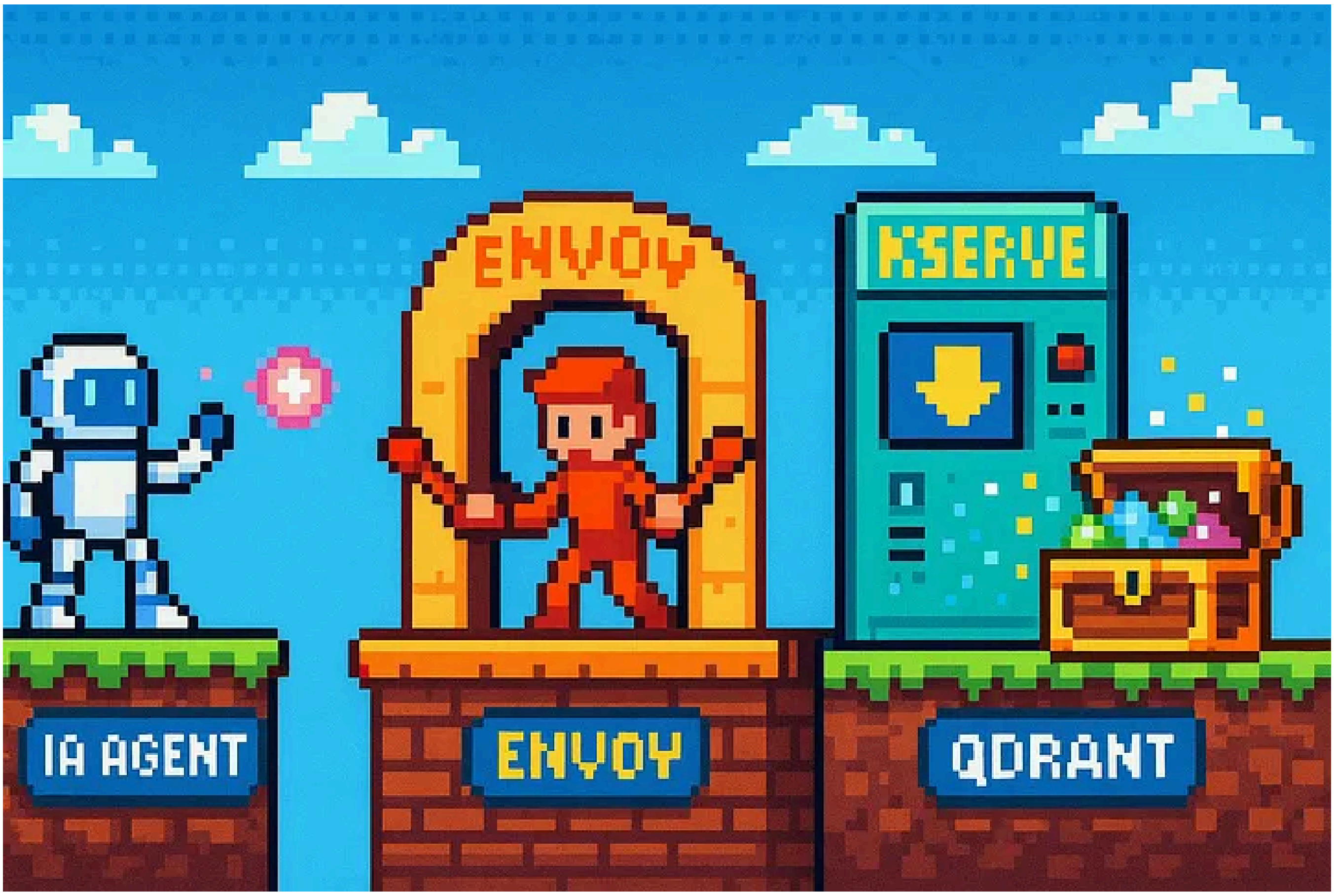
Cloud-Native RAG: embeddings + vectors, fully self-hosted

15 min read · 1 day ago

using System; Follow



> *An agent without a pre-trained model needs your data. You inject your documents into a vector database the agent can query to answer questions or drive actions for your specific needs.*

We will see in this new article **how to host your RAG solution (embeddings + vector DB) fully self-hosted** in an **industrial** way. Then we'll actually **use it** to implement RAGs and connect your agents. Examples are in **.NET**. Sorry to the Python lovers — Python doesn't have a monopoly on AI 🙂

The principle is basic: **to create an embedding, one POST is enough**, and integration with **QDrant** is straightforward via SDKs across stacks (Python, .NET, Java, Rust…). For agents (LangChain, Semantic Kernel…), **all integrate natively with QDrant.**

As usual, deployment scripts are done with **Terraform** to deploy Kubernetes resources.

We'll use:

- **KServe** to host our embedding inference model.
- **Envoy AI Gateway** to securely and uniformly expose the inference model.
- **QDrant** for the vector database.
- **.NET 10** to implement a rag tool to index documments and mcp tool for the agent.

All of it deployed on a **Kubernetes** cluster.

