



# CDW Documentation

## Advanced RAG Implementation in Azure

---

# Advanced RAG Implementation in Azure

## Prerequisites

- **Azure Web App Resource Created**
- **Azure Bot Resource Created**
- **Key Vault Created**
- **OpenAI Chat Bot created**
- **OpenAI Embedding Model Created**
- **Azure AI Search Resource and Index created**
- **Azure CLI (v2.60+): az version**
- **Node.js 20 installed**
- **Teams access** with permission to upload custom apps (tenant setting).

## Embed Your Dataset in Azure AI Search

### 1) Configure environment

Create/update **.env** (local) for ingest. Use the **Admin** search key here:

```
# Azure Search (Admin for ingest)
```

```
AZURE_SEARCH_ENDPOINT=https://<your-search>.search.windows.net
```

```
AZURE_SEARCH_API_KEY=<SEARCH-ADMIN-KEY>
```

```
AZURE_SEARCH_INDEX_NAME=docs
```

```
# Azure OpenAI (Embeddings + chat)
```

```
OPENAI_ENDPOINT=https://<your-aoai>.openai.azure.com
```

```
OPENAI_API_KEY=<AOAI-KEY>
```

```
OPENAI_API_VERSION=2024-06-01
```

```
OPENAI_EMBEDDING_DEPLOYMENT=<your-embedding-deployment>
```

```
# Index field mapping (must match your index schema)
```

```
VECTOR_FIELD=contentVector
```

```
CONTENT_FIELD=contents # ← your index uses "contents"
```

TITLE\_FIELD=title

URL\_FIELD=url

METADATA\_FIELD=metadata

PARENT\_ID\_FIELD=parentId

# Retrieval tuning (ingest uses embedding; runtime uses these too)

USE\_SEMANTIC\_RANKER=false # leave false during debugging; not needed for ingest

TOP\_K=8

TEMPERATURE=0.0

MAX\_TOKENS\_ANSWER=800

## 2) Install deps

npm install

## 3) Create/Update index schema (done by ingest.js)

- Our ingest.js creates docs with fields like:
  - id (key)
  - title (searchable, retrievable)
  - contents (searchable, retrievable) ← **critical**
  - url (retrievable)
  - metadata (retrievable, facetable/filterable if you want)
  - parentId (retrievable, filterable)
  - contentVector (vector field with your embedding profile)
- If index exists, we try to update it; if schema drift is large, **delete the old index** and re-run.

Common schema errors we fixed earlier

- **Invalid field name** - ensure no inline comments / stray text in env var names.
- **'ext' not present** - we removed unsupported subfields from metadata.
- **Invalid keys** - we sanitized IDs (allowUnsafeKeys not needed).

## 4) Put your documents in the ingest folder

ingest/

data/

Affirm-onboarding.docx

Deploying a Real-Time Endpoint.docx

Etc.

## 5) Run the ingest

```
node ingest/ingest.js
```

Expected output (example):

Index exists. Updating (if schema changed)...

Found 41 docs

Uploading 93 chunks...

Uploaded 93/93

Ingest complete.

## Upload/Deploy Code to the Web App

### Zip Deploy (fastest manual)

**Note:** Example code for the advanced RAG implementation is included at the bottom of the document.

1. Create a zip **from your project root** (include package.json or web.config as needed).
  - **macOS/Linux:**

```
zip -r app.zip . -x "*.git*" -x "node_modules/*" -x "*.DS_Store"
```

- **Windows (PowerShell):**

```
Compress-Archive -Path * -DestinationPath app.zip -Force
```

```
<HTML><ol start="2" style="list-style-type: decimal;"></HTML>  
<HTML><li></HTML><HTML><p></HTML>Push the  
zip:<HTML></p></HTML><HTML></li></HTML><HTML></ol></HTML>
```

```
az webapp deploy \
```

```
-resource-group $RG \
```

```
-name $APP \
```

```
-src-path app.zip \
```

```
-type zip
```

```
<HTML><ol start="3" style="list-style-type: decimal;"></HTML>
<HTML><li></HTML><HTML><p></HTML>Check
logs:<HTML></p></HTML><HTML></li></HTML><HTML></ol></HTML>
```

```
az webapp log tail -g $RG -n $APP
```

## Configure Environment Variables (App Settings)

App settings are injected as environment variables at runtime.

**Note:** It is recommended to keep secrets such as API keys or Endpoints in Azure Key Vault. For a secure implementation, please keep secrets in Key Vault and reference those in their coordinated environment variables.

1. Web App → **Configuration** → **App settings**
2. Add Key:Value entries for the environment variables necessary for your RAG application:
  - AZURE\_SEARCH\_API\_KEY - "Azure Search API Key"
  - AZURE\_SEARCH\_ENDPOINT - "Azure Search Endpoint"
  - AZURE\_SEARCH\_INDEX\_NAME - "Name of your index created in Azure AI Search"
  - CONTENT\_FIELD - contents
  - LLM\_RERANK - *boolean*
  - MAX\_TOKENS\_ANSWER - 1000
  - METADATA\_FIELD - metadata
  - MicrosoftAppId - "App ID for your Azure Bot"
  - MicrosoftAppPassword - "App Password for your Azure Bot"
  - MicrosoftAppTenantId - "Tenant ID for your Azure Bot"
  - MicrosoftAppType - "App Type for your Azure Bot"
  - NODE\_ENV - production
  - OPENAI\_API\_KEY - "OpenAI API Key"
  - OPENAI\_API\_VERSION - "OpenAI Model Version"
  - OPENAI\_DEPLOYMENT - "OpenAI Model Deployment Name"
  - OPENAI\_EMBEDDING\_DEPLOYMENT - "OpenAI Embedding Model Deployment Name"
  - OPENAI\_ENDPOINT - "OpenAI Endpoint"
  - PARENT\_ID\_FIELD - parentID
  - RETRIEVER\_MULTIQUERY\_N - "Value between 1-5"
  - RETRIEVER\_USE\_HYDE - *boolean*
  - SCM\_DO\_BUILD\_DURING\_DEPLOYMENT - *boolean*
  - TEMPERATURE - 0
  - TITLE\_FIELD - title
  - TOP\_K - "Value between 1-12"
  - URL\_FIELD - url
  - USE\_SEMANTIC\_RANKER - *boolean*
  - VECTOR\_FIELD - contentVector
3. **Save** and **Restart** the Web App.

## Enable Teams Channel (in Azure Bot)

1. Open your **Bot Channels Registration**.
2. **Channels** → **Microsoft Teams** → **Configure** → **Save**.
3. Optionally enable **Streaming** if your bot uses it.

**Tenant restriction:** Your tenant admin may require approval for custom apps. Confirm **Teams admin center** → **Teams apps** → **Setup policies** and **Manage apps** allow custom uploads.

## Create the Teams Bot

You can use **Teams Developer Portal** (recommended) or hand-craft manifest.json.

