText, chunkIndex, start, start + length));
            chunkIndex++;
        }
```

Module 4 — FindSentenceBoundary

```csharp
private static int FindSentenceBoundary(string text, int start, int maxLength)
{
    var searchStart = start + maxLength / 2;

    // Prefer paragraph breaks
    var newlinePos = text.LastIndexOf('\n', start + maxLength - 1,
        maxLength - (searchStart - start));
    if (newlinePos > searchStart)
        return newlinePos - start + 1;

    // Then sentence enders (. ? !)
    for (var i = start + maxLength - 1; i >= searchStart; i--)
    {
        if (Array.IndexOf(SentenceEnders, text[i]) >= 0
            && i + 1 < text.Length && char.IsWhiteSpace(text[i + 1]))
            return i - start + 1;
    }

    // Then any whitespace
    var spacePos = text.LastIndexOf(' ', start + maxLength - 1,
        maxLength - (searchStart - start));
    if (spacePos > searchStart) return spacePos - start;
```

Module 4 — Chunked DocumentIndexer

```csharp
var chunks = chunker.Chunk(request.Text);

for (var i = 0; i < chunks.Count; i++)
{
    // Single-chunk: keep original id; multi-chunk: derive per-chunk id
    var pointIdStr = chunks.Count == 1
        ? sourceId
        : $"{sourceId}_chunk_{i}".ToDeterministicGuid().ToString("D");

    var vector = await embeddings.EmbedAsync(chunk.Text, ct);

    // Multi-chunk metadata
    if (chunks.Count > 1)
    {
        point.Payload[SourceDocId]  = sourceId;
        point.Payload[ChunkIndex]   = i.ToString();
        point.Payload[TotalChunks]  = chunks.Count.ToString();
    }

    // Tags & properties copied to EVERY chunk
    if (request.Tags is not null)
        foreach (var (key, value) in request.Tags)
            point.Payload[$"{TagPrefix}{key}"] = value;
}
```

Module 4 — PayloadKeys Update

```
public static class PayloadKeys
{
    public const string Text           = "text";
    public const string IndexedAtMs    = "indexed_at_ms";
    public const string TagPrefix      = "tag_";
    public const string PropertyPrefix = "prop_";
    public const string SourceDocId    = "source_doc_id";   // NEW
    public const string ChunkIndex     = "chunk_index";     // NEW
    public const string TotalChunks    = "total_chunks";    // NEW
}
```

A chunked point's payload

```
{
  "text": "...chunk text...",
  "indexed_at_ms": 1718500000000,
  "source_doc_id": "b61e52cb-d639-1056-874c-0b77556478f5",
  "chunk_index": "1",
  "total_chunks": "4",
  "tag_category": "history"
}
```

Group results by `source_doc_id` to reconstruct the original document.

Module 4 — Models

```
public sealed class ChunkingOptions
{
    public int MaxChunkSize { get; set; } = 2000;
    public int Overlap { get; set; } = 200;
}

public record TextChunk(
    string Text,
    int Index,
    int StartOffset,
    int EndOffset
);
```

Configurable via environment variables

```
CHUNKING_MAX_SIZE=200 CHUNKING_OVERLAP=50 dotnet run
```

Response for a chunked document

```
{ "pointId": "b61e ... ", "totalChunks": 4, "chunkPointIds": ["b61e ... ", "b931 ... ", "d454 ... ", "5ab7 ... "] }
```

Module 4 — Try It

Index a long document (3000+ chars)

Use the **coffee history article** from the module README. It produces **4 chunks.**

Search for specific sections

```
{ "queryText": "How did coffee spread from Africa to Europe?", "k": 5 }
```

Different chunks from the same document match with different scores. The chunk about European arrival ranks highest.

Chat with chunked documents

```
{ "question": "Compare coffeehouses in the Middle East vs England" }
```

The RAG pipeline retrieves chunks from different parts of the same document, assembling cross-section context.

Module 5

User Interface

~20 min · Bonus module

Module 5 — Static File Middleware

Three lines in `Program.cs`:

```
app.UseDefaultFiles();   // maps "/" → "/index.html"
app.UseStaticFiles();    // serves everything in wwwroot/
```

Root URL now serves the frontend (`index.html`). The `GET /api/info` and `GET /health` endpoints use the same `MapInfoEndpoints` extension method as every other module.

Single-file frontend

- `wwwroot/index.html` — no build step, no npm, no bundler
- Pico.css — classless CSS from CDN (~10 KB)
- **Vanilla JavaScript** — plain `fetch()` calls to the API

Module 5 — Four Tabs

Tab	Endpoints	Features
Chat	`POST /chat`	Conversation UI, expandable sources, score bars
Search	`POST /search/topk`, `/threshold`, `/metadata`	Three modes, tag widgets, threshold slider
Documents	`POST /documents`, `/documents/batch`	Single + batch indexing, chunk count feedback
Status	`GET /api/info`, `GET /health`	Config grid, auto-refreshing health indicator

Reusable Components

- **Tag chip widget** — key + value input → dismissible pills (used in 6 forms)
- **Score color function** — green ≥0.7, gold ≥0.4, red below
- **Theme selector** — Auto / Light / Dark with localStorage persistence
- **Auto-refresh** — Health indicator updates every 15 seconds

Module 5 — Try It

1. Start & open