High-Level architecture diagram

```
+----------------------------+
|        Your Agent          |
| (LangChain / SK / etc.)    |
+------------+---------------+
             |  OpenAI schema (HTTP / JSON)
             v
+----------------------------+
|     Envoy AI Gateway       |
|  - Auth, rate limit        |
|  - Observability           |
|  - Unified routing         |
+------------+---------------+
             |  Cluster-internal HTTP
             v
+----------------------------+
|          KServe            |
|  InferenceService bge-m3   |
```

```
+------------+--------------+
             |  pulls model (HF) at init
             v
+---------------------------+
|     bge-m3 Predictor      |
|         (CPU)             |
+------------+--------------+
             | embeddings (float[])
             v
+---------------------------+
|          Qdrant           |
|   Vector search + payload |
+---------------------------+
```

Why self-host?

Starting in **SaaS** for embeddings (OpenAI, Anthropic, Mistral…) is the right way. GPU instances (even though embeddings can run on CPU) have a steep fixed cost and there's infrastructure to maintain.

But when **costs explode** or when you have **sovereignty constraints** for your data, **self-hosted** becomes the way to go.

Kubernetes and the **open-source** community make cloud-native AI more accessible than ever. That's exactly what we'll build together here.

Step 1 — Install KServe CRDs

Prereqs: Kubernetes > **1.30**, **cert-manager** (> **1.15.0**) on your cluster.

> **Why install CRDs separately?**
> Always preferable to install CRDs in a **common infrastructure layer**, and install implementations in separate terraform layers.

```
resource "helm_release" "kserve" {

  name             = "kserve-crd"
  namespace        = "kube-system"
  create_namespace = false
  chart            = "kserve-crd"
  repository       = "oci://ghcr.io/kserve/charts"
  version          = "v0.16.0" # Verion of 2025 october
}
```

CRDs deployed

```
usingsystem@% kubectl get crd | grep kserve
clusterservingruntimes.serving.kserve.io                    2025-07-24T18:21:5
clusterstoragecontainers.serving.kserve.io                  2025-07-24T18:21:5
inferencegraphs.serving.kserve.io                           2025-07-24T18:21:5
inferenceservices.serving.kserve.io                         2025-07-24T18:21:5
localmodelcaches.serving.kserve.io                          2025-07-24T18:21:5
localmodelnodegroups.serving.kserve.io                      2025-07-24T18:21:5
localmodelnodes.serving.kserve.io                           2025-07-24T18:21:5
servingruntimes.serving.kserve.io                           2025-07-24T18:21:5
trainedmodels.serving.kserve.io                             2025-07-24T18:21:5
```

What they do :

- **InferenceService** — classic KServe resource for deploying a predictive/generative model behind a stable URL. Defines predictor (runtime, model, resources), ingress, and autoscaling.
- **ClusterStorageContainer** — cluster-level definition of model sources (OCI, model kits, HF, S3) and the init-container that downloads/extracts models. Other CRDs reference it via storage config.
- **LocalModelNodeGroup** — labels a set of nodes where KServe may pre-cache models locally; sets disk quotas for cache.
- **LocalModelNode** — per-node state of the local cache (downloaded/failed/not ready). Controlled by KServe for fine-grained troubleshooting.
- **LocalModelCache** — declares that a given model must be **pre-downloaded** to specific node groups (reducing cold-start latency).

Step 2 — Install KServe (Standard mode)

```
resource "kubernetes_namespace" "kserve" {
  metadata {
    name = "kserve"

    labels = {
      provisioned_by  = "terraform"
    }
```

```
  }
}

resource "helm_release" "kserve" {

  name       = "kserve"
  namespace  = kubernetes_namespace.kserve.metadata[0].name
  chart      = "kserve"
  repository = "oci://ghcr.io/kserve/charts"
  version    = "v0.16.0" # Version of 2025 october

  values = [
    yamlencode({
      kserve = {
        controller = {
           = "Standard"
        }
        storage = {
          resources = {
            requests = {
              cpu    = "100m"
              memory = "2Gi"
            }
            limits = {
              cpu    = "1"
              memory = "8Gi"
            }
          }
        }
      }
    })
  ]
}
```

> **Standard vs Knative**
> **Standard** → KServe creates vanilla Kubernetes resources (Deployment, Service, HPA/KEDA, Ingress). Recommended for prod and LLMs if you don't need scale-to-zero.
> **Knative** → Serverless (scale-to-zero, revisions, canary). Powerful, but requires Knative (+ data plane like Istio) and currently **doesn't work natively with Envoy AI Gateway** in our path.

Storage resources matter
`storage` sets resources for the **init-container** that downloads models. Defaults are often too low for multi-GB models → pod crashes. You can override per backend (HF/S3/etc.).

All chart values here.

Validate KServe install :

```
usingsystem@~ % kubectl get all -n kserve
NAME                                                  READY   STATUS    R
pod/kserve-controller-manager-65b6cc8469-szx2w        2/2     Running   0
pod/kserve-localmodel-controller-manager-5b4778d898-l5562   1/1     Running   0

NAME                                       TYPE        CLUSTER-IP       EXTER
service/kserve-controller-manager-service   ClusterIP   XXX.XXX.XX.XX    <none
service/kserve-webhook-server-service       ClusterIP   XXX.XXX.XXX.XXX  <none
```

> Classic CRD/controller pattern: KServe's controller watches your KServe manifests and creates the needed Kubernetes resources.