### Developer Portal method

1. In Teams, open **Developer Portal** (app) → **Apps** → **New app**.
2. **App details:** Name, short/long description, developer info, icons:
  - **Color icon:** 192×192 PNG
  - **Outline icon:** 32×32 PNG (transparent background)
3. **Bots** → **Set up** → *Existing bot* → paste your **Microsoft App ID**.
  - Scope: **Personal**, **Team**, and/or **Group chat** per your needs.
  - **Commands** (optional): define /help, /status, etc.
4. **Single-sign on** (optional): you can configure SSO with your Entra app here.
5. **Domains and permissions:** include your web domain(s) if you use tabs.
6. **Publish** → **Test and distribute** → **Install to Teams** (or **Submit to org** if required).

### Manual manifest (advanced)

Create a folder with manifest.json, color.png, outline.png, zip them, then upload in **Teams Admin Center** or **Developer Portal**.

### Sample manifest.json (v1.13)

```
{
  "$schema":
  "https://developer.microsoft.com/en-us/json-schemas/teams/v1.22/MicrosoftTeams.schema.json",
  "manifestVersion": "1.13",
  "version": "1.0.0",
  "id": "<GUID>",
  "packageName": "com.example.ragbot",
  "name": { "short": "RAG Bot", "full": "RAG Bot for Q&A" },
  "developer": {
    "name": "Your Org",
```

```
“websiteUrl”: “https://www.example.com”,
“privacyUrl”: “https://www.example.com/privacy”,
“termsOfUseUrl”: “https://www.example.com/tos”
},
“description”: {
“short”: “Bot for testing purposes”,
“full”: “An enterprise knowledge assistant deployed on Azure.”
},
“icons”: { “color”: “color.png”, “outline”: “outline.png” },
“accentColor”: “#6264A7”,
“bots”: [
{
“botId”: “<MICROSOFT_APP_ID>”,
“scopes”: [“personal”, “team”, “groupchat”],
“isNotificationOnly”: false
}
],
“permissions”: [“identity”, “messageTeamMembers”],
“validDomains”: [“app-rag-prod.azurewebsites.net”]
}
```

Zip contents (no parent folder) and upload.

## Install in Teams and Test

- In Teams, click Apps Manage your apps Upload an app
- Click Upload a custom app, then upload your Zip file created in the Teams Developer Portal.
- If you see errors, check **App Service logs** and **Bot resource** → **Issues** panel.
- You will then be able to message your bot like any other Teams user.

## Example Code for Azure Web App

bench.js

```
require('dotenv').config();

const os = require('os');

const { answerQuestion } = require('../openaiService');

function arg(name, def) {

  const prefix = `-${name}=`;

  const a = process.argv.find(x => x.startsWith(prefix));

  return a ? a.slice(prefix.length) : def;

}

const q = arg('q', 'What is our onboarding process?');

const n = Number(arg('n', 20));

const conc = Number(arg('concurrency', 4));

function sleep(ms) { return new Promise(r => setTimeout(r, ms)); }

async function worker(id, jobs, results) {

  while (true) {

    const i = jobs.next();

    if (i.done) break;

    const t0 = Date.now();

    try {

      const r = await answerQuestion(q);

      const t = Date.now() - t0;

      results.push({ ok: true, latencyMs: t, usage: r.metrics.usage });

      process.stdout.write('.');

    } catch (e) {

      const t = Date.now() - t0;
```

```
results.push({ ok: false, latencyMs: t, error: String(e) });  
  
process.stdout.write('x');  
  
}  
  
await sleep(50);  
  
}  
  
}
```

```
async function main() {  
  
const jobs = (function*(){ for (let i=0;i<n;i++) yield i; }());  
  
const results = [];  
  
const ps = [];  
  
for (let i=0;i<conc;i++) ps.push(worker(i, jobs, results));  
  
await Promise.all(ps);  
  
console.log('\nDone.');
```

  

```
const lat = results.map(r => r.latencyMs).sort1;  
  
}
```

```
if (require.main === module) main().catch(console.error);
```