```
cd module-05
docker compose up -d
dotnet run --project src/Qdrant.Demo.Api
```

Visit http://localhost:8080/ — you see the Chat tab.

2. Status tab — check health and config
3. Documents tab — index single + batch documents
4. Search tab — try all three modes
5. Chat tab — ask questions, inspect sources

> Swagger UI is still available at `/swagger` if you need it.

Module 5 — Exercises

Exercise 5.1 — Theme switching

The nav bar has a theme selector: **Auto / Light / Dark.** Try switching.

Persisted to `localStorage` — survives page reloads.

Exercise 5.2 — Custom system prompt

In the Chat tab → Advanced settings:

> *"You are a pirate. Answer in pirate language, but still base your answers on the provided context documents."*

Tone changes, facts stay grounded.

🎉 Workshop Complete!

You’ve built a full RAG solution from scratch.

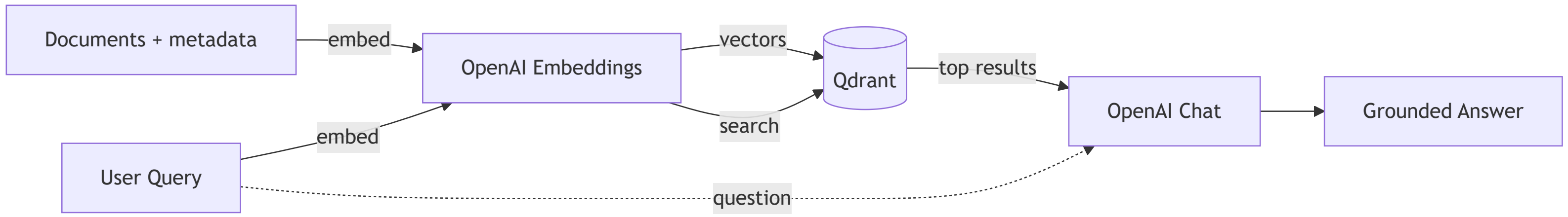
What You Built

- **Module 0**: Setup: Qdrant connection, Swagger, health check
- **Module 1**: Document indexing with embeddings and batch operations
- **Module 2**: Retrieval: similarity search, metadata, filtering
- **Module 3**: Generation: RAG pipeline, custom prompts, filtered chat
- **Module 4**: Text chunking with sentence-boundary awareness
- **Module 5**: User Interface: static frontend for every endpoint

Full API Reference

Endpoint	Method	Description	Module
`/api/info`	GET	Service info	0
`/health`	GET	Health check	0
`/documents`	POST	Index a single document	1
`/documents/batch`	POST	Batch document indexing	1
`/search/topk`	POST	Top-K similarity search	2
`/search/threshold`	POST	Threshold similarity search	2
`/search/metadata`	POST	Metadata-only search	2
`/chat`	POST	Full RAG pipeline	3

Architecture Recap



Component	Technology
Vector database	Qdrant (Docker)
API framework	.NET 10 Minimal API
Embeddings	OpenAI `text-embedding-3-small` (1536 dim)
Chat	OpenAI `gpt-4o-mini`
Frontend	Vanilla JS + Pico.css

Key Takeaways

1. **Embeddings** capture meaning, not keywords

Semantic search finds relevant documents even when words don’t match.

2. **RAG grounds the LLM** in your data

No hallucination — the model answers from indexed documents.

3. **Chunking** makes long documents searchable

Each section gets its own vector → more focused retrieval.

4. **Metadata filtering** narrows the search space

Pre-filtering by tags before computing similarity is very efficient.

5. **The RAG pattern is composable**

System prompts, filters, thresholds, chunking — mix and match.

What to Explore Next

Extension	Description
Streaming chat	`IChatClient.GetStreamingResponseAsync` for real-time token delivery
Multi-turn conversation	Add `History` field to `ChatRequest` for follow-up questions
Token-aware chunking	`Microsoft.ML.Tokenizers` for exact token counts
Different vector DB	Swap Qdrant for Pinecone, Weaviate, Milvus, or ChromaDB

Glossary

Term	Definition
Embedding	Fixed-length float array representing semantic meaning of text
Vector	A list of numbers (1536 floats in our case)
Cosine similarity	Measure of vector similarity: 0 (unrelated) to 1 (identical)
Collection	Qdrant container for vectors of the same dimensionality
Point	Entry in a collection: id + vector + payload
Payload	Key/value metadata stored alongside a vector
Upsert	Insert-or-update: same id → overwrite
Grounding	Providing factual context to an LLM
Hallucination	LLM generating plausible but incorrect information
Chunk	Smaller segment of a long document

Qdrant.Client SDK Tips

Common gotchas

Issue	Solution
`Range` type ambiguity	Fully qualify: `Qdrant.Client.Grpc.Range`
`SearchAsync` parameters	Use `payloadSelector: true` (not `withPayload`)
`limit` type	Cast to `ulong` : `(ulong)req.K`
`Value` type	Use `Qdrant.Client.Grpc.Value` (not `Google.Protobuf`)
No `NumberValue`	Use `DoubleValue` and `IntegerValue` separately
`filter` parameter	Expects `Filter?` , not `Condition?`

> These trip up everyone at first — keep this slide handy when writing Qdrant code!

Thank you!

📦 Repository: `github.com/PeterMilovcik/Qdrant.Demo`

💼 LinkedIn: `linkedin.com/in/peter-milovcik`