Step 3 — Choose the embedding model

We'll use **bge-m3**. It's a solid CPU-mode starting point: **multilingual**, versatile, easy to deploy.

If your top priority is absolute top scores and you can allocate a GPU, **gte-Qwen2–7B-instruct** or **jina-v3** can do better.

We'll proceed with **bge-m3**, downloaded from **Hugging Face**:

≈ **3 GB**. If you only need English, consider **bge-large-en-v1.5**, **bge-base-en-v1.5**, or smaller **bge-small-en-v1.5**.

Family table :

```
Model Name                  Dimension    Sequence Length    Introduction
BAAI/bge-m3                 1024         8192               multilingual; unified
BAAI/bge-large-en-v1.5      1024         512                English model
BAAI/bge-base-en-v1.5       768          512                English model
BAAI/bge-small-en-v1.5      384          512                English model
```

Step 4 — Host bge-m3 on KServe and expose REST

```
resource "kubectl_manifest" "inference_service" {

  yaml_body = yamlencode({
    apiVersion = "serving.kserve.io/v1beta1"
    kind       = "InferenceService"
    metadata = {
      name      = "bge-m3"
      namespace = "using-system-models"
      annotations = {
        "serving.kserve.io/autoscalerClass" = "hpa"
      }
    }
    spec = {
      predictor = {
        minReplicas = 2
        maxReplicas = 3
        scaleTarget = 80
        scaleMetric = "cpu"

        model = {
          modelFormat = {
            name = "huggingface"
          }
          args = [
            "--model_name=bge-m3",
            "--backend=huggingface",
            "--task=text_embedding",
          ]
          storageUri = "hf://BAAI/bge-m3"

          resources = {
            requests = {
              cpu    = "2"
              memory = "12Gi"
            }
            limits = {
              cpu    = "4"
              memory = "24Gi"
            }
          }
        }

      }
    }
  })
}
```

> We host fully on **CPU** (no GPU). Add `tolerations` and `nodeSelector` under `predictor` to land on the right nodes.

Autoscaling :

```
"serving.kserve.io/autoscalerClass" = "hpa"

minReplicas = 2
maxReplicas = 3
scaleTarget = 80
scaleMetric = "cpu"
```

This scales 2→3 replicas with an 80% CPU trigger. **HPA** is simple; for metric-based scaling (e.g., workflow requests) use **KEDA**. For scale-to-zero, use **Knative**.

Model args :

```
modelFormat = { name = "huggingface" }
args = [
  "--model_name=bge-m3",
  "--backend=huggingface",
  "--task=text_embedding",
]
storageUri = "hf://BAAI/bge-m3"
```

- Use HF backend, set task `text_embedding`, download from `hf://BAAI/bge-m3`.

Kubernetes resources created :

```
usingsystem@~ % kubectl get all -n using-system-models
NAME                                       READY   STATUS    RESTARTS   AGE
pod/bge-m3-predictor-6d6f6c49c7-7skmt     2/2     Running   0          84m
pod/bge-m3-predictor-6d6f6c49c7-z2v9z     2/2     Running   0          84m

NAME                       TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)
service/bge-m3-predictor   ClusterIP   XXX.XXX.XXX.XX   <none>        80/TCP

NAME                                READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/bge-m3-predictor    2/2     2            2           3d18h
```

```
NAME                                                    REFERENCE
horizontalpodautoscaler.autoscaling/bge-m3-predictor    Deployment/bge-m3-predic
```

Test with curl :

```
curl -X POST "http://bge-m3-predictor.using-system-models.svc.cluster.local/ope
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "Accept-Encoding: identity" \
  -d '{
    "model": "bge-m3",
    "input": "This is just a basic embedding test."
  }'
```

Response :


```
{
  "id": "0ebbb54d-802b-4af1-8738-f28f770e194f",
  "object": "embedding",
  "created": 1761117146,
  "model": "bge-m3",
  "data": [
    {
      "index": 0,
      "object": "embedding",
      "embedding": [
        -0.01863192766904831,
        0.0039021808188408613,
        -0.011211015284061432,
        -0.00893788319081068,
        0.009318851865828037,
        ...
        0.020269472151994705
      ]
    }
  ],
  "usage": {
    "prompt_tokens": 12,
    "total_tokens": 12,
    "completion_tokens": 0,
    "prompt_tokens_details": null
```

```
    }
  }
```

> Pod init can be slow (model download). We'll cover **KServe Local Model Cache** in bonus to slash cold-start times.

Step 5 — Add Envoy AI Gateway (security, routing, observability)

Not mandatory for simple needs, but crucial for **auth**, **rate limiting**, **observability**, **fallbacks**, **federating** models.

- Doc: https://aigateway.envoyproxy.io/
- Install guide : https://usingsystem.io/install-an-ai-gateway-with-envoy-gateway-ai-7988ac28f901
- OpenAI backend example : https://usingsystem.io/openai-backend-with-envoy-ai-gateway-3cc4c438effb

We'll expose **KServe bge-m3** almost like a SaaS provider.