bot.js

```
const { ActivityHandler, MessageFactory } = require('botbuilder');  
  
const { answerQuestion } = require('./openaiService');
```

```
function renderSources(citations) {
```

```
if (!citations || !citations.length) return ; const lines = citations.map(c => { const title  
= c.title || c.id || 'Source'; const url = c.url ? ` (${c.url})` : ;
```

```
return ` - ${c.key} **${title}**${url}`;
```

```
});

return `\\n\\n**Sources**\\n${lines.join('\\n')}`;

}
```

```
class EchoBot extends ActivityHandler {
```

```
  constructor() {
```

```
    super();
```

```
    this.onMessage(async (context, next) => {
```

```
      const userInput = (context.activity.text || ).trim(); if (!userInput) { await
      context.sendActivity('Please type a question.');
```

```
      return; } try { const result = await answerQuestion(userInput); const text =
      `${result.answer}${renderSources(result.citations)}`; await
      context.sendActivity(MessageFactory.text(text, text)); } catch (err) {
      console.error('Error processing message:', err); await
      context.sendActivity('Oops, something went wrong when talking to AI.');
```

```
      await next(); }); this.onMembersAdded(async (context, next) => { const
      welcomeText = 'Hello and welcome! Ask me any question about our documents.';
      for (const member of context.activity.membersAdded) { if (member.id !==
      context.activity.recipient.id) { await
      context.sendActivity(MessageFactory.text(welcomeText, welcomeText)); } }
      await next(); }); } } module.exports.EchoBot = EchoBot; evaluate.js
      require('dotenv').config(); const fs = require('fs'); const path =
      require('path'); const { answerQuestion, _internals } =
      require('../openaiService'); // ----- CLI args ----- function arg(name,
      def) { const p = `-${name}=`; const a = process.argv.find(x =>
      x.startsWith(p)); if (!a) return def; const v = a.slice(p.length); if (v ===
      'true') return true; if (v === 'false') return false; const n = Number(v);
      return Number.isFinite(n) ? n : v; } const K = arg('k', 5); const LIMIT =
      arg('limit', 0); // 0 = all const USE_JUDGE = arg('judge', false) ||
      String(process.env.EVAL_USE_JUDGE || 'false').toLowerCase() === 'true'; //
      ----- Helpers ----- function loadJsonl(p) { const lines =
      fs.readFileSync(p, 'utf8').split(/\\r?\\n/).filter(Boolean); return
      lines.map2); const hits = refs.filter(r => ids.has(r.id)).length; return { hits,
      total: refs.length, recall: hits / refs.length }; } function
      parentRecallAtK(retrieved, parentRefs, k = K) { if
      (!Array.isArray(parentRefs) || !parentRefs.length) return null; const pids =
      new Set(retrieved.slice(0, k).map(r => r.parentId || r.id)); const hits =
      parentRefs.filter(r => pids.has(r.id)).length; return { hits, total:
      parentRefs.length, recall: hits / parentRefs.length }; } function
      mustMentionCoverage(answer, must) { if (!Array.isArray(must) || !must.length)
      return null; const a = (answer || ).toLowerCase();
```

```
const checks = must.map(m => ({ term: m, present: a.includes(String(m).toLowerCase()) }));

const pct = checks.reduce3);

if (!xs.length) return null;

return xs.reduce4) {

console.error('No dataset.jsonl found in ./eval. Create one (JSONL) with at least {"id","question"}.');

process.exit(1);

}

const rowsAll = loadJsonl(datasetPath);

const rows = LIMIT ? rowsAll.slice(0, LIMIT) : rowsAll;

const results = [];

for (const row of rows) {

const options = {};

if (row.filter) options.filter = row.filter; // OData filter passthrough

const res = await answerQuestion(row.question, options);

const rDoc = recallAtK(res.retrieved, row.references, K);

const rPar = parentRecallAtK(res.retrieved, row.parent_refs, K);

const mention = mustMentionCoverage(res.answer, row.must_mention);

let judge = null;

if (USE_JUDGE) judge = await llmJudge(row, res);

results.push({

id: row.id,

question: row.question,

references: row.references || null,
```

```
parent_refs: row.parent_refs || null,
must_mention: row.must_mention || null,
filter: row.filter || null,
answer: res.answer,
citations: res.citations,
metrics: {
  retrievalLatencyMs: res.metrics.retrievalLatencyMs,
  generationLatencyMs: res.metrics.generationLatencyMs,
  usage: res.metrics.usage || null,
  expansions: res.metrics.expansions || null,
  recallAtK: rDoc,
  parentRecallAtK: rPar,
  mustMention: mention,
  judge
}
});

const recStr = rDoc ? rDoc.recall.toFixed(2) : 'n/a';
const lat = res.metrics.generationLatencyMs;
console.log(`✓ ${row.id} recall@${K}=${recStr} gen=${lat}ms`);
}

// Summary
const p = path.join(__dirname, `results-${Date.now()}.json`);
fs.writeFileSync(p, JSON.stringify(results, null, 2));
console.log('Saved', p);

const rVals = results.map(r => r.metrics.recallAtK?.recall).filter(v => typeof v === 'number');
```

```
const prVals = results.map(r => r.metrics.parentRecallAtK?.recall).filter(v => typeof v === 'number');
const retLat = results.map(r => r.metrics.retrievalLatencyMs).filter(Number.isFinite);
const genLat = results.map(r => r.metrics.generationLatencyMs).filter(Number.isFinite);

const gScores = results.map(r => r.metrics.judge?.groundedness).filter(Number.isFinite);
const relScores = results.map(r => r.metrics.judge?.relevance).filter(Number.isFinite);
const compScores = results.map(r => r.metrics.judge?.completeness).filter(Number.isFinite);

const summary = {
  count: results.length,
  k: K,
  avgRecallAtK: mean(rVals),
  avgParentRecallAtK: mean(prVals),
  avgRetrievalLatencyMs: mean(retLat),
  avgGenerationLatencyMs: mean(genLat),
  judgeAverages: USE_JUDGE ? {
    groundedness: mean(gScores),
    relevance: mean(relScores),
    completeness: mean(compScores)
  } : null
};

console.log('\nSummary:\n', JSON.stringify(summary, null, 2));
}

if (require.main === module) {
  main().catch(err => { console.error(err); process.exit(1); });
}
```

```
index.js

const path = require('path');

require('dotenv').config({ path: path.join(__dirname, '.env') });

const restify = require('restify');

const {
  CloudAdapter,
  ConfigurationServiceClientCredentialFactory,
  createBotFrameworkAuthenticationFromConfiguration
} = require('botbuilder');

// (Optional) Application Insights for runtime telemetry
try {
  if (process.env.APPINSIGHTS_CONNECTION_STRING) {
    require('applicationinsights').setup().start();
    console.log('[init] Application Insights enabled');
  }
} catch (e) {
  console.warn('[init] App Insights init skipped:', e?.message || e);
}