We need **AIServiceBackend**, **ReferenceGrant**, and **AIGatewayRoute**.

```
variable "name" {
  description = "The name of the model to be deployed."
  type        = string
  defaut      = "bge-m3"
}

variable "namespace" {
  description = "The namespace in which to deploy the model."
  type        = string
}

variable "ai_gateway_namespace" {
  description = "The namespace for the AI Gateway."
  type        = string
}

variable "ai_gateway_name" {
  description = "The name of the AI Gateway."
  type        = string
}
```

```
locals {
  ai_backend_name         = "${var.name}-predictor"
  ai_backend_service_name = "ai-backend-service-${var.name}"
}

resource "kubectl_manifest" "ai_service_backend" {

  depends_on = [
    kubectl_manifest.inference_service
  ]

  yaml_body = yamlencode({
    apiVersion = "aigateway.envoyproxy.io/v1alpha1"
    kind       = "AIServiceBackend"
    metadata = {
      name      = local.ai_backend_service_name
      namespace = var.ai_gateway_namespace
    }
    spec = {
      schema = {
        name = "OpenAI"
      }
      backendRef = {
        name      = local.ai_backend_name
        namespace = var.namespace
        kind      = "Service"
        group     = ""
        port      = 80
      }
      timeouts = {
        request = "300s"
      }
    }
  })
}

resource "kubectl_manifest" "ai_service_backend_grant" {

  depends_on = [
    kubectl_manifest.ai_service_backend
  ]

  yaml_body = yamlencode({
    apiVersion = "gateway.networking.k8s.io/v1beta1"
    kind       = "ReferenceGrant"
    metadata = {
      name      = "envoy-gateway-ref-grant-${var.name}"
      namespace = var.namespace
    }
    spec = {
      from = [
        {
          group     = "gateway.networking.k8s.io"
```

```
          kind      = "HTTPRoute"
          namespace = var.ai_gateway_namespace
        }
      ]
      to = [
        {
          group = ""
          kind  = "Service"
        }
      ]
    }
  })
}

resource "kubectl_manifest" "ai_backend_route" {

  depends_on = [
    kubectl_manifest.inference_service,
    kubectl_manifest.ai_service_backend,
    kubectl_manifest.ai_service_backend_grant
  ]

  yaml_body = yamlencode({
    apiVersion = "aigateway.envoyproxy.io/v1alpha1"
    kind       = "AIGatewayRoute"
    metadata = {
      name      = "envoy-ai-gateway-${var.name}-route"
      namespace = var.ai_gateway_namespace
    }
    spec = {
      schema = {
        name = "OpenAI"
      }
      parentRefs = [
        {
          name      = var.ai_gateway_name
          namespace = var.ai_gateway_namespace
          kind      = "Gateway"
          group     = "gateway.networking.k8s.io"
        }
      ]
      rules = [
        {
          matches = [
            {
              headers = [
                {
                  type  = "Exact"
                  name  = "x-ai-eg-model"
                  value = var.name
                }
              ]
            }
```

```
          ]
          backendRefs = [
            {
              name   = local.ai_backend_service_name
              weight = 100
            }
          ]
        }
      ]
      llmRequestCosts = [
        {
          metadataKey = "llm_input_token"
          type        = "InputToken"
        },
        {
          metadataKey = "llm_output_token"
          type        = "OutputToken"
        },
        {
          metadataKey = "llm_total_token"
          type        = "TotalToken"
        }
      ]
    }
  })
}
```

What this does

- **AIServiceBackend** — declares the backend with **OpenAI** schema and points to `Service/bge-m3-predictor:80`.
- **ReferenceGrant** — authorizes a cross-namespace **HTTPRoute** to reference that **Service**.
- **AIGatewayRoute** — attaches to your **Gateway**, matches header `x-ai-eg-model: bge-m3`, routes to the backend, and records token usage via `llmRequestCosts`.

```
[ Client ] --OpenAI /v1/embeddings--> [ Envoy AI Gateway ]
   |    header: x-ai-eg-model: bge-m3          |
   |-------------------------------------------v
                           [ AIGatewayRoute (OpenAI schema) ]
                                        |
                                        v
                              [ AIServiceBackend ]
                                        |
```

```
                       v
             [ Service/bge-m3-predictor:80 ]
```

KServe URL prefix tweak (important for Envoy AI Gateway integration) :

```
resource "kubectl_manifest" "inference_service" {

  yaml_body = yamlencode({
    apiVersion = "serving.kserve.io/v1beta1"
    kind       = "InferenceService"
    metadata = {
      name      = "bge-m3"
      namespace = "using-system-models"
      annotations = {
        "serving.kserve.io/autoscalerClass" = "hpa"
      }
    }
    spec = {
      predictor = {
        ....
        env = [
          {
            name  = "KSERVE_OPENAI_ROUTE_PREFIX",
            value = ""
          }
        ]

      }
    }
  })
}
```

Discover models via Envoy :

```
curl -X GET "http://envoy-ai-gateway.envoy-gateway-system.svc.cluster.local/mod
  -H "Content-Type: application/json" \
  -H "Accept: application/json"
```

```
{
  "object": "list",
```

```
  "data": [
    {
      "id": "gpt-4",
      "object": "model",
      "created": 1687882411,
      "owned_by": "Envoy AI Gateway"
    },
    {
      "id": "bge-m3",
      "object": "model",
      "created": 1687882411,
      "owned_by": "Envoy AI Gateway"
    },
    ...
  ]
}
```

Query embeddings via Envoy :

```
curl -X POST "http://envoy-ai-gateway.envoy-gateway-system.svc.cluster.local/v1
  -H "Content-Type: application/json" \
  -H "Accept: application/json" \
  -H "Accept-Encoding: identity" \
  -d '{
    "model": "bge-m3",
    "input": "This is just a basic embedding test."
  }'
```

Step 6 — Install QDrant (vector DB)

Qdrant is a Rust vector DB for high-dimensional similarity. In cluster mode, it shards and replicates for HA. The Helm chart enables distributed mode by bumping `replicaCount` and setting `config.cluster.enabled: true`. Production hardening: storage, network, anti-affinity, observability, upgrade strategies.

Terraform :

```
resource "kubernetes_namespace" "qdrant" {
  metadata {
    name = "qdrant"

    labels = {
```

```
      provisioned_by  = "terraform"
    }
  }
}

resource "helm_release" "qdrant" {

  name       = "qdrant"
  namespace  = kubernetes_namespace.qdrant.metadata[0].name
  chart      = "qdrant"
  repository = "https://qdrant.github.io/qdrant-helm"
  version    = "1.15.5" // Version en date d'octobre 2025

  values = [
    yamlencode({

      replicaCount = 3

      resources = {
        requests = {
          cpu    = "2"
          memory = "12Gi"
        }
        limits = {
          cpu    = "4"
          memory = "24Gi"
        }
      }

      persistence = {
        enabled = true
        size    = "30Gi"
      }

      podDisruptionBudget = {
        enabled        = true
        maxUnavailable = 1
      }

      affinity = {
        podAntiAffinity = {
          preferredDuringSchedulingIgnoredDuringExecution = [{
            weight = 100
            podAffinityTerm = {
              topologyKey = "kubernetes.io/hostname"
              labelSelector = {
                matchLabels = { "app.kubernetes.io/name" = "qdrant" }
              }
            }
          }]
        }
      }
```

```
        config = {
          cluster = {
            enabled = true
          }
        }
      })
    ]
  }
```

- **PodDisruptionBudget (PDB) — `maxUnavailable = 1`:**
  Limits **voluntary** disruptions (node drains, upgrades, cluster autoscaler evictions) so that at most **one** Qdrant pod can be down at a time. With `replicaCount >= 2`, Kubernetes will serialize evictions to keep the service available. This is particularly helpful during rolling upgrades of nodes or when draining for maintenance.

- **Pod anti-affinity (preferred) across hosts:**
  `preferredDuringSchedulingIgnoredDuringExecution` with `topologyKey = "kubernetes.io/hostname"` nudges the scheduler to **spread Qdrant pods across different nodes**. This reduces correlated failure risk (one node loss shouldn't take out all replicas).
  Using **preferred** (instead of **required**) keeps scheduling flexible when capacity is tight. If you need strict separation, you can switch to `requiredDuringSchedulingIgnoredDuringExecution`—but be aware it may block scheduling in small clusters.

Services :

```
usingsystem@~ % kubectl get svc -n qdrant -o wide
NAME              TYPE        CLUSTER-IP      EXTERNAL-IP   PORT(S)
qdrant            ClusterIP   XXX.XXX.XX.XXX  <none>        6333/TCP,6334/TCP,
qdrant-headless   ClusterIP   None            <none>        6333/TCP,6334/TCP,
```

> **CPU vs GPU**
> For **query**, GPU won't change much. Gains are mainly for **heavy indexing** with large embeddings (bge-m3 is 1024-D), where GPU can bring notable speedups (often ~10×).

Step 7 — Development: .NET end-to-end

To index your documents, create an embedding and send vectors into QDrant.

Create embedding with a single POST :

```
var embeddingApiUrl = "http://envoy-ai-gateway.envoy-gateway-system.svc.cluster
var httpClient = new HttpClient();

async Task<List<float>> CreateEmbeddingAsync(string text)
{
    // Build JSON manually to avoid warnings
    var escapedText = text.Replace("\\", "\\\\").Replace("\"", "\\\"");
    var requestBody = $"{{\"model\":\"bge-m3\",\"input\":\"{escapedText}\"}}";

    var content = new StringContent(requestBody, System.Text.Encoding.UTF8, "ap
    var response = await httpClient.PostAsync(embeddingApiUrl, content);
    response.EnsureSuccessStatusCode();

    var responseBody = await response.Content.ReadAsStringAsync();
    using var embeddingDoc = JsonDocument.Parse(responseBody);

    // Extract embedding vector from response
    var embeddingArray = embeddingDoc.RootElement
        .GetProperty("data")[0]
        .GetProperty("embedding");

    var vectors = new List<float>();
    foreach (var value in embeddingArray.EnumerateArray())
    {
        vectors.Add((float)value.GetDouble());
    }

    return vectors;
}
```