// Import your bot implementation
const { EchoBot } = require('./bot');

// — Server setup —
const server = restify.createServer();
server.use(restify.plugins.bodyParser());
```

```
server.use(restify.plugins.queryParser());

// Prefer Azure's injected PORT. Fall back to 8080 locally, then 3978 (bot default).
const rawPort = process.env.PORT || process.env.port || 8080;
const port = Number(rawPort);
console.log('[startup] PORT from env =', rawPort, '→ binding to', port);

server.listen(port, () => {
  console.log(` ${server.name} listening on ${server.url}`);
  console.log(` Node: ${process.version} | ENV: ${process.env.NODE_ENV || 'development'}`);
  console.log('POST /api/messages (Bot endpoint)');
  console.log('GET /api/health (Health check)');
});

// Health endpoint for App Service
server.get('/api/health', (_req, res, next) => {
  res.send(200, {
    ok: true,
    uptimeSec: Math.round(process.uptime()),
    node: process.version
  });
  return next();
});

// 1) What is Search returning right now?
const { _internals } = require('./openaiService');

server.get('/api/search-diag', async (req, res) => {
  try {
```

```
const { _internals } = require('./openaiService');

const q = req.query.q || ''; const items = await _internals.searchOnce({ query: q });
res.send(200, { ok: true, q, returned: items.length, items: items.slice(0,
5).map(x => ({ id: x.id, title: x.title, contentPreview: (x.content || '').slice(0,
200)

}))

});

} catch (e) {

res.send(503, { ok: false, error: e?.message || String(e) });

}

});

// 2) End-to-end answer (to see retrieved count + citations)

const { answerQuestion } = require('./openaiService');

server.get('/api/answer', async (req, res) => {

try {

const { answerQuestion } = require('./openaiService');

const q = req.query.q || ''; const out = await answerQuestion(q); res.send(200, { ok:
true, query: q, retrievedCount: out.retrieved?.length || 0, citationsCount:
out.citations?.length || 0, answer: out.answer, sampleRetrieved:
out.retrieved?.slice(0,3)?.map(r => ({ id: r.id, title: r.title, preview:
(r.content || '').slice(0,160)

})) || []

});

} catch (e) {

res.send(503, { ok: false, error: e?.message || String(e) });

}

});

// How many docs are in the index?

server.get('/api/search-stats', async (_req, res) => {

try {
```

```
const { SearchClient, AzureKeyCredential } = require('@azure/search-documents');

const endpoint = process.env.AZURE_SEARCH_ENDPOINT.trim();

const index = process.env.AZURE_SEARCH_INDEX_NAME.trim();

const key = (process.env.AZURE_SEARCH_QUERY_KEY || process.env.AZURE_SEARCH_API_KEY).trim();

const client = new SearchClient(endpoint, index, new AzureKeyCredential(key));

const count = await client.getDocumentsCount();

res.send(200, { ok: true, index, count });

} catch (e) {

res.send(503, { ok: false, error: e?.message || String(e) });

}

});

// Vector diag: embed the query and run a pure vector search

server.get('/api/search-diag-vector', async (req, res) => {

try {

const q = (req.query && req.query.q) ? String(req.query.q) : ; if (!q) return res.send(400, { ok: false, error: 'q required' }); const { _internals } = require('./openaiService'); const vec = await _internals.embedMemo(q); const items = await _internals.searchOnce({ query: , vector: vec });

res.send(200, {

ok: true,

q,

returned: items.length,

items: items.slice(0, 5).map(x => ({

id: x.id,

title: x.title,

contentPreview: (x.content || ).slice(0, 200) }))) }); } catch (e) { res.send(503, { ok: false, error: e?.message || String(e) }); } }); const axios = require('axios'); // Ultra-raw: call Azure Search REST API directly (bypasses SDK iterator) server.get('/api/search-diag-raw', async (req, res) =>
```

```
{ try { const endpoint = String(process.env.AZURE_SEARCH_ENDPOINT ||
).trim().replace(/V+$/, ); const index = String(process.env.AZURE_SEARCH_INDEX_NAME
|| ).trim();

const apiKey = String(process.env.AZURE_SEARCH_QUERY_KEY ||
process.env.AZURE_SEARCH_API_KEY || ).trim(); if (!endpoint || !index ||
!apiKey) { return res.send(400, { ok: false, error: 'Missing
endpoint/index/apiKey env vars' }); } const q = (req.query && typeof
req.query.q === 'string') ? req.query.q : ;

const url = `${endpoint}/indexes('${encodeURIComponent(index)}')/docs/search.post.search?api-
version=2024-07-01`;

// IMPORTANT: set select conservatively; match your schema

// If your text field is `contents`, keep "contents" below.

const body = {

search: q, // (match-all) or your query string top: 5, select:
'id,title,contents,url' // adjust if your fields are named differently };
const { data } = await axios.post(url, body, { headers: { 'api-key': apiKey,
'Content-Type': 'application/json', 'Accept':
'application/json;odata.metadata=none' }, timeout: 10000 }); const items =
Array.isArray(data.value) ? data.value.map(d => ({ id: d.id, title: d.title ||
null, contentPreview: String(d.contents ?? d.content ?? ).slice(0, 200),

url: d.url || null

})) : [];

res.send(200, { ok: true, q, returned: items.length, items });

} catch (e) {

res.send(503, { ok: false, error: e?.response?.data?.error?.message || e?.message || String(e) });

}

});

// Simple root info (optional)

server.get('/', (_req, res, next) => {

res.send(200, { ok: true, message: 'RAG bot is running', time: new Date().toISOString() });

return next();
```

```
});

// — Bot Framework adapter (credentials from env) —
const credentialsFactory = new ConfigurationServiceClientCredentialFactory({
  MicrosoftAppId: process.env.MicrosoftAppId,
  MicrosoftAppPassword: process.env.MicrosoftAppPassword, // aka Client Secret
  MicrosoftAppType: process.env.MicrosoftAppType || 'MultiTenant',
  MicrosoftAppTenantId: process.env.MicrosoftAppTenantId
});

const botFrameworkAuthentication =
  createBotFrameworkAuthenticationFromConfiguration(null, credentialsFactory);

const adapter = new CloudAdapter(botFrameworkAuthentication);

// Catch-all for errors
adapter.onTurnError = async (context, error) => {
  console.error('[onTurnError]', error);
  try {
    await context.sendTraceActivity(
      'OnTurnError Trace',
      `${error}`,
      'https://www.botframework.com/schemas/error',
      'TurnError'
    );
    await context.sendActivity('Oops—something went wrong processing your message.');
```

```
  } catch (_) {
    // ignore secondary failures
```

```
}  
};  
  
// Create the bot instance  
const bot = new EchoBot();  
  
// Bot messages endpoint (required by Azure Bot Service)  
server.post('/api/messages', async (req, res) => {  
  await adapter.process(req, res, (context) => bot.run(context));  
});  
  
// Streaming (Teams / Skills)  
server.on('upgrade', async (req, socket, head) => {  
  const streamingAdapter = new CloudAdapter(botFrameworkAuthentication);  
  streamingAdapter.onTurnError = adapter.onTurnError;  
  await streamingAdapter.process(req, socket, head, (context) => bot.run(context));  
});  
  
ingest.js  
require('dotenv').config();  
const mammoth = require('mammoth');  
const fs = require('fs');  
const path = require('path');  
const axios = require('axios');  
  
const { SearchClient, SearchIndexClient, AzureKeyCredential } = require('@azure/search-  
documents');  
  
const SEARCH_ENDPOINT = process.env.AZURE_SEARCH_ENDPOINT;
```

```
const SEARCH_INDEX = process.env.AZURE_SEARCH_AZURE_SEARCH_INDEX_NAME ||
process.env.AZURE_SEARCH_INDEX_NAME; // support both if user mis-typed

const SEARCH_KEY = process.env.AZURE_SEARCH_API_KEY;

const OPENAI_ENDPOINT = process.env.OPENAI_ENDPOINT;

const OPENAI_API_KEY = process.env.OPENAI_API_KEY;

const OPENAI_API_VERSION = process.env.OPENAI_API_VERSION || '2024-06-01';

const OPENAI_EMBEDDING_DEPLOYMENT = process.env.OPENAI_EMBEDDING_DEPLOYMENT || 'text-embedding-3-large';

const VECTOR_FIELD = process.env.VECTOR_FIELD || 'contentVector';

const CONTENT_FIELD = process.env.CONTENT_FIELD || 'content';

const TITLE_FIELD = process.env.TITLE_FIELD || 'title';

const URL_FIELD = process.env.URL_FIELD || 'url';

const METADATA_FIELD = process.env.METADATA_FIELD || 'metadata';

const PARENT_ID_FIELD = process.env.PARENT_ID_FIELD || 'parentId';

function arg(name, def){ const p=`-${name}`; const a=process.argv.find(x⇒x.startsWith(p));
return a ? a.slice(p.length) : def; }

const dataDir = require('path').resolve(arg('dir', require('path').join(__dirname, 'data')));

const indexClient = new SearchIndexClient(SEARCH_ENDPOINT, new
AzureKeyCredential(SEARCH_KEY));

const searchClient = new SearchClient(SEARCH_ENDPOINT, SEARCH_INDEX, new
AzureKeyCredential(SEARCH_KEY));

function approxTokenLen(s) { return Math.ceil5 {
text = fs.readFileSync(full, 'utf8');
} else if (ext === '.json') {
// Accept JSON too—either stringify or use a specific field if you prefer
```

```
const raw = fs.readFileSync(full, 'utf8');

try {
  const obj = JSON.parse(raw);
  text = typeof obj === 'string' ? obj : JSON.stringify(obj, null, 2);
} catch {
  text = raw;
}

} else {
  // Unsupported type — skip
  continue;
}

if (!text) continue;

const title = path.parse(f).name;
docs.push({
  id: title,
  title,
  url: null,
  content: text,
  metadata: { source: f }
});
}

return docs;
}

async function run() {
```

```

await ensureIndex();

const docs = await readDocs(dataDir);

console.log(` Found ${docs.length} docs`);

const actions = [];

for (const doc of docs) {

const chunks = chunkText(doc.content);

// Make the parent id safe for Azure Search keys: letters/digits/_/!/=

const parentKey = String(doc.id || ).replace(/[^\A-Za-z0-9_\-\=]/g, '_'); for (let i =
0; i < chunks.length; i++) { const chunk = chunks[i]; // Chunk key is
parentKey + '-' + index (no spaces or '#') const id = `${parentKey}-${i}`;
const vec = await embed(chunk); actions.push({ '@search.action':
'mergeOrUpload', id, [TITLE_FIELD]: doc.title, [URL_FIELD]: doc.url,
[CONTENT_FIELD]: chunk, [METADATA_FIELD]: doc.metadata, // keep only declared
subfields (e.g., source/category/tags) [PARENT_ID_FIELD]: parentKey,
[VECTOR_FIELD]: vec }); } console.log(`Uploading ${actions.length}
chunks...`); for (let i = 0; i < actions.length; i += 1000) { const batch =
actions.slice(i, i + 1000); await searchClient.mergeOrUploadDocuments(batch);
console.log(`Uploaded ${Math.min(i + batch.length,
actions.length)}/${actions.length}`); } console.log('Ingest complete.');
```

```

if (require.main === module) { run().catch(err => { console.error(err);
process.exit(1); }); } openaiService.js require('dotenv').config(); const
axios = require('axios'); const { SearchClient, AzureKeyCredential } =
require('@azure/search-documents'); // --- Helpers / Config --- const
cleanName = (v, def) => (v ?? def ?? ).split('#')[0].trim();

const cleanBool = (v, def = 'false') => String(v ?? def).trim().toLowerCase() === 'true';

const cleanNum = (v, def) => {

const n = Number(String(v ?? ).trim()); return Number.isFinite(n) ? n : def; };
const sanitizeUrl = s => String(s || ).trim().replace(/\/+$/, ); const
SEARCH_ENDPOINT = sanitizeUrl(process.env.AZURE_SEARCH_ENDPOINT); const
SEARCH_INDEX = cleanName(process.env.AZURE_SEARCH_INDEX_NAME); const
SEARCH_KEY = cleanName(process.env.AZURE_SEARCH_API_KEY); const
OPENAI_ENDPOINT = sanitizeUrl(process.env.OPENAI_ENDPOINT); const
OPENAI_API_KEY = cleanName(process.env.OPENAI_API_KEY); const
OPENAI_API_VERSION = cleanName(process.env.OPENAI_API_VERSION, '2024-06-01');
const OPENAI_DEPLOYMENT = cleanName(process.env.OPENAI_DEPLOYMENT, 'gpt-4o-
mini'); const OPENAI_EMBEDDING_DEPLOYMENT =
cleanName(process.env.OPENAI_EMBEDDING_DEPLOYMENT, 'text-embedding-3-large');
const VECTOR_FIELD = cleanName(process.env.VECTOR_FIELD, 'contentVector');
const CONTENT_FIELD = cleanName(process.env.CONTENT_FIELD, 'content'); //
```

```

env-preferred const TITLE_FIELD = cleanName(process.env.TITLE_FIELD,
'title'); const URL_FIELD = cleanName(process.env.URL_FIELD, 'url'); const
METADATA_FIELD = cleanName(process.env.METADATA_FIELD, 'metadata'); const
PARENT_ID_FIELD = cleanName(process.env.PARENT_ID_FIELD, null); const
SEMANTIC_CONFIGURATION = cleanName(process.env.SEMANTIC_CONFIGURATION,
'default'); const USE_SEMANTIC_RANKER =
cleanBool(process.env.USE_SEMANTIC_RANKER, 'true'); const
SEMANTIC_USE_CAPTIONS = cleanBool(process.env.SEMANTIC_USE_CAPTIONS,
'false'); const SEMANTIC_USE_ANSWERS =
cleanBool(process.env.SEMANTIC_USE_ANSWERS, 'false'); const LLM_RERANK =
cleanBool(process.env.LLM_RERANK, 'false'); const TOP_K =
cleanNum(process.env.TOP_K, 8); const TEMPERATURE =
cleanNum(process.env.TEMPERATURE, 0); const MAX_TOKENS_ANSWER =
cleanNum(process.env.MAX_TOKENS_ANSWER, 800); // Retrieval feature toggles
const RETRIEVER_MULTIQUERY_N = cleanNum(process.env.RETRIEVER_MULTIQUERY_N,
1); const RETRIEVER_USE_HYDE = cleanBool(process.env.RETRIEVER_USE_HYDE,
'false'); // Clients const searchClient = new SearchClient(SEARCH_ENDPOINT,
SEARCH_INDEX, new AzureKeyCredential(SEARCH_KEY)); // Timing helpers
function nowMs() { return Date.now(); } async function searchOnceRest({
query, top = TOP_K }) { const endpoint =
String(process.env.AZURE_SEARCH_ENDPOINT || ).trim().replace(/\/+$/, ); const index
= String(process.env.AZURE_SEARCH_INDEX_NAME || ).trim();