Qdrant client (Nuget Qdrant.Client) :

```
var qdrantHost = "qdrant.qdrant.svc.cluster.local";
var qdrantClient = new QdrantClient(qdrantHost);
```

Create (or recreate) a collection :

```
async Task CreateOrRecreateCollectionAsync(string collectionName, uint vectorSi
{
    // Check if collection exists and delete if it does
    var collections = await qdrantClient.ListCollectionsAsync();
    if (collections.Contains(collectionName))
    {
        await qdrantClient.DeleteCollectionAsync(collectionName);
    }

    // Create collection
    await qdrantClient.CreateCollectionAsync(
        collectionName: collectionName,
        vectorsConfig: new VectorParams { Size = vectorSize, Distance = Distanc
    );
}
```

Insert an embedding :

```
await CreateOrRecreateCollectionAsync("myCollection", 1024);
...
string text = "...";
string language = "en-US";

var embedding = await CreateEmbeddingAsync(text);

                var point = new PointStruct
                {
                    Id = Guid.NewGuid(),
                    Vectors = embedding.ToArray(),
                    Payload =
                    {
                        ["text"] = text,
                        ["language"] = language ?? "en-us"
                        ...
                    }
                };

await qdrantClient.UpsertAsync("myCollection", new[] { point });
```

> When your agent queries the vector DB, it retrieves nearest vectors and the **payload** becomes the context. Structure the payload thoughtfully.

Agent integration

Two options :

- Use native Qdrant integrations in your agent framework if available.
- Or keep it simple via a **Tool** or a **Model Context Protocol (MCP)** server.

In this article we will choose the option 2.

Service building context from Qdrant :

```
using System.Text;
using System.Text.Json;
using Qdrant.Client;

namespace ms_doc_assistant.Services;

public class QdrantRagService
{
    private readonly QdrantClient _qdrantClient;
    private readonly HttpClient _embeddingHttpClient;
    private readonly string _embeddingApiUrl;
    private readonly string _embeddingModel;

    public QdrantRagService(
        string qdrantHost,
        string embeddingApiUrl,
        string embeddingModel)
    {
        _qdrantClient = new QdrantClient(qdrantHost);
        _embeddingHttpClient = new HttpClient();
        _embeddingApiUrl = embeddingApiUrl;
        _embeddingModel = embeddingModel;
    }

    /// <summary>
    /// Build context string from search results for the agent
    /// </summary>
    public async Task<string> BuildContextForQueryAsync(string query, int limit
    {
        var documents = await SearchDocumentsAsync(query, limit);

        if (documents.Count == 0)
        {
            return "No relevant documents found.";
        }

        var contextBuilder = new StringBuilder();
        contextBuilder.AppendLine("### Relevant Knowledge Base Articles:");
        contextBuilder.AppendLine();
```

```
        foreach (var (title,
                      category,
                      tags,
                      summary,
                      content,
                      score) in documents)
        {
            contextBuilder.AppendLine($"**{title}** ({category}) [relevance: {s

            if (tags.Length > 0)
            {
                contextBuilder.AppendLine($"Tags: {string.Join(", ", tags)}");
            }

            if (!string.IsNullOrWhiteSpace(summary))
            {
                contextBuilder.AppendLine($"Summary: {summary}");
            }

            if (content.Length > 0)
            {
                contextBuilder.AppendLine("Key Points:");
                foreach (var section in content)
                {
                    contextBuilder.AppendLine($"  - {section}");
                }
            }

            contextBuilder.AppendLine();
        }

        return contextBuilder.ToString();
    }

    private async Task<List<(string title,
            string category,
            string[] tags,
            string summary,
            string[] content,
            double score)>>
        SearchDocumentsAsync(
            string query,
            int limit = 5)
    {
        var queryEmbedding = await CreateEmbeddingAsync(query);

        var results = await _qdrantClient.SearchAsync(
            collectionName: "knowledge_base",
            vector: queryEmbedding.ToArray(),
            limit: (ulong)limit
        );
```

```
            return results.Select(r => (
                title: r.Payload["title"].StringValue,
                category: r.Payload["category"].StringValue,
                tags: r.Payload["tags"].ListValue.Values.Select(v => v.StringValue)
                summary: r.Payload["summary"].StringValue,
                content: r.Payload["content"].ListValue.Values.Select(v => v.String
                score: (double)r.Score
            )).ToList();
        }

        /// <summary>
        /// Create embedding for text using the embedding API
        /// </summary>
        private async Task<List<float>> CreateEmbeddingAsync(string text)
        {
            var escapedText = text.Replace("\\", "\\\\").Replace("\"", "\\\"");
            var requestBody = $"{{\"model\":\"{_embeddingModel}\",\"input\":\"{esca

            var content = new StringContent(requestBody, Encoding.UTF8, "applicatio
            var response = await _embeddingHttpClient.PostAsync(_embeddingApiUrl, c
            response.EnsureSuccessStatusCode();

            var responseBody = await response.Content.ReadAsStringAsync();
            using var embeddingDoc = JsonDocument.Parse(responseBody);

            var embeddingArray = embeddingDoc.RootElement
                .GetProperty("data")[0]
                .GetProperty("embedding");

            var vectors = new List<float>();
            foreach (var value in embeddingArray.EnumerateArray())
            {
                vectors.Add((float)value.GetDouble());
            }

            return vectors;
        }
    }
```