```

```

const apiKey = String(process.env.AZURE_SEARCH_QUERY_KEY ||
process.env.AZURE_SEARCH_API_KEY || ).trim(); const url =
`${endpoint}/indexes('${encodeURIComponent(index)}')/docs/search.post.search?
api-version=2024-07-01`; const body = { search: query ?? ,top };

```

```

const { data } = await axios.post(url, body, {

```

```

headers: {

```

```

'api-key': apiKey,

```

```

'Content-Type': 'application/json',

```

```

'Accept': 'application/json;odata.metadata=none'

```

```

},

```

```

timeout: 10000

```

```

});

```

```

const arr = Array.isArray(data.value) ? data.value : [];

```

```

return arr.map(d => ({

```

```

id: d.id,

```

```

content: String(d[CONTENT_FIELD] ?? d.contents ?? d.content ?? ), title: d[TITLE_FIELD] ??

```

```

d.title ?? null, url: d[URL_FIELD] ?? d.url ?? null, metadata:
d[METADATA_FIELD] ?? d.metadata ?? {}, parentId: PARENT_ID_FIELD ?
(d[PARENT_ID_FIELD] ?? null) : null, score: d['@search.score'] ?? null,
rerankScore: null })); } // --- Reliability: retry + memo --- const sleep
= (ms) => new Promise(r => setTimeout(r, ms)); async function withRetry(fn,
desc = 'request', { tries = 4, base = 300 } = {}) { let lastErr; for (let i =
0; i < tries; i++) { try { return await fn(); } catch (e) { const status =
e?.response?.status; const retryable = status === 206 || status === 408 ||
status === 429 || (status >= 500 && status <= 599); if (!retryable || i ===
tries - 1) { lastErr = e; break; } const wait = base * Math.pow(2, i) +
Math.floor(Math.random() * 100); console.warn(`${desc} failed
(status=${status}), retrying in ${wait}ms...`); await sleep(wait); } } throw
lastErr; } const _memo = new Map(); const _get = (k) => _memo.get(k); const
_set = (k, v) => { if (_memo.size > 500) _memo.clear(); _memo.set(k, v); };
// --- Azure OpenAI calls --- async function embed(text) { const url =
`${OPENAI_ENDPOINT}/openai/deployments/${OPENAI_EMBEDDING_DEPLOYMENT}/embeddi
ngs?api-version=${OPENAI_API_VERSION}`; const { data } = await withRetry( ()
=> axios.post(url, { input: text }, { headers: { 'api-key': OPENAI_API_KEY,
'Content-Type': 'application/json' } })), 'embed' ); return
data.data[0].embedding; } async function embedMemo(text) { const k =
`emb:${text}`; const hit = _get(k); if (hit) return hit; const v = await
embed(text); _set(k, v); return v; } async function chat(messages, opts =
{}) { const url =
`${OPENAI_ENDPOINT}/openai/deployments/${OPENAI_DEPLOYMENT}/chat/completions?
api-version=${OPENAI_API_VERSION}`; const payload = { messages, temperature:
TEMPERATURE, max_tokens: MAX_TOKENS_ANSWER, ...opts }; const { data } = await
withRetry( () => axios.post(url, payload, { headers: { 'api-key':
OPENAI_API_KEY, 'Content-Type': 'application/json' } })), 'chat' ); return
data; } // --- Query Expansion --- async function
multiQueryExpand(userQuery, n = 3) { const k = `mq:${n}:${userQuery}`; const
hit = _get(k); if (hit) return hit; const sys = { role: 'system', content:
'You expand search queries. Return only diverse paraphrases as JSON array of
strings.' }; const usr = { role: 'user', content: `Create ${n} diverse
rewrites of:\n"${userQuery}"\nReturn JSON array, no prose.` }; const res =
await chat([sys, usr], { temperature: 0.2, max_tokens: 200 }); let arr = [];
try { arr = JSON.parse(res.choices[0].message.content.trim()); } catch { arr
= []; } const out = Array.isArray(arr) ? arr.filter(s => typeof s ===
'string') : []; _set(k, out); return out; } async function
hydeDoc(userQuery) { const k = `hyde:${userQuery}`; const hit = _get(k); if
(hit) return hit; const sys = { role: 'system', content: 'You draft a concise
answer-like passage that could hypothetically match a knowledge base. Keep it
< 1600 chars.' }; const usr = { role: 'user', content: userQuery }; const res
= await chat([sys, usr], { temperature: 0.2, max_tokens: 400 }); const out =
(res.choices[0].message.content || ).slice(0, 1600);

_set(k, out);

return out;

}

```

```
// ----- Retrieval Strategies -----  
  
function buildCommonOptions({ filter, select } = {}) {  
  // Only select what we know exists from env; no alternates here.  
  
  const want = select || [  
    CONTENT_FIELD, // e.g., "contents"  
    TITLE_FIELD, // e.g., "title"  
    URL_FIELD, // e.g., "url"  
    METADATA_FIELD, // e.g., "metadata"  
    'id',  
    PARENT_ID_FIELD // may be null  
  ].filter(Boolean);  
  
  const base = {  
    top: TOP_K,  
    includeTotalCount: false  
  };  
  // select: want  
  
  if (filter) base.filter = filter;  
  
  if (USE_SEMANTIC_RANKER) {  
    base.queryType = 'semantic';  
  
    const sc = (SEMANTIC_CONFIGURATION || ).trim(); if (sc) base.semanticConfiguration  
    = sc; base.queryLanguage = 'en-us'; if (SEMANTIC_USE_CAPTIONS) base.captions  
    = 'extractive|highlight-false'; if (SEMANTIC_USE_ANSWERS) base.answers =  
    'extractive|count-1'; } return base; } function mapSearchResult@ { const d  
    = r.document || r; // HARDENED: never return empty content if either field  
    is present const content = d[CONTENT_FIELD] ?? d.contents ?? d.content ?? ;  
  
  return {  
    id: d.id ?? d['@search.action'],
```

content,

title: d[TITLE\_FIELD] ?? d.title ?? null,

url: d[URL\_FIELD] ?? d.url ?? null,

metadata: d[METADATA\_FIELD] ?? d.metadata ?? {},

parentId: PARENT\_ID\_FIELD ? (d[PARENT\_ID\_FIELD] ?? null) : null,

score: r['@search.score'] ?? r.score,

rerankScore: null

};

}

// RRF helper

function rrfFuse(lists, k = 60) {

const scores = new Map();

for (const list of lists) {

list.forEach<sup>6)</sup>;

});

}

const itemById = new Map();

for (const list of lists) for (const it of list) if (!itemById.has(it.id)) itemById.set(it.id, it);

return Array.from(scores.entries()).sort<sup>7)</sup>;

}

async function searchOnce({ query, vector, fields, filter }) {

const opts = buildCommonOptions({ filter });

if (vector) {

const fieldList = Array.isArray(fields)

? fields

```

: [ (fields || VECTOR_FIELD) ].map(s => String(s || ).trim()).filter(Boolean);
opts.vectorSearchOptions = { queries: [{ kind: 'vector', vector,
kNearestNeighborsCount: TOP_K, fields: fieldList, }] }; } else { // BM25
path: explicitly tell Search which fields to match on opts.searchFields =
[CONTENT_FIELD, TITLE_FIELD].filter(Boolean); } const results = []; const
isAsyncIterable = (x) => x && typeof x[Symbol.asyncIterator] === 'function';
const run = async (o) => { const iter = searchClient.search(query || , o);

if (isAsyncIterable(iter)) {

for await (const r of iter) results.push(mapSearchResult@);

} else if (iter?.results && isAsyncIterable(iter.results)) {

for await (const r of iter.results) results.push(mapSearchResult@);

} else if (Array.isArray(iter?.results)) {

for (const r of iter.results) results.push(mapSearchResult@);

}

};

try {

await run(opts);

} catch (e) {

const msg = String(e?.message || ); const status = e?.statusCode ||
e?.response?.status; const selectProblem = (status === 400 && /Parameter
name:\s*\$select/i.test(msg)) || /Could not find a property named .* on type
'search\.\document'/i.test(msg); const overload =
/semanticPartialResponse|CapacityOverloaded/i.test(msg) || status === 206 ||
status === 503; if (selectProblem) { console.warn('Invalid $select detected
→ retrying without select'); const fallback = { ...opts }; delete
fallback.select; await run(fallback); } else if (USE_SEMANTIC_RANKER &&
overload) { console.warn('Semantic ranker overloaded → falling back to BM25
for this query. '); const fallback = { ...opts }; delete fallback.queryType;
delete fallback.semanticConfiguration; delete fallback.queryLanguage; delete
fallback.captions; delete fallback.answers; await run(fallback); } else {
throw e; } } // ☐ If BM25 (non-vector) still returned 0, fallback to REST
(we know it works) if (!vector && results.length === 0) { try { const
restHits = await searchOnceRest({ query, top: TOP_K }); if (restHits.length)
return restHits; } catch (e) { console.warn('REST fallback failed:',
e?.message || e); } } return results; } function collapseByParent(items,
maxPerParent = 3) { if (!PARENT_ID_FIELD) return items; const groups = new
Map(); for (const it of items) { const key = it.parentId || it.id; if
(!groups.has(key)) groups.set(key, []); groups.get(key).push(it); } const
collapsed = []; for (const arr of groups.values()) { arr.sort8};

```

```
collapsed.push(...arr.slice(0, maxPerParent)); } return collapsed.slice(0,
TOP_K); } async function retrieve(userQuery, { filter } = {}) { const t0 =
nowMs(); const lists = []; // Core keyword/semantic lists.push(await
searchOnce({ query: userQuery, filter })); // Vector on the raw query
(memoized) try { const qvec = await embedMemo(userQuery); lists.push(await
searchOnce({ query: , vector: qvec, filter }));

} catch (e) {

console.warn('vector(query) failed:', e?.response?.status || e?.message || e);

}

// Multi-query expansion (toggleable)

let expansions = [];

if (RETRIEVER_MULTIQUERY_N > 0) {

try {

expansions = await multiQueryExpand(userQuery, RETRIEVER_MULTIQUERY_N);

for (const q of expansions) {

lists.push(await searchOnce({ query: q, filter }));

}

} catch (e) {

console.warn('multiQueryExpand failed (continuing):', e?.response?.status || e?.message || e);

}

}

// HyDE (toggleable)

if (RETRIEVER_USE_HYDE) {

try {

const pseudo = await hydeDoc(userQuery);

const pvec = await embedMemo(pseudo);

lists.push(await searchOnce({ query: , vector: pvec, filter })); } catch (e) {
console.warn('HyDE failed (continuing):', e?.response?.status || e?.message
|| e); } } // Fuse & collapse const fused =
collapseByParent(rrfFuse(lists)); const latencyMs = nowMs() - t0; return {
```

```
items: fused.slice(0, TOP_K), latencyMs, expansions }); } async function
rerankWithLLM(query, items) { if (!LLM_RERANK || !items.length) return items;
const sys = { role: 'system', content: 'Rank passages by relevance to the
query. Return JSON array of {id, score} 0..100.' }; const passages =
items.map(p => ({ id: p.id, title: p.title || ,url:p.url|| , content: (p.content
|| ).slice(0, 1200) }));
```

```
const usr = { role: 'user', content: `Query: ${query}\nPassages:\n${JSON.stringify(passages, null,
2)}\nReturn only JSON array.` };
```

```
const res = await chat([sys, usr], { temperature: 0.0, max_tokens: 400 });
```

```
let scores = [];
```

```
try { scores = JSON.parse(res.choices[0].message.content); } catch {}
```

```
const byId = new Map(scores.map(s => [String(s.id), Number(s.score) || 0]));
```

```
for (const it of items) it.rerankScore = byId.get(String(it.id)) ?? null;
```

```
items.sort9;
```

```
return items;
```

```
}
```

```
function toCitableChunks(items) {
```

```
return items.map10;
```

```
}
```

```
async function synthesizeAnswer(userQuery, chunks) {
```

```
const sys = { role: 'system', content:
```

```
`You answer using only the provided sources. Cite like [1], [2] inline.
```

```
If unsure or missing info, say you don't know.
```

```
Return a JSON object: { "answer": string, "citations": [{"key": "[n]", "id": "...", "title": "...", "url": "..."}]
}. `};
```

```
const usr = { role: 'user', content: `Question:\n${userQuery}\n\nSources:\n${JSON.stringify(chunks,
null, 2)} ` };
```

```
const data = await chat([sys, usr], { temperature: TEMPERATURE, max_tokens:
MAX_TOKENS_ANSWER });
```

```
const raw = data.choices[0].message.content;
```

```
let parsed;  
  
try { parsed = JSON.parse(raw); } catch {  
  
parsed = { answer: raw, citations: chunks.map(c => ({ key: c.key, id: c.id, title: c.title, url: c.url }));  
  
}  
  
parsed.citations = parsed.citations || chunks.map(c => ({ key: c.key, id: c.id, title: c.title, url: c.url }));  
  
return { ...parsed, usage: data.usage || null };  
  
}
```

```
async function answerQuestion(userQuery, options = {}) {  
  
const retrieval = await retrieve(userQuery, options);  
  
let items = retrieval.items;  
  
items = await rerankWithLLM(userQuery, items);  
  
  
  
const topChunks = toCitableChunks(items.slice(0, TOP_K));  
  
const t0 = nowMs();  
  
const synthesis = await synthesizeAnswer(userQuery, topChunks);  
  
const genLatencyMs = nowMs() - t0;  
  
  
  
return {  
  
answer: synthesis.answer,  
  
citations: synthesis.citations,  
  
retrieved: items,  
  
metrics: {  
  
retrievalLatencyMs: retrieval.latencyMs,  
  
generationLatencyMs: genLatencyMs,  
  
expansions: retrieval.expansions,  
  
usage: synthesis.usage || null  
  
}
```

```
};
```

```
}
```

```
module.exports = {
```

```
  answerQuestion,
```

```
  _internals: { retrieve, rerankWithLLM, toCitableChunks, embed, embedMemo, chat,  
    multiQueryExpand, hydeDoc, searchOnce }
```

```
};
```

```
package.json
```

```
{
```

```
  "name": "openai-bot",
```

```
  "version": "1.0.0",
```

```
  "description": "Testing openai bot",
```

```
  "author": "Generated using Microsoft Bot Builder Yeoman generator v4.22.1",
```

```
  "license": "MIT",
```

```
  "main": "index.js",
```

```
  "scripts": {
```

```
    "start": "node ./index.js",
```

```
    "watch": "nodemon ./index.js",
```

```
    "lint": "eslint .",
```

```
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
```

```
  },
```

```
  "repository": {
```

```
    "type": "git",
```

```
    "url": "https://github.com"
```

```
  },
```

```
  "dependencies": {
```

```

"@azure/search-documents": "^12.1.0",
"axios": "^1.5.0",
"botbuilder": "~4.22.1",
"dotenv": "~8.2.0",
"mammoth": "^1.10.0",
"restify": "~11.1.0"
},
"devDependencies": {
"eslint": "^7.0.0",
"eslint-config-standard": "^14.1.1",
"eslint-plugin-import": "^2.20.2",
"eslint-plugin-node": "^11.1.0",
"eslint-plugin-promise": "^4.2.1",
"eslint-plugin-standard": "^4.0.1",
"nodemon": "^2.0.4"
},
"engines": { "node": ">=20 <21" }
}