Expose as an MCP Tool (via nuget ModelContextProtocol.AspNetCore) :

```
using System.ComponentModel;
using ModelContextProtocol.Server;
using ms_doc_assistant.Services;

namespace ms_doc_assistant.Tools;

[McpServerToolType]
```

```
public class RagTool(QdrantRagService ragService)
{
    [McpServerTool(Name = "SearchKnowledgeBase")]
    [Description("Search for relevant technical documents in the knowledge base
    public async Task<string> SearchKnowledgeBase(
        [Description("The search query for the internal documentation or proced
    {
        return await ragService.BuildContextForQueryAsync(query);
    }
}
```

And now the bonus tracks !

Bonus #1 — Complete RAG loader

```
#:package Qdrant.Client@1.12.0

using System.Text.Json;
using Qdrant.Client;
using Qdrant.Client.Grpc;

var embeddingApiUrl = Environment.GetEnvironmentVariable("EMBEDDING_API_URL") ?
var embeddingModel = Environment.GetEnvironmentVariable("EMBEDDING_MODEL") ?? "
var qdrantHost = Environment.GetEnvironmentVariable("QDRANT_HOST") ?? "qdrant.q
var datasetsPath = Environment.GetEnvironmentVariable("DATASETS_PATH") ?? "../.
var datasetsSearchPattern = Environment.GetEnvironmentVariable("DATASETS_SEARCH

// Initialize HTTP client for embeddings
var httpClient = new HttpClient();

// Initialize Qdrant client
var qdrantClient = new QdrantClient(qdrantHost);

// Create or recreate collection
await CreateOrRecreateCollectionAsync("knowledge_base", 1024);

var jsonFiles = Directory.GetFiles(datasetsPath, datasetsSearchPattern);
Console.WriteLine($"Number of files found: {jsonFiles.Length}\n");

foreach (var filePath in jsonFiles)
{
    Console.WriteLine($"Processing file: {Path.GetFileName(filePath)}");
    using var document = JsonDocument.Parse(File.ReadAllText(filePath));

    var documents = document.RootElement.EnumerateArray().ToList();

    await Parallel.ForEachAsync(documents, new ParallelOptions { MaxDegreeOfPar
    {
```

```
            var title = doc.GetProperty("title").GetString();
            var category = doc.GetProperty("category").GetString();
            var tags = doc.GetProperty("tags");
            var summary = doc.GetProperty("summary");
            var content = doc.GetProperty("content");

            Console.WriteLine($"\n Processing Document: {title}");

            if (string.IsNullOrWhiteSpace(title))
                return;

            var titleEmbedding = await CreateEmbeddingAsync(title);

            var point = new PointStruct
            {
                Id = Guid.NewGuid(),
                Vectors = titleEmbedding.ToArray(),
                Payload =
                {
                    ["title"] = title,
                    ["category"] = category ?? "general",
                    ["tags"] = tags.EnumerateArray().Select(t => t.GetString()!).To
                    ["summary"] = summary.GetString() ?? "",
                    ["content"] = content.EnumerateArray().Select(s => s.GetString(
                }
            };

            await qdrantClient.UpsertAsync("knowledge_base", new[] { point });
        });
    }

    Console.WriteLine("\nProcessing completed!");

    #region Helper Methods

    // Create embedding from text using embedding API
    async Task<List<float>> CreateEmbeddingAsync(string text)
    {
        var escapedText = text.Replace("\\", "\\\\").Replace("\"", "\\\"");
        var requestBody = $"{{\"model\":\"{embeddingModel}\",\"input\":\"{escapedTe

        var content = new StringContent(requestBody, System.Text.Encoding.UTF8, "ap
        var response = await httpClient.PostAsync(embeddingApiUrl, content);
        response.EnsureSuccessStatusCode();

        var responseBody = await response.Content.ReadAsStringAsync();
        using var embeddingDoc = JsonDocument.Parse(responseBody);

        var embeddingArray = embeddingDoc.RootElement
            .GetProperty("data")[0]
            .GetProperty("embedding");

        var vectors = new List<float>();
```

```
        foreach (var value in embeddingArray.EnumerateArray())
        {
            vectors.Add((float)value.GetDouble());
        }

        return vectors;
    }

    // Create or recreate a Qdrant collection
    async Task CreateOrRecreateCollectionAsync(string collectionName, uint vectorSi
    {
        var collections = await qdrantClient.ListCollectionsAsync();
        if (collections.Contains(collectionName))
        {
            Console.WriteLine($"Collection '{collectionName}' already exists, delet
            await qdrantClient.DeleteCollectionAsync(collectionName);
            Console.WriteLine($"Collection '{collectionName}' deleted\n");
        }

        await qdrantClient.CreateCollectionAsync(
            collectionName: collectionName,
            vectorsConfig: new VectorParams { Size = vectorSize, Distance = Distanc
        );
        Console.WriteLine($"Collection '{collectionName}' created successfully\n");
    }

    #endregion
```

Bonus #2 — KServe Local Model Cache (reduce cold-starts drastically)

Key idea: enable `localmodel` in KServe → a **DaemonSet** coordinates per-node caches. You define:

- **ClusterStorageContainer** (how to download `hf://` models),
- **LocalModelNodeGroup** (where to store on nodes; PVC/hostPath + node affinity),
- **LocalModelCache** (which model to pre-download and to which node groups).