```

1)

```

a,b)⇒a-b); const p = (x) ⇒ lat[Math.floor((lat.length-1)*x)]; const summary = { q, n, concurrency:
conc, p50: p(0.50), p90: p(0.90), p95: p(0.95), p99: p(0.99), errors: results.filter(r ⇒ !r.ok).length,
node: process.version, cpu: os.cpus()[0]?.model || 'unknown' }; console.log(JSON.stringify(summary,
null, 2)

```

2)

```

l, i) ⇒ { try { return JSON.parse(l); } catch { return { id: `row${i+1}`, question: l }; } }; } function
recallAtK(retrieved, refs, k = K) { if (!Array.isArray(refs) || !refs.length) return null; const ids = new
Set(retrieved.slice(0, k).map(r ⇒ r.id)

```

3)

```

s,c)⇒ s + (c.present?1:0), 0) / checks.length; return { coverage: pct, checks }; } async function
llmJudge(row, result) { // Optional LLM-as-a-judge: groundedness, relevance, completeness (0..5) //
Uses your chat deployment via openaiService._internals.chat const { chat } = _internals; const sys =
{ role: 'system', content: [ 'You are scoring an answer for a Retrieval-Augmented Generation (RAG)
system.', 'Return a JSON object with integer scores 0..5: {groundedness, relevance, completeness}
and a one-sentence "notes".', 'Definitions:', '- groundedness: The answer is explicitly supported by the
provided citations/passages. Penalize hallucinations.', '- relevance: The answer addresses the user

```

```

question (on-topic).', '- completeness: The answer covers the key points needed for a useful
response.' ].join('\n') }; const payload = { role: 'user', content: [ `Question: ${row.question}`,
`Answer: ${result.answer}`, `Citations: ${JSON.stringify(result.citations, null, 2)}`, `Top Passages:
${JSON.stringify(result.retrieved.slice(0,
K).map(p=>({id:p.id,parentId:p.parentId,title:p.title,content:(p.content||).slice(0,1000)})),
null, 2)}`, 'Return ONLY valid JSON.' ].join('\n\n') }; try { const data =
await chat([sys, payload], { temperature: 0.0, max_tokens: 200 }); const txt
= data.choices?.[0]?.message?.content?.trim() || '{}'; const obj =
JSON.parse(txt); const g = Math.max(0, Math.min(5,
Number(obj.groundedness)||0)); const r = Math.max(0, Math.min(5,
Number(obj.relevance)||0)); const c = Math.max(0, Math.min(5,
Number(obj.completeness)||0)); return { groundedness: g, relevance: r,
completeness: c, notes: obj.notes || }; } catch (e) { return { groundedness: null,
relevance: null, completeness: null, error: String(e) }; } } function mean(arr) { const xs = arr.filter(x
=> typeof x === 'number' && Number.isFinite(x
4)
a,b=>a+b,0)/xs.length; } // ----- Main ----- async function main() { const datasetPath =
path.join(__dirname, 'dataset.json!'); if (!fs.existsSync(datasetPath
5)
s||).length / 4); } function chunkText(text, { chunkTokens = 700,
overlapTokens = 100 } = {}) { const tokens = approxTokenLen(text); const
estCharsPerTok = (text.length || 1) / Math.max(tokens, 1); const chunkChars =
Math.floor(chunkTokens * estCharsPerTok); const overlapChars =
Math.floor(overlapTokens * estCharsPerTok); const chunks = []; let start = 0;
while (start < text.length) { const end = Math.min(start + chunkChars,
text.length); chunks.push(text.slice(start, end)); start = end -
overlapChars; if (start < 0) start = 0; if (end === text.length) break; }
return chunks; } async function embed(text) { const url =
`${OPENAI_ENDPOINT}/openai/deployments/${OPENAI_EMBEDDING_DEPLOYMENT}/embeddi
ngs?api-version=${OPENAI_API_VERSION}`; const { data } = await
axios.post(url, { input: text }, { headers: { 'api-key': OPENAI_API_KEY,
'Content-Type': 'application/json' } }); return data.data[0].embedding; }
async function ensureIndex() { const definition = { name: SEARCH_INDEX,
fields: [ { name: 'id', type: 'Edm.String', key: true, filterable: false,
sortable: false, facetable: false }, { name: TITLE_FIELD, type: 'Edm.String',
searchable: true }, { name: URL_FIELD, type: 'Edm.String', searchable: false,
filterable: true }, { name: CONTENT_FIELD, type: 'Edm.String', searchable:
true }, { name: METADATA_FIELD, type: 'Edm.ComplexType', fields: [ { name:
'source', type: 'Edm.String', filterable: true, facetable: true }, { name:
'category', type: 'Edm.String', filterable: true, facetable: true }, { name:
'tags', type: 'Collection(Edm.String)', filterable: true, facetable: true }
]}, { name: PARENT_ID_FIELD, type: 'Edm.String', searchable: false,
filterable: true }, { name: VECTOR_FIELD, type: 'Collection(Edm.Single)',
searchable: true, filterable: false, sortable: false, facetable: false,
vectorSearchDimensions: 3072, // text-embedding-3-large
vectorSearchProfileName: 'vdb' } ], semanticSearch: {
defaultConfigurationName: (process.env.SEMANTIC_CONFIGURATION ||
'default').trim(), configurations: [ { name:
(process.env.SEMANTIC_CONFIGURATION || 'default').trim(), prioritizedFields:
{ titleField: { name: TITLE_FIELD }, prioritizedContentFields: [{ name:
CONTENT_FIELD }]} } ] }, vectorSearch: { algorithms: [{ name: 'hnsf', kind:

```

```
'hsw' }], profiles: [{ name: 'vdb', algorithmConfigurationName: 'hsw' }] }
}; try { await indexClient.getIndex(SEARCH_INDEX); console.log('Index
exists. Updating (if schema changed)...'); await
indexClient.createOrUpdateIndex(definition); } catch { console.log('Creating
index...'); await indexClient.createIndex(definition); } } async function
readDocs(dir) { if (!fs.existsSync(dir)) return []; const files =
fs.readdirSync(dir); const docs = []; for (const f of files) { const full =
path.join(dir, f); const ext = path.extname(f).toLowerCase(); let text =
null; if (ext === '.docx') { // Convert Word (DOCX) → plain text const
buffer = fs.readFileSync(full); const { value } = await
mammoth.extractRawText({ buffer }); text = (value || ).trim(); } else if
(^(.txt|md|markdown)$/i.test(f
6)
item, idx) => { const prev = scores.get(item.id) || 0; scores.set(item.id, prev + 1 / (k + idx + 1
7)
a,b=>b[1]-a[1]).map((id) => itemById.get(id
8)
a, b) => (b.rerankScore ?? b.score ?? 0) - (a.rerankScore ?? a.score ?? 0
9)
a, b) => (b.rerankScore ?? 0) - (a.rerankScore ?? 0
10)
it, idx) => ({ key: `[${idx + 1}]`, id: it.id, title: it.title || `Doc ${idx + 1}`, url: it.url || null, content:
(it.content || "").slice(0, 2000) }
```