Your manifests (HF secret, storage initializer, NodeGroup, LocalModelCache, reconciler DaemonSet) are **kept exactly as provided** in your original text, including the tips about new nodes / SPOT nodes and the annotation trick to force reconciliation.

After that: delete your predictor pods → restart is **much faster** (model already on disk). If downloads fail, check logs in `kserve-localmodel-jobs`.

Conclusion

Self-hosting a **cloud-native RAG** stack is not only possible — it's clean:

- **KServe** serves **bge-m3** for embeddings (CPU-friendly),
- **Envoy AI Gateway** unifies access with an OpenAI-style API plus auth, quotas, and metrics,
- **Qdrant** stores vectors with strong recall/latency and simple operations.

Links

- KServe
- KServe chart values
- Envoy AI Gateway
- Qdrant
- Qdrant Helm
- Hugging Face bge-m3
- Install Envoy AI Gateway
- OpenAI backend with Envoy
- NuGet Qdrant.Client
- NuGet Aspire.Qdrant.Clien
- NuGet ModelContextProtocol.AspNetCore
- Knative
- KEDA
- cert-manager

Benchmarked **bge-m3** vs **gte-Qwen2–7B-instruct** on your data? Tuned Qdrant HNSW for 1024-D? Hit quirks with Envoy ↔ KServe pathing? Share your configs, metrics, and war stories.

Kubernetes Mlops Envoy Proxy Terraform AI

Follow

Published in T3CH

1.3K followers · Last published 7 hours ago

Snoop & Learn about Technology, AI, Hacking, Coding, Software, News, Tools, Leaks, Bug Bounty, OSINT & Cybersecurity !i! But, not limited 2, anything that is Tech Linked...You'll probably find here ! ;)—Stay ahead with Latest Tech News! -> You write about? Just ping to join !

Follow

Written by using System;

157 followers · 103 following

Medium blog from Matthieu Vlad, .NET Senior Dev, DevOps & Cloud Architect. Support me by send your tips : usingsystem.eth ❤️

No responses yet

Bgerby

What are your thoughts?

More from using System; and T3CH

In T3CH by using System;

Bitnami's Keycloak Shake-Up: What Broke, How to Buy Time, and Where to Migrate (with Terraform +...

This post is a follow-up to my earlier article, "Open-source and cloud-native identity management and gateway" .

Sep 18 97 1

In T3CH by Frost

How I Instantly Know You Used ChatGPT

These days, AI tools are everywhere, and ChatGPT is leading the pack. It's easy, and it's tempting to use for just about everything: school...

Sep 9 169 6

In T3CH by Ali Hamza

Mastering Docker on WSL2: A Complete Guide Without Docker Desktop

Streamline Your Docker Experience on Windows with WSL2 — No Docker Desktop Needed!

Jan 7 188 7

In T3CH by using System;

Deploying Your AKS Cluster with Terraform: Key Points for a Successful Production Rollout

Azure Kubernetes Service (AKS) is Microsoft Azure's fully managed Kubernetes offering. It simplifies many operational complexities around...

Dec 29, 2024 · 62 · 1

See all from using System;

See all from T3CH

Recommended from Medium

In Data, AI and Beyond by Julius Nyerere Nyambok

Generate AWS architectural diagrams using this simple method

A combination of LLMs and MCP Servers.

Oct 14 64 1

DhanushKumar

LLM poisoning

What is LLM poisoning ?

5d ago 278 12

In Netflix TechBlog by Netflix Technology Blog

Behind the Streams: Real-Time Recommendations for Live Events Part 3

By: Kris Range, Ankush Gulati, Jim Isaacs, Jennifer Shin, Jeremy Kelly, Jason Tu

6d ago 91 1

In AWS Tip by Osman ALP

Mastering Kubernetes Observability: Deploying Grafana with Loki, Tempo, Mimir, and Alloy

The Modern Observability Trinity: Metrics, Logs, Traces, and Telemetry Collection

Oct 13 10

In AWS in Plain English by Aman Pathak | DevOps | AWS | K8s | Terraform

AWS Outage Root Cause Revealed: How a DNS Mismatch Affected DynamoDB and Core Services

Discover what caused the October 2025 AWS us-east-1 outage—a DNS mismatch that broke DynamoDB and triggered cascading failures across...

6d ago 68 4

In Opensearch Stories by Franco martin

Running OpenSearch in AWS

Is it any good?

Oct 7 1

See more recommendations